IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 22, 2019, accepted May 27, 2019, date of publication June 5, 2019, date of current version June 19, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2920947

Performance Evaluation of Data Race Detection Based on Thread Sharing Analysis With Different Granularities: An Empirical Study

LILI BO[1], SHUJUAN JIANG[1,2], JUNYAN QIAN[3], RONGCUN WANG[1,2], AND YAFEI YAO[1,2]

[1]School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China
[2]Mine Digitization Engineering Research Center of Ministry of Education of the People's Republic of China, Xuzhou 221116, China
[3]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China

Corresponding author: Shujuan Jiang (shjjiang@cumt.edu.cn)

This work was supported by the Fundamental Research Funds for the Central Universities under Grant 2017XKZD03.

ABSTRACT Thread Sharing Analysis (TSA) plays an important role in concurrent program testing. Providing a TSA to a data race detector may speed up the runtime logging and improve the performance of data race detection. In this paper, we focus on the empirical study of the performance of data race detection based on TSA with different granularities. First, three granularities are considered, including object, field and ''field + array element''. Then, an empirical study is conducted to evaluate the performance of data race detection based on three dynamic TSA approaches. The results show that data race detection based on the TSA with ''field + array element'' granularity outperforms those with object and field granularities.

INDEX TERMS Software testing, concurrent program testing, thread sharing analysis (TSA), data race detection, dynamic analysis

I. INTRODUCTION

Concurrent programs have gradually become popular to fully utilize multicore CPUs. Data race, a type of concurrency bug, is difficult to expose, to detect and to fix. A data race occurs when two or more different threads concurrently access (i.e., read-write, write-read, or write-write) a shared variable without any synchronization mechanisms.

Thread Sharing Analysis (TSA) aims to determine whether a program statement can read or write thread-shared data. It is defined as the problem of locating thread-shared data accesses, i.e., shared access points (SAP). TSA is the basis of concurrent program comprehension, compiler optimization and software testing. For example, if an access is not to the shared data, a race detector can ignore this access and save much more analysis time.

A great number of data race detection techniques have been developed to find data races. Commonly, there are two data race detection models: happens-before relation [1], [2] and lock-sets [3], [4]. In terms of program execution, data race detection techniques can be divided into static detection [5], [6] and dynamic detection [7], [8]. Static techniques extract program information by analyzing control flow, data flow and synchronization effects of the current program. Dynamic techniques execute target programs and try to find data races based on the execution traces.

However, most of data race detection techniques are based on escape analysis which is a coarse-grained thread shared analysis approach [9], [10]. Escape analysis is a classic thread shared analysis approach. It identifies a shared object by checking if the object escapes from a thread or a method creating it. However, an object is escaped does not mean that all the data in the object are shared. Recently, Huang pointed out some limitations of escape analysis and proposed a field-sensitive, object-insensitive dynamic-TSA algorithm [11]. This algorithm addresses the runtime overhead problem. It is based on a location-based approach and tracks memory accesses at each program location at most twice. If the same field or array object is accessed by two different threads from two different program locations, or twice from the same program location, with at least one write, the field or array object is marked as shared. All statements accessing it are marked SAPs. Therefore, dynamic-TSA algorithm is on the field granularity. However, for efficiency, the dynamic-TSA algorithm ignores different array elements, which may produce many false positives. These non-SAPs can increase

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Pietrantuono.

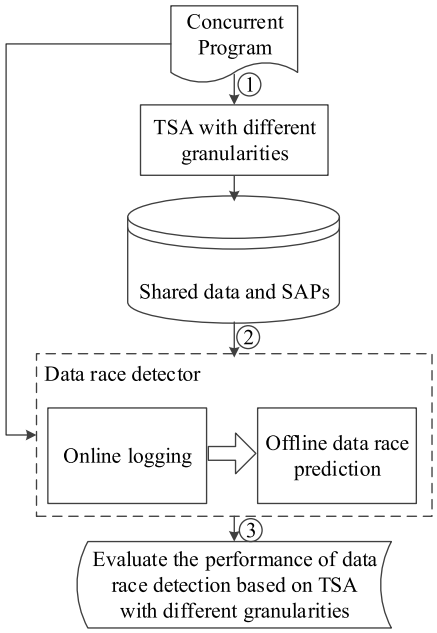


FIGURE 1. Overview of our approach.

the runtime overhead for data race detection. How is the performance of data race detection with the increase of analysis granularities?

Based on the above questions, we conduct an empirical study to evaluate the performance of data race detection based on TSA with different granularities, including object, field and "field + array element". We use a data race detector called RVPredict, which is a maximal sound predictive race detection tool [12]. Our experiments are conducted on 12 widely used java multi-threaded programs. The results show that data race detection based on the TSA with "field + array element" granularity obtains the best performance.

The remainder of this paper is organized as follows. Our approach is described in Section II. Section III presents an empirical study to show its validity. Section IV summarizes the related work. Finally, we draw conclusions of this paper in Section V.

II. OUR APPROACH

A. OVERVIEW

Data race detection can benefit a lot from TSA. Given the shared data and shared access points, data race detectors can eliminate a lot of instrumentations on concurrent programs under test, saving online logging time. In addition, thread shared analysis may affect the result of effectiveness of data race detector. In this paper, we evaluate the performance of data race detection based on TSA with different granularities, which mainly considers the time benefit from TSA.

Fig. 1 presents the overview of our approach. It consists of three steps: thread sharing analysis, data race detection and performance evaluation. In the first step, we identify shared data and shared access points in the concurrent program by using three TSA approaches with different granularities (i.e., objects, fields, fields + array elements). In the second step, data race detectors analyze the concurrent program with the results of three TSA approaches, respectively. Take RVPredit for example, it first logs the shared access events online and then predicts data races offline. The final step is performance evaluation. We evaluate the performance of data race detection based on different grained TSA in terms of time overhead.

B. TSA WITH DIFFERENT GRANULARITIES

TSA is a concurrent testing methodology, which can be described from two perspectives, i.e., static analysis and dynamic analysis. Given a concurrent program P, static TSA first constructs an Information Flow Group (IFG) or a Call Graph (CG) for the whole program, and then traverses the access statements reachable from each static thread to identify SAPs. If an object (field) is accessed by two or more static threads, with at least one write access, this object (field) is considered to be shared. The corresponding statements on the shared object (field) are marked as SAPs. Unlike static TSA, dynamic TSA tracks the thread memory accesses information during the execution of program P. If the same object (field) is accessed by two different threads from two different program locations, or twice from the same program location, with at least one write, the object (field) is considered to be shared. The statements which access this object (field) are marked as SAPs.

