A Concurrent Harmful Races Identification Algorithm using Hadoop and Multiple Cloud Servers

Suxia Zhu\textsuperscript{a,b,*}, Yanan Wu\textsuperscript{a,b}, Guanglu Sun\textsuperscript{a,b}, and Jianda Sun\textsuperscript{a}

\textsuperscript{*School of Computer Science and Technology, Harbin University of Science and Technology, Harbin, 150080, China}
\textsuperscript{aResearch Center of Information Security and Intelligent Technology, Harbin University of Science and Technology, Harbin, 150080, China}

Abstract

Data race is widespread in multi-thread programs and can lead to serious failures. To improve the reliability of programs, many race detectors have been proposed. However, most of the detectors use binary instrumentation to detect potential races, imposing higher runtime overhead. Most of the potential races are false positive, which consumes manual effort to identify the harmful races. In order to reduce runtime overhead of identifying harmful races, we propose two concurrent strategies that reduce runtime overhead from the detection potential races stage and the verification harmful races stage. Unlike previous work, the detection and verification races are in one execution. In this paper, the Hadoop distributed system is used to detect the potential races concurrently from the trace, and then the weighted Round-Robin algorithm is used to divide all potential races to multiple cloud servers. Harmful races are verified concurrently in multiple cloud servers by controlling thread scheduling. The experimental results show that our method for identifying harmful races is more efficient. Compared with RaceFuzzer and ReceChecker, the runtime overhead is reduced by an average of 72% and 46% respectively. In addition, a good speedup is achieved in this paper.

Keywords: harmful race; multi-thread programs; Hadoop; thread scheduler; multiple cloud servers; weighted Round-Robin

(Submitted on July 9, 2018; Revised on August 8, 2018; Accepted on September 15, 2018)

© 2018 Totem Publisher, Inc. All rights reserved.

1. Introduction

In recent years, due to the rapid development of multi-core processors, multi-thread programming technology has been attracting more and more attention and has been widely used. However, because multi-thread programs are nondeterministic, that is, they depend on the results of interleaving among threads, bugs in concurrent programs are hard to detect \cite{1}. Data race is one of the most common types of concurrency bugs \cite{2}. Data race occurs when two or more threads access the same memory location, at least one of which is writing data, and the execution order is not enforced by the synchronization \cite{3}. While data race is widespread in concurrent programs, only 5-24\% \cite{4} of data races cause program failures (segment errors, memory accesses, errors in run results, exceptions thrown). Harmful races that are hard to be detected may result in serious damage after the software is deployed. For example, the 2003 Northeast blackout resulted in a power outage for nearly three days in the United States and northeastern Canada, resulting in more than 30 billion dollars of losses due to a small concurrency bug in the power management system \cite{5}. It can be seen that the existence of data race has seriously affected the reliability of the software, and there is an urgent need to detect harmful races before the software release.

Fortunately, previous researchers have put forward many tools to help programmers detect data race. We divide the detectors into three categories: static, dynamic, and hybrid. The static detectors \cite{6-7} examine the concurrent program without executing it, the dynamic detectors \cite{8} monitor memory accesses and synchronization operations by instrumentation during execution, and the hybrid detectors \cite{9} do both. Those detectors report data races and improve software reliability. Unfortunately, some detectors implemented as a binary instrumentation tool impose higher runtime overhead. Only 5-24\% of the reported races have an observably harmful effect, so manual effort is required to identify the harmful races that can lead to program failures. However, it can be extremely time consuming for a programmer to identify a harmful race from a...
number of potential races, and a harmful race must be fixed urgently before a software release. Therefore, efficient and automatic identification of the harmful races is desired. RaceFuzzer [10] executes the program with an active thread scheduler, checking whether a failure occurs automatically. RaceCheck [11] groups the potential races in one group. Multiple potential races in one group can be verified in one execution. Although those detectors identify the harmful races automatically [12], they still suffer from several limitations on efficiency. This greatly affects the software testing process.