In this paper, we employ TSA with three granularities, i.e., object, field and "field + array element". The details are described as follows.

1). TSA with object granularity. Standard escape analysis is a TSA approach which checks if an object escapes a thread or a method creating it. If an object is thread-escaped, escape analysis considers that all data associated with the object are shared. Moreover, all static variables as well as any object reachable from the static variables are thread-escaped. Therefore, escape analysis is a coarse-grained TSA and it is on object granularity. In the empirical study, we use dynamic escape analysis (DEA) as the TSA approach with object granularity.

2). TSA with field granularity. Dynamic-TSA proposed by Huang is a TSA approach which identifies shared fields and shared array objects. It is a location-based, field-sensitive, object-insensitive analysis approach. Only if the same field is accessed by two different threads from two different program locations, or twice from the same program location, with at least one write, the field is considered to be shared. For each shared field or shared array object, all statements accessing it are classified as shared access points. To pursuit efficiency, for any program location, it analyzes at most two memory accesses performed at that location and ignores different array indexes. In the empirical study, we use dynamic-TSA as the TSA approach with field and array object granularity.

3). TSA with "field + array element" granularity. Identifying shared array elements accurately can accelerate data

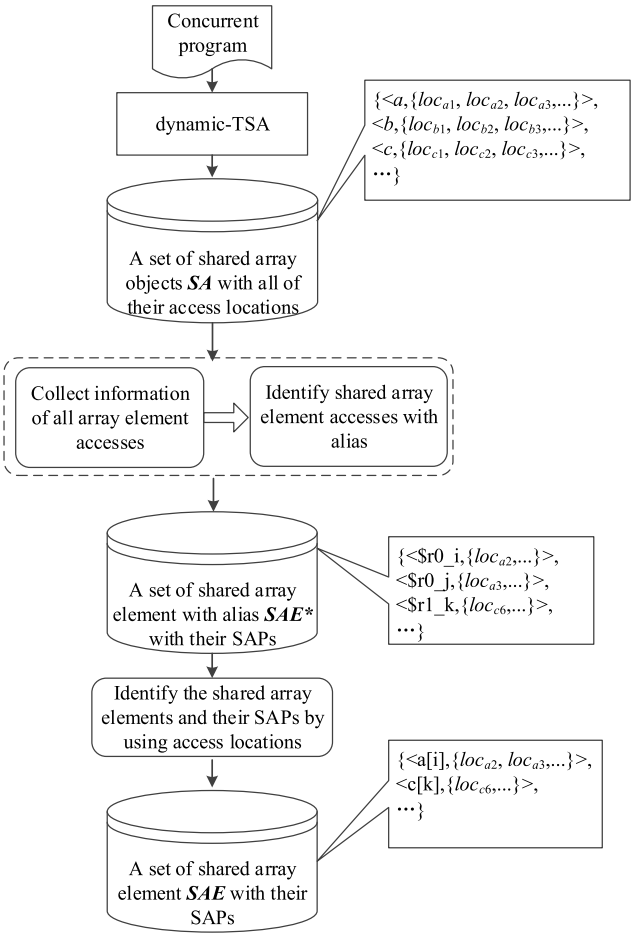


FIGURE 2. Overview of $DTSA_{FAE}$.

race detection and data race fix. This is more obvious in the case of "hot-path", such as loop accesses of an array object. However, identifying shared array element accesses is difficult because of alias problem. None of TSA approaches distinguish between different array elements. In this paper, we design $DTSA_{FAE}$ (Dynamic TSA with "Field + Array Element" granularity) to identify shared fields and shared array elements.

The process of shared array elements identification is shown in Fig. 2. First, we use dynamic-TSA to analyze a concurrent program, which can obtain a set of shared array objects *SA* with all of their access locations. Next, we collect information of all array element accesses. As soon as an array element access happens, we check if the current element is shared. The current element is shared when it is accessed by two different threads, with at least one write. A challenge is that the names of the shared array elements are still unknown. Then, we find the true name of shared array elements by using access locations. For example, we can get $<\$r0:\{loc_{a2}, loc_{a3},\ldots\}>$ from *SAE**. Besides, we can find from SA that array object a is accessed in program location loc_{a2} and loc_{a3}. Therefore, we know that $\$r0$ is the alias of array object a. Finally, we obtain the shared array elements and their SAPs.

Overall, thread sharing analysis can directly impact on the performance of data race detection. Given accurate SAPs, a data race detector can eliminate a lot of instrumentations on non-SAPs, which can save much time and memory space.

C. AN ILLUSTRATION

We use an example in Fig. 3 to show the TSA results with three granularities.

As described in Fig. 3, there are in total two threads (i.e., main thread and Thread1) accessing a shared object s with two instance fields x, y, a static field *str* and an array reference a. To indicate the access type, a simple annotation with different color is added after the corresponding statement. For example, R(x) means reading x, and W(x) means writing x.

The analysis results obtained by thread sharing analysis approaches with three granularities are shown in Table 1. Columns 1-6 respectively represent granularity, approach, shared data, SAPs, the number of SAPs and data races detected by RVPredict. In this paper, SAP is represented by the line number. Data race is represented in the form of "(s:< line1, line2>)", where s indicates the shared data, line1 and line2 indicate statement positions of data race.

Row 2 in Table 1 presents the analysis results obtained by escape analysis which is on object granularity. Classical escape analysis identifies the instance object s as shared, because it escapes the thread (i.e., main thread) which creates it. Moreover, escape analysis considers all the fields of s (including x, y, *str*, a) and all accesses to the fields as shared since they are the accesses to the shared object s. In addition, as the shared field x can be written concurrently by main thread and Thread1, the data race on x can be easily detected by RVPredict. However, it will miss the data race on a[0] because escape analysis mistakenly identifies b as thread-local.

Row 3 in Table 1 presents the analysis results obtained by dynamic-TSA which is on field granularity. Dynamic-TSA identifies field *str* as thread-local since *str* is only accessed by main thread. Besides, dynamic-TSA identifies field y as thread-local since y is only read after initialization and it is immutable. Hence, the statements of accesses to *str* (line 13) and y (line 14) are non-SAPs. Furthermore, as b is an alias of a, dynamic-TSA identifies all the accesses to the array object a as SAPs, i.e., lines 15, 16, 23, 25. However, dynamic-TSA cannot distinguish between different array elements. Therefore, although RVPredict can detect the data races (<16,23>) and (<23,16>), the shared array element is unknown.

Row 4 in Table 1 presents the analysis results obtained by $DTSA_{FAE}$ which is on "field + array element" granularity. a[1] is only read after initialization. Thus, it is thread-local. The statements in line 15 and line 25 are non-SAPs. Finally, RVPredict can report the data races occur on the shared fields x and shared array element a[0], i.e., (x:<6,12>), (x:<12,6>), (a[0]:<16,23>), (a[0]:<23,16>). The results

```
1  public class Shared{
2     int x, y;
3     static String str;
4     int a[] = new int[2];
5     public int incX()
6        {return y+(x++);}
          //R(y), R(x), W(x)
7  }
```

```
8  public class Test{
9     public static void main(String[] args){
10       Shared s = new Shared();
11       new Thread1(s).start();
12       s.x = 5;     //W(x)
13       s.str = "a static string";     //W(str)
14       print("y = " + y);      //R(y)
15       int [] b = s.a;   //R(a)
16       b[0] = 1;    //W(b[0])
17    }
```

```
18  static class Thread1 extends Thread{
19     Shared s;
20     public Thread1 (Shared s)
21        {this.s = s;}
22     public void run(){
23        s.a[0] = 2;      //R(a), W(a[0])
24        print(s.incX());
25        print("a[1] = " + s.a[1]);   //R(a), R(a[1])
26     }
27  }
28 }
```

FIGURE 3. An example program.

TABLE 1. Analysis results of the program nn fig. 3.

Granularity	Approach	Shared data	SAPs	#SAPs	Data races
object	escape analysis	*s*, *x*, *y*, *str*, *a*	6,12,13,14,15,23,25	7	(*x*:<6,12>)(*x*:<12,6>)
field	dynamic-TSA	*x*, *a*	6,12,15,16,23,25	6	(*x*:<6,12>)(*x*:<12,6>) (<16,23>)(<23,16>)
field+array element	$DTSA_{FAE}$	*x*, *a*[0]	6,12,16,23	4	(*x*:<6,12>)(*x*:<12,6>) (*a*[0]:<16,23>)(*a*[0]:<23,16>)

TABLE 2. Description of experimental subjects.

Programs	LOC	Scale	#Threads	Native(ms)	Description
AirlineTicket	95	Small	11	18	Airline ticket program
Account	155	Small	3	60	Bank account program
Shop	280	Small	4	57	Supplier-Customer system
BoundedBuffer	536	Small	4	159	Bounded buffer
StringBuffer	1320	Middle	3	82	String buffer
RayTracer	1924	Middle	2	109	Measure the performance of a 3D ray tracer
Cache4j	3897	Middle	4	105	Cache for Java objects
ArrayList	5866	Middle	3	100	List for storing array elements
SpecJBB-2005	18K	Large	3	1585	Evaluate the performance of server side Java
Avrora	93K	Large	9	3471	Discrete event simulator of a sensor network
Sunflow	109K	Large	13	933	Render a set of images using ray tracing
Lusearch	410K	Large	8	794	Text search of keywords

imply that, with $DTSA_{FAE}$, we can know the conflict data as well as the responding locations.

Given precise SAPs to a data race detector, it can eliminate the instrumentations on those non-SAPs, which reduces the logging overhead and improves the runtime performance. From Table 1, we can see, three approaches that work on object, field and "field + array element" granularities mark seven, six and four SAPs, respectively. $DTSA_{FAE}$ marks the least number of SAPs, which is two less than dynamic-TSA and three less than escape analysis. Furthermore, this does not reduce the number of data races reported by RVPredict. Compared with RVPredict based on dynamic-TSA and escape analysis, RVPredict based on $DTSA_{FAE}$ can report complete data races.

III. EMPIRICAL STUDY

In this section, we conduct an empirical study on a suit of widely used multi-threaded programs to investigate the validity of our approach. First, we describe the experimental subjects and the experimental design. Then, we present the results and analyze them in details.

A. EXPERIMENTAL SUBJECTS

We select 12 multi-threaded Java programs for our evaluation. 10 programs are from five common benchmark suits, i.e., IBM Contest benchmark suite [13] (Account, Shop), an open library from Suns JDK 1.4.2 (StringBuffer, ArrayList), Java Parallel Grande (JPG) benchmark suite [14] (RayTracer), SIR [15] (AirlineTicket, BoundedBuffer) and Dacapo (Dacapo-9.12-bach) [16] (Avrora, Sunflow, Lusearch). Cache4j and SpecJBB-2005 are from reference [17]. These multi-threaded Java programs are widely used in data race detection.

The details of all subjects are shown in Table 2, where columns 1-6 represent program name, size in lines of code, scale, number of dynamic threads, native running time and a simple description. The subject programs are sorted in

TABLE 3. Four scenes about TSA.

Scene	Granularity	Approach
S1	no-TSA	without TSA
S2	object	DEA
S3	field	dynamic-TSA
S4	field + array element	$DTSA_{FAE}$

increasing order by their scale. Our experimental subjects contain small (LOC < 1000), middle (1000 < LOC < 10,000) and large (LOC > 10,000) concurrent programs. Besides, ''Native(ms)'' (column 5) indicates the running time without any analysis tools.

B. EXPERIMENTAL DESIGN

Our experiments are conducted on a four-core Intel i7 3.06GHz machine with Java HotSpot 1.7 and 8GB memory running Ubuntu-14.04. We compare the performance of data race detection based on three dynamic TSA approaches with different granularities, which are described in Section II.B. Both TSA approaches with field granularity and with ''field + array element'' granularity are modified from dynamic-TSA proposed by Jeff Huang.

The data race detection function in our experiments is performed on RVPredict, a recent, source available data race detection tool. RVPredict is a maximal sound predictive race detector. It contains two steps: trace collection and predictive race analysis. In trace collection, a sequentially consistent trace which contains shared data accesses, thread synchronizations, and branch events is logged after static instrumentation. In predictive race analysis, a constraint is constructed for every conflicting operation pair (COP), and then it is solved through an SMT solver. For scalability and practicality, RVPredict employs a windowing strategy [18]. Large traces are divided into a sequence of fixed-size windows. After that, RVPredict performs race analysis on each window separately.