In order to alleviate the limitations of previous detectors and improve the efficiency of identifying harmful races, this paper presents two concurrent strategies that use Hadoop distributed architecture to detect the potential races and verify the harmful races in multiple cloud servers respectively. Before verifying all the potential races, we divide all potential races to multiple cloud servers in a weighted Round-Robin algorithm, so that the total runtime of identification harmful races is minimal. To verify whether a potential race can result in program failure, we control the thread scheduler to enforce the order of memory access.

2. The Methodology of Identifying Harmful Data Races Concurrently

In this paper, an efficient algorithm for concurrent identification harmful races according to program interleaving invariant [3] is proposed. Figure 1 shows the architecture of our algorithm. It mainly includes two phases: (1) concurrent detection of the potential races and (2) concurrent verification of the potential races. The first phase is divided into two parts. First, the PIN tool is used to extract the original traces of the program’s dynamic execution. The HashMap is used to classify the original traces to multiple sets according to the invariant address [13]. Then, the classified trace sets are stored in Hadoop’s HFDS [14], taking full advantage of the power of the cluster for high-speed parallel operations of MapReduce to find all thread interleaving invariants (potential races) according to the happens-before relation. Then, in the second phase, we use the weighted Round-Robin algorithm to divide all the detected potential races to multiple cloud servers. The harmful data races that result in the failure of the program are identified concurrently by the thread scheduler on multiple cloud servers. In general, our work has made the following contributions:

(1) To our knowledge, this is the first paper that uses a concurrent method to detect potential races based on Hadoop distributed architecture. Since the original traces are classified to multiple sets according to invariant addresses, which prevents interference, Hadoop [15] can detect the potential races from each trace set concurrently on many parallel computing nodes. Our experimental results show that the efficiency of detection potential races is significantly higher than other detectors.

(2) To reduce the total runtime of verifying a large amount of potential races, we use a weighted Round-Robin algorithm to divide all potential races to multiple cloud servers. In order to make full use of every cloud server, we use a pre-execution to test the performance of each cloud server instead of blindly dividing potential races to multiple cloud servers in our paper. By the optimizing strategy, all cloud servers can verify the distributed potential races in almost the same time. Comparing with RaceFuzzer and RaceCheck, the runtime of verification in our method is reduced significantly, with an average of 72% and 46% respectively.

3. Multi-Thread Program Interleaving Invariant

Program invariant is used to describe the amount of program behavior that remains constant while the program is running [16], often in formalized descriptions and assertion statements. Some invariant is used among different threads with each instruction executing with read and write memory operations [17]. Data races often occur during access memory location among different threads. Therefore, this paper detects potential races based on multi-thread interleaving invariants, which are dependent on memory access during program running. In this paper, the interleaving invariant is recorded by an ordered pair of <W, R>. W is a write operation, R is a read operation, LcWr is a local write operation, RmWr is a remote write operation, and RmRd is a remote read operation. Depending on the conditions of data race occurring, two or more threads
access the same memory location without synchronization, and at least one is a write operation. Therefore, this paper selects the thread interleaving invariant of \( <\text{LcWr}, \text{RmWr}> \) or \( <\text{LcWr}, \text{RmRd}> \) in the same memory location to mark the interleaving order among threads.

Figure 2 shows the data race code fragment from MySQL. In this code, thread 1 and thread 2 may execute the function `ap_buffered_log_write` at the same time. In this function, the read statement (3 lines) and write statement (7 lines) of the variable `buf->outcnt` are not protected by the lock. Programmers can think that write 7 lines buf->outcnt value is 3 lines buf->outcnt value, forming \( <\text{LcRd}, \text{LcWr}> \), which is a true invariant, as shown by the solid arrows in Figure 2. However, during the dynamic execution of the program, it is possible that the execution sequence follows the dotted arrow in Figure 2. That is, before executing 7 line statement in thread 1, thread 2 has executed 3 and 7 lines of the statement, thus causing the value of buf->outcnt to be changed, and the buf->outcnt value is not what the programmer expects. In this code, (Thread1: 3, Thread2: 7) and (Thread2: 7, Thread1: 7) forming \( <\text{LcWr}, \text{RmRd}>, <\text{LcWr}, \text{RmWr}> \) are false invariants, which are defined as the potential races because they access the same shared variables without synchronization and at least one of them is a write access.