To evaluate the performance of data race detection based on TSA, we set four different scenes that are listed in Table 3. The analysis granularity increases by rows. Column 2 and column 3 represent the granularity and the corresponding approach we used in our implementation. In special, ''S1'' means to execute data race detection without any TSA approaches, i.e., tracking all accesses. The last three scenes which are introduced in Section II.B indicate to track the shared access points obtained from DEA, dynamic-TSA and $DTSA_{FAE}$, respectively. As three TSA approaches are dynamic, all data are averaged over 10 runs.

C. EXPERIMENTAL RESULTS AND ANALYSIS

1) SIZE OF TRACE

RVPredict collects an execution trace that contains synchronization operations and shared memory accesses. Therefore, the size of trace becomes a factor which affects the performance of data race detection. Table 4 lists, for each program, program name (column 1) and size of trace that RVPredict collected in four scenes (column 2-5). Column 2 shows the size of trace collected by RVPredict without any TSA approach. Columns 3-5 show the size of trace collected by RVPredict based on TSA with granularities on object, field and ''field + array element'', respectively.

TABLE 4. Size of trace collected by RVPredict in four scenes.

Programs	S1	S2	S3	S4
AirlineTicket	100	100	95	90
Account	126	82	65	50
Shop	2120	**397**	**477**	379
BoundedBuffer	2228	1943	1337	1116
StringBuffer	81	70	60	36
RayTracer	23836	22531	22524	19043
Cache4j	718256	705681	683733	653114
ArrayList	303	**157**	**164**	88
SpecJBB-2005	479413	**267817**	**273282**	132317
Avrora	1404160146	**502142545**	**616669987**	578794545
Sunflow	505891688	-	281123465	208934640
Lusearch	322827448	**175678647**	**301655570**	162360628

As shown in Table 4, TSA can reduce the size of trace significantly. Compared with tracking all accesses (S1), the size of trace can be reduced by 30%, 33% and 50% on average after using TSA approach with the granularities on average after using TSA approach with the granularities on object field and ''field + array element'', respectively. There are five programs in which the size of trace collected by RVPredict based on TSA with field granularity is larger than that with object granularity. They are Shop, ArrayList, SpecJBB-2005, Avrora and Lusearch, which are bold in Table 4. The reason is that, there are shared array element accesses in these programs. Dynamic escape analysis that is at object granularity does not work for array indexing accesses. Both dynamic-TSA and $DTSA_{FAE}$ track array element accesses. The difference is that, dynamic-TSA reports all statements accessing the shared array object as SAPs and $DTSA_{FAE}$ can distinguish between different array elements, which only reports the shared array elements.

In addition, we find that, for Sunflow, we cannot get its size of trace at the object granularity because DEA runs out of memory for it. This also implies that dynamic-TSA and $DTSA_{FAE}$ are efficient than DEA.

2) EFFICIENCY

To further explore the performance of data race detection based on TSA with different granularities, we compared the runtime overhead of RVPredict in four scenes.

The comparison results for nine small and middle-scale programs are displayed in Fig. 4, where x-axis indicates three phrases of dynamic data race detection (i.e., instrumentation, logging and prediction), and y-axis indicates the time cost in four scenes. For example, for most programs, large amount of time is used for instrumentation. But for BoundedBuffer, the time consumed for prediction is much more than that for

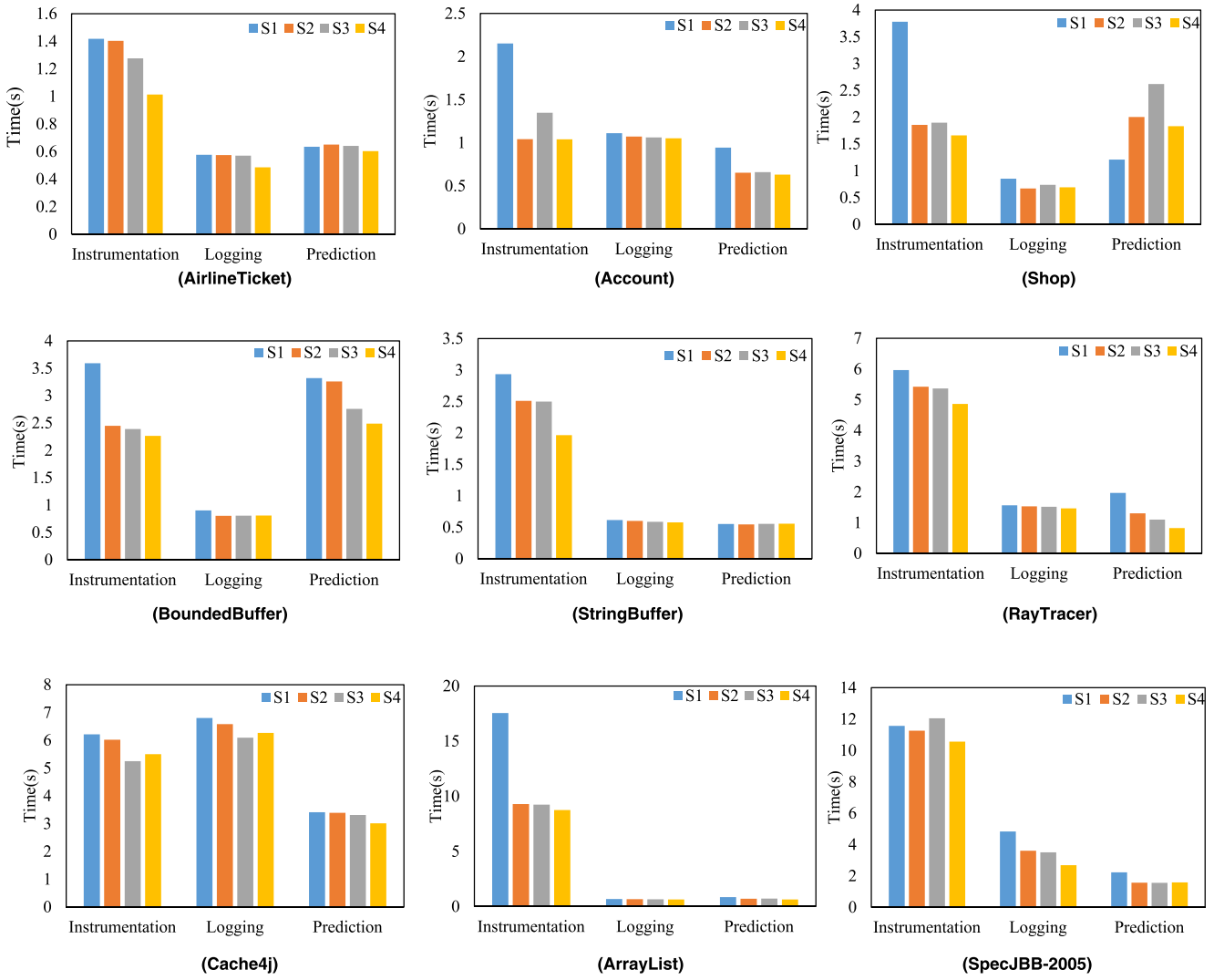


FIGURE 4. Runtime overhead of RVPredict in four scenes for nine small-scale and middle-scale programs.

logging. The reason is that, there are 13 data races needed to be checked in BoundedBuffer, which is more than that in other programs. For Cache4j, the time cost in three phrases is uniform because the trace collected by RVPredict is much longer than others.

Moreover, we find that, the instrumentation time and the logging time are related to the size of trace in Table 4 as RVPredict instruments on these events and records them. For example, for Shop, the size of trace collected by RVPredict at field granularity is 80 more than that at object granularity, then RVPredict consumes 0.04s and 0.07s more time for instrumentation and logging shared events, respectively.

In addition, we summarize the runtime overhead of RVPredict in four scenes for three large programs, which is shown in Table 5. Table 5 lists, for each large program, program name (column 1), runtime in three phases at four different granularities (columns 2-5), and the corresponding reduction compared with that in S1 (columns 6-8). Rows 1-6 describe the instrumentation time of RVPredict in four scenes. Rows 7-12 describe the online logging time of RVPredict in four scenes. Rows 13-18 describe the offline prediction time of RVPredict in four scenes. "−" in Table 5 indicates that we cannot get the SAPs identified by DEA as it analyzes Sunflow over one hour and runs out of memory.

As shown in Table 5, TSA can reduce the logging overhead significantly. It reduces the instrumentation time, online logging time and prediction time by 26.52%, 52.95%, 56.02% and 34.76%, 63.91%, 59.91% at field and "field + array element" granularity, respectively. Although RVPredict in S2 reduces more time than that in S3 and S4 for Avrora and Lusearch, it is unavailable for Sunflow. What's more, RVPredict based on the TSA approach worked at array element outperforms that based on object and field granularity.

From Fig. 4 and Table 5, we can conclude that, for most programs, the larger the size of trace is, the more time it consumes for offline prediction, such as AirlineTicket,

TABLE 5. Runtime overhead Of RVPredict in four scenes for three large programs.

Programs	Instrumentation Time(s)				Reduction		
	S1	S2	S3	S4	S2	S3	S4
Avrora	19.15	13.56	15.98	16.97	29.19%	16.55%	11.37%
Sunflow	19.99	-	16.04	13.76	-	19.78%	31.15%
Lusearch	33.07	14.36	18.77	12.65	56.58%	43.24%	61.75%
Average						26.52%	34.76%
Programs	Online Logging Time(s)				Reduction		
	S1	S2	S3	S4	S2	S3	S4
Avrora	9730.12	2445.13	3835.92	3137.55	74.87%	60.58%	67.75%
Sunflow	2650.17	-	1498.12	1174.24	-	43.47%	55.69%
Lusearch	2004.23	679.34	906.06	635.48	66.10%	54.79%	68.29%
Average						52.95%	63.91%
Programs	Offline Prediction Time(s)				Reduction		
	S1	S2	S3	S4	S2	S3	S4
Avrora	2450.79	743.86	1163.74	906.47	69.65%	52.52%	63.01%
Sunflow	831.58	-	451.16	503.42	-	45.75%	39.46%
Lusearch	514.06	172.65	155.29	116.97	66.41%	69.79%	77.25%
Average						56.02%	59.91%

TABLE 6. Total time for data race detection with three granularities.

Programs	Thread Sharing Analysis (TSA)			Data Race Detection (RVPredict)			Total		
	DEA(s)	dynamic-TSA(s)	$DTSA_{FAE}$(s)	S2(s)	S3(s)	S4(s)	S2(s)	S3(s)	S4(s)
AirlineTicket	0.085	0.064	0.061	2.627	2.487	2.101	2.712	2.551	2.162
Account	0.565	0.56	0.563	2.765	2.761	2.721	3.33	3.321	3.284
Shop	0.079	0.073	0.084	4.527	4.253	4.182	4.606	4.326	4.266
BoundedBuffer	0.204	0.179	0.175	6.512	5.955	5.563	6.716	6.134	5.738
StringBuffer	0.089	0.085	0.085	3.658	3.642	3.102	3.747	3.727	3.187
RayTracer	0.184	0.173	0.158	8.264	7.985	7.153	8.448	8.158	7.311
Cache4j	1.693	1.513	1.551	15.993	14.654	14.781	17.686	16.167	16.332
ArrayList	0.177	0.157	0.156	10.669	10.619	10.033	10.846	10.776	10.189
SpecJBB-2005	526.23	150.15	118.02	16.42	16.60	14.82	542.65	166.75	132.84
Avrora	1876.41	86.14	306.28	3202.55	5015.64	4060.99	5078.96	5101.78	4367.27
Sunflow	3600	93.24	125.15	-	1965.31	1691.42	-	2058.55	1816.57
Lusearch	185.11	98.17	289.07	866.35	1080.12	765.09	1051.46	1178.29	1054.16

Account, Shop, BoundedBuffer, Avrora. However, for Lusearch, although the size of trace at object granularity is smaller than that at field granularity, the offline prediction time at object granularity is much more. The reason is that, the windowing strategy RVPredict employed divides the trace into a sequence of fixed-size windows and performs race analysis on each window separately. A shorter trace may increase the number of races in a window, which can consume more time for prediction.

To present an unbiased estimation, we calculate the total time of data race detection, which consists of the time of thread sharing analysis and the runtime overhead of RVPredict. The results of 12 programs are summarized in Table 6. Columns 2-4 report the time of DEA, dynamic-TSA and $DTSA_{FAE}$, respectively. Columns 5-7 report the time consumed for RVPredict based on TSA with three granularities (object, field and "field + array element"). Columns 8-10 represent the total time of data race detection.