![Figure 2. A real world example of data race from MySQL.](image)

4. Detecting Potential Races Concurrently based on Hadoop

This section shows the use of Hadoop [18] to detect potential races in the dynamic execution of multi-thread programs. The PIN tool is first used to dynamically instrument the program to extract the original traces. We remove the repeated read and write traces during the extraction process. The HashMap based on the unordered mapping is used to classify original traces into multiple sets according to the shared variable address [13]. The classified traces are stored into the Hadoop [18] Distributed File System (HDFS), and the MapReduce framework [14] decomposes the detection algorithms into many parallel computing instructions. A large number of computing nodes detect all the false invariants from the above trace sets concurrently. Figure 3 is the system architecture to detect potential races in multi-thread programs.

![Figure 3. Concurrent detection potential races system architecture](image)

4.1. Extracting the Original Traces of Multi-Thread Program

We use our PIN tool to instrument the multi-thread program that extracts the original traces of the process running. PIN is a program instrumentation tool provided by Intel Corporation [19] that allows the programmer to insert an arbitrary code anywhere in an executable program. In order to detect races, the extracted program traces should contain useful information for detection race. Among them, the thread number (TID) is indispensable, that is, it is conducive to mark the interaction order among the threads. The instruction counter (PC) can identify instructions to determine the running status of the
program. The timestamp (T) can record the execution time of the instruction. The instruction operation (R/W) is used to identify the instruction to read or write memory. The ADDR is used to identify the operation that the instruction made on a particular variable.

The programs contain many repeated operations such as repeated read and write during execution. These repeated read and write operations are redundant data that have no benefit in extracting multi-thread interleaving invariants, because all the records have the same effect as recording one trace. Therefore, the algorithm only records the first read trace when it encounters repeated read operations in the same thread. Similarly, only the trace information on the last write instruction is recorded when encountering repeated write operations in the same thread. We also classify the original traces to multiple sets based on HashMap, and it does not reduce the search efficiency as the amount of data increases. Each classified trace set represents all the operations on a certain variable, which is beneficial to reduce the search space when extracting multi-thread interleaving invariants. In this paper, keys are directly mapped to memory addresses for fast addressing, that is, Addr = HASH (key). In addition, HashMap based on unordered mapping is used in the trace classification process [20], because we need to search and insert nodes frequently. The unordered mapping is a vector, and the hash [20] function determines the insertion position. When a conflict occurs, a separate link method is used to solve the conflict by hanging the linked list under the corresponding vector element node as shown in Figure 4. The HashMap also saves the time of red-and-black tree adjustment compared to normal mappings. At the same time, an index of each trace set will be generated to improve the efficiency of detecting potential races, which contains the memory address and the number of set elements. If there are fewer than two elements in a trace set, we will not process the set.

![Figure 4. HashMap based on unordered mapping](image)

### 4.2. Detecting Potential Races in Traces based on Hadoop

Detection data race based on multi-thread interleaving invariant has some flaws, such as large amounts of traces and slow processing speed. In order to reduce the detection time, we propose a parallel strategy Hadoop [14] distributed system to detect the potential races, making full use of the power of the cluster for high-speed parallel computing. Hadoop is an open-source distributed cluster platform that includes a distributed file system, HDFS, and the programming model, MapReduce. MapReduce is a flexible paradigm that enables the development of large-scale distributed programs. It consists of two distinct functions, namely Map and Reduce, which are combined together in a divide-and-conquer way. The Map function is responsible for handling the parallelization, while the Reduce function collects and merges the results. In particular, a master node splits the initial input into several pieces, with each one being identified by a unique key, and distributes them via the Map function to several slave nodes. Those nodes work in parallel and independently from each other, performing the same task on a different piece of input. As soon as each Mapper finishes its own job, the output is identified and collected via the Reducer function. In particular, each Mapper produces a set of intermediate key value pairs that are exploited by one or more Reducer to group together all the intermediate values.