The data in Table 6 show that, for small programs, both the time of TSA and the time of RVPredict reduce along with the increase of analysis granularity. For middle-scale and large programs, although RVPredict which based on the TSA with field granularity has more runtime overhead than that with object granularity, dynamic-TSA is much efficient than DEA. Obviously, DEA runs out of memory for Sunflow, making it unavailable for RVPredict. Therefore, in terms of total time, the performance of RVPredict based on the TSA with field granularity is comparable to that with object granularity.

Similarly, although TSA with "field + array element" granularity consumes more time to distinguish different array

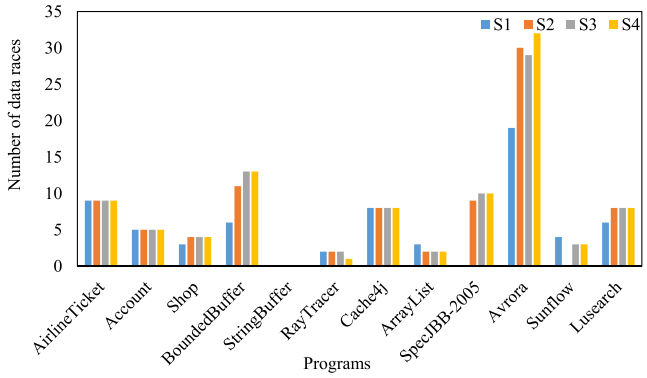


FIGURE 5. Number of data races detected by RVPredict in four scenes.

elements compared to that with field granularity, RVPredict based on TSA with ''field + array element'' granularity spends less time on online logging. The data in the Total columns also shows that RVPredict based on the TSA with ''field + array element'' granularity outperforms than that with field and object granularities.

3) EFFECTIVENESS

We evaluate the effectiveness of thread sharing analysis with different granularities in terms of the number of data races detected by RVPredict. The time threshold of offline prediction is set to one hour.

Fig. 5 shows, for each program, the number of data races detected by RVPredict in four scenes. The blue, red, green and purple bars represent RVPredict without TSA and that with DEA, dynamic-TSA as well as DTSA_{FAE}, respectively. For example, RVPredict detects no data race in StringBuffer in four scenes. For Sunflow, RVPredict with DEA detects no data race because DEA analyzes it out of time and memory. For BoundedBuffer, RVPredict with DTSA_{FAE} and dynamic-TSA detects two more data races than that with DEA and they are two real data races through our manual check. In addition, for Avrora, RVPredict with DTSA_{FAE} detects more data races than that with other thread sharing analysis approaches. Also, they are real data races.

However, for ArrayList and Sunflow, RVPredict without TSA detects one more data races than that with other three TSA approaches. This is probably for two reasons: the first is that dynamic analysis is unsound, which causes some SAPs that may involve data races not logged in the trace, and the second is there are more shared data accesses in a window of RVPredict.

Overall, the numbers of data races are similar among four scenes. This mainly attributes to the detection mechanism of RVPredict, which formulates race detection as a constraint solving problem. Concretely, it encodes the control flow and a minimal set of feasibility constraints as a group of first-order logic formulas, and then solves them by an SMT solver. Therefore, although there are many potential data races at the beginning, only the real data races are finally reported. Furthermore, the results also imply that fine-grained thread sharing analysis can scarcely reduce the number of data races, which enhances the availability of DTSA_{FAE} in practice.

D. THREATS TO VALIDITY

We find several threats to the validity of our experiments. They can be summarized into two aspects.

1) INTERNAL VALIDITY

There are two threats to internal validity. One is the performance evaluation criterion, the other is the fixed-size windowing strategy.

For the first one, the performance generally includes not only time cost but also space cost, which is not considered in our experiments. All experiments are conducted on a machine with 8G memory. The situation of running out of memory only occurs when we analyze Sunflow with DEA. In fact, the space cost can be shown by the size of trace which contains synchronization operations and shared memory accesses, because statements should be instrumented before them for further recording. Therefore, the logging overhead of both time and memory space will increase along with the length of trace.

For the second threat to internal validity, the effectiveness of data race detection with four different granularities mainly depends on the size of windows employed by RVPredict. RVPredict is sound and maximal. Soundness means every detected race is real, which eliminates false positives. Maximality means RVPredict does not miss any race that can be detected by any sound dynamic race detector based on the same trace. However, the number of detected data races may be affected by the number of shared data accesses in each window. For example, if two shared data accesses that involves a data race are divided into two windows, it will produce a false negative. As we can see from Table 4, the sizes of traces collected by RVPredic are different for a program after applying different TSA approaches. Therefore, a strategy of self-adapting size of windows may be helpful to further improve the effectiveness of data race detection.

2) EXTERNAL VALIDITY

Threats to external validity arise when selecting the TSA approaches and the experimental subjects.

In our experiments, we selected two dynamic thread sharing analysis approaches (i.e., DEA, and dynamic-TSA) and designed a dynamic DTSA_{FAE} approach to evaluate the performance of RVPredic. A natural character of dynamic algorithm is unsound, which may cause the recorded traces are different every time. To alleviate this problem, we run each program 10 times and take the average value.

Additionally, although our experimental subjects involve small, middle and large scale programs and they are widely evaluated in data race detection, we cannot ensure that our conclusion still holds for all programs. Further generalizability requires more programs that are with different characters and come from various domains.

IV. RELATED WORK

To date, many researchers have proposed a large number of data race detection techniques. How to improve the performance of data race detection has gradually become a hot topic in concurrent software testing. In this section, we first summarized the related work on TSA and its application on data race detection. Then, we reviewed the empirical studies which are relevant to our work.

Escape analysis is a classic TSA approach which has been widely applied for many years. It focuses on the object granularity. Generally, escape analysis can be divided into static escape analysis and dynamic escape analysis. Originally, Choi *et al.* [9], Whaley and Rinard [10] used static escape analysis for stack allocations and synchronization elimination. Later on, with static escape analysis, Naik *et al.* [5]–[19] implemented Chord and Jade to detect data race and deadlock, respectively. Besides, Halpert *et al.* [20] proposed to applied static escape analysis to lock allocation and achieved good performance. However, static escape analysis should construct inter-procedural information flow graph (IFG) for the whole program, and propagate shared nodes in the graph, which makes it difficult to scale to large programs. The experimental subjects in our empirical study involves programs with different scales, including small programs, middle-scaled programs and large systems. Therefore, for scalability, we use dynamic TSA approaches with three granularities to obtain SAPs.