In this paper, Figure 5 shows the proposed architecture for detection potential races based on Hadoop. It is composed of the following main components: a Master, a number of Mappers, a Reducer, and a detection races Algorithm, together with two other units, InputFormat and OutputFormat, which are responsible for splitting the data for the Mappers and storing the Reducer output into HDFS respectively. The MapReduceJob is the core of the detection potential races, since it allows us to parallelize extraction thread interleaving invariant and computation over the nodes. In particular, a MapReduceJob consists of three phases (i.e., **Split, Map, and Reduce**) in which each component performs its proper task as detailed in the
following.

**Split.** In this phase, the InputFormat module gets the trace sets from the HDFS and processes them in order to split them for distribution among the Mapper modules. The number of input splits is dynamically computed on the basis of the number of available Mappers. In more details, the InputFormat begins to emit the \(<\text{address}, \text{trace set}>\) pairs.

**Map.** In this phase, each Mapper carries out its task on the received input split in a parallel and independent way. In particular, each Mapper is responsible for detecting all multi-thread interleaving invariants (potential races) of each input split according to the happens-before relation. Since the most likely data race is often a neighboring operation, it is necessary to consider the context of the current instruction instead of the full-text relationship. Therefore, the advantages of the stack are very apparent when detecting potential races. The stack is characterized by last-in, first-out. It is suitable for detecting potential races. Algorithm 1 shows the processes of Mapper. Once such a process is completed, each mapper generates a new pair \(<\text{key}, \text{value}>, \text{value} = \text{a list of multi-thread interleaving invariants (i.e., } <\text{LcWr, RmWr}> \text{ or } <\text{LcWr, RmRd}>\)\>. The key is the address of each shared variable. Then, the Master module uses it to assign the Reducers and collect the outputs produced by the Mappers that will constitute the input for Reducer. The outputs are saved into the HDFS by the OutputFormat in order to be used in the verification harmful race phase.

**Reduce.** Once the Master component has collected all the Mapper outputs, a single Reducer is responsible for computing the total number of races. It is a simple add operation for the length of each input value.

---

**Algorithm 1 Function RaceDetection(traceSet)**

**Input:** traceSet

**Output:** nRaceNum, lstRace

1. while(1)
2. szContent=traceSet.readline();
3. if szContent == "#eof":
4.   stack.clear();
5.   stackInvar.clear();
6.   break;
7. if szContent.op == 'W':
8.   if stack.isEmpty():
9.     stack.push(szContent);
10.   else:
11.     szWrite=stack.top();
12.     if szWrite.tid != szContent.tid:
13.       stack.push(szContent);
14.     else if szContent.op == 'R':
15.       if stack.isEmpty():
16.         szWrite1=stack.pop();
17.         Inv = mKInvariant(szWrite1, szContent);
18.         nResult=WWRCheck(inv, stackInvar);
19.         nRaceNum = nRaceNum + nResult;
20.       if (nResult = 1):
21.         lstRace.push(errorMsg);
22.         stackInvar.push(inv);
23.         stack.clear();
24.       return nRaceNum, lstRace
The algorithm is described as follows: each time, the mapper task reads a trace from the split trace set. If the trace is the "#eof", the mapper task closes the current trace set and clears the stack. If the trace is a write instruction and the stack is empty, the write instruction pushed. Otherwise, if the trace is in the same thread and write instruction as the top-of-stack element, the top-of-stack element is discarded and the current trace is pushed, because the writing instruction for the same thread does not affect the number of races. If the current trace and the top of the stack elements are in different threads, the current trace is pushed directly into the stack. Conversely, if the current trace is a read instruction, it will cause the trace in the stack to pop out. When the stack is empty, there is no need to detect the trace. Otherwise, it is judged whether the current trace and the top element of the stack are located in the same thread. If the thread is in the same thread, the interleaving invariant <LcWr, LcRd> is extracted. This invariant is very safe and does not need to worry about data race. On the other hand, if it is located on a different thread, the top element of the stack is popped and the thread interleaving invariant <RmWr, LcRd> is extracted. This invariant is very dangerous and may cause race. Then, empty the stack to avoid the remaining trace of the stack and the new trace to generate a new dependency.

5. Identifying Harmful Races Concurrently based on Cloud Servers

The potential races detected in the previous phase through Hadoop include a large amount of infeasible races that are false positives. Only 5-24% of potential races are harmful races that can lead to program failures. In order to identify harmful races efficiently, we divide potential races into N cloud servers by the weighted Round-Robin algorithm and verify these races concurrently by controlling the thread scheduling, which can identify the harmful races.

5.1. Dividing Races

In order to reduce the total verification time, we use pre-execution to help divide the potential race [12]. For example, if we need to allocate the multi-thread program FFT to two servers, we first perform FFT on both servers and record the execution time of the FFT on each server. If the FFT execution time on Server A is twice that of Server B, we send one race in FFT to A and two races to B. Pre-execution can help us divide the races on N servers. If we need to divide the potential races of another program, we first need to do some pre-execution of the program. After recording the execution time of the program on each server, the potential races are divided according to the execution time of the program on each server.

Figure 6 shows our strategy for dividing potential races to multiple cloud servers, which is a weighted Round-Robin algorithm. For example, the execution time of the multi-thread program on each server is 1s, 2s, 3s, and 4s respectively, and there are four cloud servers (A, B, C, and D). The numbers of the circles indicate the order in which the race is sent. Our principle is to verify the divided potential races on each server in a similar total time. In Figure 6, race 1 is sent to server A. To determine which server race 2 should be sent to, we calculate each server’s time. If race 2 is sent to server A, the time spent on A is two seconds (A needs to validate two races). If race 2 is sent to server B, the time spent on B is two seconds. The situation on machine C is similar to B. If race 2 is sent to server D, the time spent on D is four seconds. Therefore, we cannot send out race 2 to server D. Since machines B, C, and D spend more time than A, we send out race 2 to the A server. Repeat this process until all potential races are sent to the server. If we need to verify the race between different programs, the order of the cloud servers may be different due to the execution time of multi-thread programs.

![Figure 6. Division potential races based on weighted Round-Robin algorithm to multiple cloud servers](image)

5.2. Controlling Thread Scheduling to Identify Harmful Race

In order to identify harmful races, we need to verify each potential race, which we have detected based on Hadoop. We try...
to create real race conditions to check whether the potential race will lead to the program failure. In most cases, the harmful races hidden in the concurrent program will not be triggered. For example, the hidden race in Atom-Aid’s kernel program (BANKACCOUNT) is obvious. However, in most cases, this race will not be exposed during native execution. We have run the original program 1000 times without triggering race. The main reason is that the instructions “Read Account -> Balance” and “Write Account -> Balance” are executed continuously in one thread so that the same instructions in other threads can hardly be executed at the same time. Although these two conflict instructions are not protected by the synchronization, the order of execution during native execution does not result in access to the same memory address. These races are easily overlooked during internal testing and can cause serious problems after deployment.

When executing the program on each cloud server, this paper dynamically creates the conditions for data race occurring by the instrumentation relevant instructions of the potential races. There are two different execution sequences for two memory accesses in an interleaving invariant (s1, s2): s1 executes before s2 or s2 executes before s1. Figure 7 (b) and (c) shows a data race in two different executive orders. Dynamic implementation can control the order of thread execution. Note that there are no instructions that access the same memory address of this race execution between s1 and s2. If a race is harmful, one of the two execution sequences will lead to a program failure. Therefore, to identify whether a race is harmful, we only need to check two different execution sequences of s1 and s2.