Dynamic escape analysis checks when thread-private data become shared. As dynamic analysis has become the focus of multithreaded program analysis and concurrency bug detection, it is widely applied in data race detection [12]. For example, Nishiyama [21], Christiaens and Bosschere [22] used dynamic escape analysis (DEA) to improve the performance of race detection. Compared with first-shared-access-based DEA [4], [21]–[23], reachability-based DEA [22], [24], [25] has more advantage to be a sound filter for data race detection. Recent work introduced a lightweight data race detection Caper for production runs [25]. Caper was the first to utilize reachability-based DEA to prune the results of static data race detection, obtaining all true data races soundly from observed executions. In our empirical study, as ''object'' is selected to be an evaluation granularity, reachability-based DEA is implemented to obtain the SAPs.

Considering some limitations of escape analysis (e.g., false positives, false negatives and hard to scale to large programs), Huang [11] proposed two scalable TSA approaches, i.e., static-TSA and dynamic-TSA. Static-TSA was an object-sensitive, field-sensitive analysis approach. It leveraged call graph and points-to analysis to traverse the reachable field and array access statements to identify SAPs. It is simple, but less memory-demanding and more precise than static escape analysis. Dynamic-TSA was a location-based, field-sensitive, object-insensitive analysis approach. The experimental results showed that it could achieve a significant performance improvement over a precise dynamic escape analysis while maintaining close precision. In our empirical study, dynamic-TSA is implemented to obtain the SAPs since ''field'' is selected to be an evaluation granularity. Furthermore, our DTSA_{FAE} is designed based on dynamic-TSA. First, a set of shared array objects with all of their access locations are obtained by dynamic-TSA. Then, all the shared array elements and their SAPs are identified by utilizing the runtime information, i.e., program locations of the accessed array elements. Therefore, our DTSA_{FAE} is more precise than dynamic-TSA.

Dynamic analysis gets more popularity in many applications, such as data race detection [12], [26]–[28], atomicity violation detection [29]–[31], record and replay systems [32]–[35]. Researchers have surveyed data race detection techniques from various views [36]–[38]. Generally, data race detection techniques are based on two models: locksets and happens-before relation. Locksets-based techniques [4], [29], [39] check that whether accesses to the same shared data are guarded by the same lock via dynamic analysis. Happens-before relation based techniques define the partial order on all events of all threads in a system. If two or more threads access a shared variable and the accesses are concurrent, a data race bug on this variable may occur. However, locksets-based techniques may report many false positives and happens-before relation based techniques may miss some real races. Our empirical study adopts RVPredict to perform data race detection. RVPredict is a dynamic and maximal sound predictive race detector. It combines locksets and weaker happens-before relations to predict data races. In addition, we adapt the original RVPredict for our evaluation. The programs under test are first analyzed by TSA approaches with three different granularities, then the obtained results are inputted to RVPredict for predicting.

From the point of empirical studies, there are only a few empirical researches on concurrent software testing. Lu *et al.* [40] provide the first comprehensive real word concurrency bug characteristic study. They examine concurrency bug patterns, manifestation and fix strategies of 105 randomly selected real world concurrency bugs from four representative server and client open source applications. Hong *et al.* [41] explore the effectiveness of concurrency coverage metrics via a comprehensive empirical investigation. The results highlight the need for additional work on concurrency coverage metrics. Melo et al. [42]–[46] conduct a series of empirical research on concurrent software testing. For example, they propose a new classification for concurrent testing tools and construct a characterization schema for concurrent software testing techniques, which provide a useful selection guide for testing practitioners [42]. Recently, they conduct a systematic mapping study to identify and analyze empirical research on concurrent software testing techniques [46]. The results show that there is little empirical evidence available about some specific concurrent testing techniques. Our empirical study is different from the above ones, because it is focus on a specific concurrency bug detection technique, i.e., data race detection. The purpose of our study is to evaluate the

performance of data race detection based on thread shared analysis with different granularities.

V. CONCLUSION

This paper presents an approach to evaluate the performance of data race detection based on TSA with different granularities. In our experiments, we evaluate the performance of a dynamic data race detector RVPredict based on TSA approaches with object, field and "field + array element" granularities, respectively. The results show that RVPredict based on the TSA with "field + array element" granularity obtains the best performance without effectiveness reduction.

ACKNOWLEDGMENT

The authors are grateful to editors and anonymous reviewers for their valuable comments and useful suggestions. Special thanks to all the individuals who participated and contributed to improve the quality and readability of this paper.

REFERENCES

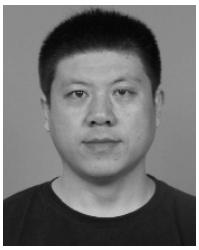
[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[2] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A deployable sampling strategy for data race detection," in *Proc. FSE*, Seattle, WA, USA, 2016, pp. 810–821.
[3] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *Proc. PADD*, Santa Cruz, CA, USA, 1991, pp. 85–96.
[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," in *Proc. SOSP*, Saint-Malo, France, 1997, pp. 27–37.
[5] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proc. PLDI*, Ottawa, ON, Canada, 2006, pp. 308–319.
[6] C. Radoi and D. Dig, "Effective techniques for static race detection in Java parallel loops," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 1–30, 2015.
[7] J. Huang and A. K. Rajagopalan, "Precise and maximal race detection from incomplete traces," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 462–476, 2016.
[8] Y. Cai and L. Cao, "Effective and precise dynamic detection of hidden races for Java programs," in *Proc. FSE*, Bergamo, Italy, 2015, pp. 450–461.
[9] J. D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 1–19, 2000.
[10] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 187–206, 1999.
[11] J. Huang, "Scalable thread sharing analysis," in *Proc. ICSE*, Austin, TX, USA, May 2016, pp. 1097–1108.
[12] J. Huang, P. O. N. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. PLDI*, Edinburgh, U.K., 2014, pp. 337–348.
[13] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proc. IPDPS*, Nice, France, Apr. 2003, pp. 22–26.
[14] L. A. Smith, J. M. Bull, and J. Obdrizálek, "A parallel Java grande benchmark suite," in *Proc. SC*, Denver, CO, USA, 2001, p. 8.
[15] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.
[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. V. Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 169–190, 2006.
[17] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Proc. ISSTA*, Toronto, ON, Canada, 2011, pp. 144–154.
[18] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 387–400, Jan. 2012.
[19] M. Naik, C. S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proc. ICSE*, Vancouver, BC, Canada, May 2009, pp. 386–396.
[20] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge, "Component-based lock allocation," in *Proc. PACT*, Brasov, Romania, Sep. 2007, pp. 353–364.
[21] H. Nishiyama, "Detecting data races using dynamic escape analysis based on read barrier," in *Proc. VM*, San Jose, CA, USA, 2004, pp. 127–138.
[22] M. Christiaens and K. D. Bosschere, "TRaDe: Data race detection for Java," in *Proc. ICCS*, Berlin, Germany, 2001, pp. 761–770.
[23] R. O'callahan and J.-D. Choi, "Hybrid dynamic data race detection," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 167–178, 2003.
[24] D. Li, W. Srisa-an, and M. B. Dwyer, "SOS: Saving time in dynamic race detection with stationary analysis," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 35–50, Oct. 2011.
[25] S. Biswas, M. Cao, M. Zhang, M. D. Bond, and B. P. Wood, "Lightweight data race detection for production runs," in *Proc. CC*, Austin, TX, USA, 2017, pp. 11–21.
[26] D. Chen, Y. Jiang, C. Xu, X. Ma, and J. Lu, "Testing multithreaded programs via thread speed control," in *Proc. ESEC/FSE*, Lake Buena Vista, FL, USA, 2018, pp. 15–25.
[27] K. Sen, "Race directed random testing of concurrent programs," in *Proc. PLDI*, Tucson, AZ, USA, 2008, pp. 11–21.
[28] D. Kini, U. Mathur, and M. Viswanathan, "Data race detection on compressed traces," in *Proc. ESEC/FSE*, Lake Buena Vista, FL, USA, 2018, pp. 26–37.
[29] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," *Sci. Comput. Program.*, vol. 71, no. 2, pp. 89–109, Apr. 2008.
[30] Q. Shi, J. Huang, Z. Chen, and B. Xu, "Verifying synchronization for atomicity violation fixing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 280–296, Mar. 2016.
[31] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "DoubleChecker: Efficient sound and precise atomicity checking," in *Proc. PLDI*, Edinburgh, U.K., 2014, pp. 28–39.
[32] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight deterministic multi-processor replay of concurrent java programs," in *Proc. FSE*, Santa Fe, NM, USA, 2010, pp. 207–216.
[33] J. Huang, C. Zhang, and J. Dolby, "CLAP: Recording local executions to reproduce concurrency failures," in *Proc. PLDI*, Seattle, WA, USA, 2013, pp. 141–152.
[34] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "CARE: Cache guided deterministic replay for concurrent java programs," in *Proc. ICSE*, Hyderabad, India, 2014, pp. 457–467.
[35] J. Wang, Y. Jiang, C. Xu, Q. Li, T. Gu, J. Ma, X. Ma, and J. Lu, "AATT+: Effectively manifesting concurrency bugs in Android apps," *Sci. Comput. Program.*, vol. 163, pp. 1–18, Oct. 2018.
[36] P. Kang, "Software analysis techniques for detecting data race," *IEICE Trans. Inf. Syst.*, vol. E100-D, no. 11, pp. 2674–2682, 2017.
[37] S. Hong and M. Kim, "A survey of race bug detection techniques for multithreaded programmes," *Softw. Test., Verification Rel.*, vol. 25, no. 3, pp. 191–217, May 2015.
[38] J. S. Alowibdi and L. Stenneth, "An empirical study of data race detector tools," in *Proc. CCDC*, Guiyang, China, May 2013, pp. 3951–3955.
[39] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware java runtime," in *Proc. PLDI*, San Diego, CA, USA, 2007, pp. 245–255.
[40] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. ASPLOS*, Seattle, WA, USA, 2008, pp. 329–339.
[41] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation," *Softw. Test., Verification Rel.*, vol. 25, no. 4, pp. 334–370, 2015.
[42] S. M. Melo, S. R. S. Souza, R. A. Silva, and P. S. L. Souza, "Concurrent software testing in practice: A catalog of tools," in *Proc. A-TEST*, Bergamo, Italy, 2015, pp. 31–40.
[43] S. M. Melo, P. S. L. Souza, and S. R. S. Souza, "Towards an empirical study design for concurrent software testing," in *Proc. SE-HPCCSE*, Nov. 2016, p. 49.
[44] S. M. Melo, S. D. R. S. de Souza, P. S. L. Souza, and J. C. Carver, "How to test your concurrent software: An approach for the selection of testing techniques," in *Proc. SEPS*, Vancouver, BC, Canada, 2017, pp. 42–43.

[45] S. M. Melo, S. do Rocio Senger de Souza, F. S. Sarmanho, and P. S. L. Souza, "Contributions for the structural testing of multithreaded programs: Coverage criteria, testing tool, and experimental evaluation," *Softw. Qual. J.*, vol. 26, no. 3, pp. 921–959, 2018.
[46] S. M. Melo, J. C. Carver, P. S. L. Souza, and S. R. S. Souza, "Empirical research on concurrent software testing: A systematic mapping study," *Inf. Softw. Technol.*, vol. 105, pp. 226–251, Jan. 2019.

LILI BO is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, China University of Mining and Technology. Her research interest includes software analysis and testing.

SHUJUAN JIANG received the Ph.D. degree from the Southeast University, in 2006. She is a Professor and Ph.D. Supervisor with the School of Computer Science and Technology, China University of Mining and Technology. Her research interests include compilation techniques and software engineering.

JUNYAN QIAN received the Ph.D. degree from Southeast University, in 2008. He is currently a Professor and a Ph.D. Supervisor with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology. His research interests include software engineering, model checking, and program verification.



RONGCUN WANG received the Ph.D. degree from Huazhong University of Science and Technology, in 2015. He is an Assistant Professor with the School of Computer Science and Technology, China University of Mining and Technology. His research interests include software testing, software maintenance, and machine learning.

YAFEI YAO is currently pursuing the M.Sc. degree with the School of Computer Science and Technology, China University of Mining and Technology. Her research interest includes software analysis and testing.