Figure 7 (a) describes how to dynamically create the condition of data race (s1, s2) occurring. This paper inserts delay control instructions before the data-related instructions in the binary code of the program. When the program is running dynamically, the PIN tool monitors the execution of the instructions of two threads. When thread 1 is about to execute instruction s1, the inserted control instruction pauses thread 1 and waits for thread 2. The instruction s2 is executed by thread 2. When thread 2 is about to execute instruction s2, the binary instrumentation code will control the order of thread 1 and thread 2. Let s2 execute before s1 (Figure 7 (c)), or let s1 execute before s2 (Figure 7 (b)). If all potential races are verified on all cloud servers, the process of harmful races identification is finished.

6. Evaluation

We have implemented our method that uses Hadoop distributed technology to detect potential races in multi-thread programs based on program interleaving invariants and concurrently identifies harmful data races on multiple cloud servers. In this section, the experimental evaluation aims at two aspects: (1) detection efficiency and (2) verification efficiency. The benchmarks are a number of C/C++ programs that are a kernel program (BANKACCOUNT, named BA) in Atom-aid and five scientific programs (FFT, LU, RADIX, OCEAN, CHOLESKY) from SPLASH2 [21]. All experiments are run on an Ubuntu system, with 4-core 8-thread CPU and 8GB of physical memory.

To compare our method with previous detectors, we selected the most related detectors: RaceFuzzer [10] and RaceChecker [11], which are also designed to identify harmful races. For RaceFuzzer, we use RELAY [22] to detect potential races, with dynamically verifies the potential races one by one. For RaceCheck, we also use RELAY to detect potential races and prune infeasible potential races by combining the happens-before relation and ad-hoc synchronization. In the verification phase, RaceCheck groups the remaining potential races, guaranteeing that potential races in one group do not interfere with each other, so multiple potential races in one group can be verified together in one execution. To evaluate the efficiency of our method, we describe the efficiency of detection and verification phase in detail respectively.

(1) Detection efficiency. In order to reduce runtime overhead of detection potential races, this paper makes partial optimization to remove redundant data when extracting the original traces. The comparison about the number of the original traces in six benchmarks is shown in Figure 8. The total amount of the optimized program trace is about 32% less than that of the unoptimized. The amount of traces actually used for detection is reduced by 10% on average, compared with the total number of traces of the optimized program. That is, the total amount of traces actually used for detection is reduced by 42%. This not only reduces the disk and memory space, but also greatly reduces the search space when extracting interleaving invariants. In particular, the total amount of traces will be reduced significantly as the complexity of the multi-thread
program increases.

![Figure 8. The total amount of traces](image)

Table 1 shows the number of potential races and the runtime of detecting potential races in three detectors. Column 2 shows that the number of potential races is maximized for RaceFuzzer, which uses RELAY with many false positives. Column 3 shows that RaceCheck has the least potential races. Although RaceCheck also uses RELAY to detect potential races, it can prune infeasible races through the happens-before relation and ad-hoc synchronization. Column 4 shows the potential races of our method, which is more than RaceCheck, because we only use the happens-before relation and ignore ad-hoc synchronization. In addition, we extract original traces through five executions in order to reduce false negatives. Columns 5, 6, and 7 show the runtime overhead of detecting potential races by RaceFuzzer, RaceCheck, and our method respectively. The original runtime of our method is longer than those of RaceFuzzer and RaceFuzzer, because we extract traces by dynamic binary instrumentation instead of using a static detector. However, when we use Hadoop, the runtime is reduced significantly. In particular, the runtime is less than that of RaceFuzzer when the node of Hadoop is 5 or 6. The main reason is that we classify the original traces to multiple sets according to memory access, from which Mappers detect potential races in a parallel, independent way.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Potential races</th>
<th>Detection time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RF</td>
<td>RC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>FFT</td>
<td>713</td>
<td>77</td>
</tr>
<tr>
<td>LU</td>
<td>1677</td>
<td>311</td>
</tr>
<tr>
<td>RADIX</td>
<td>958</td>
<td>368</td>
</tr>
<tr>
<td>OCEAN</td>
<td>1677</td>
<td>425</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>8135</td>
<td>30</td>
</tr>
</tbody>
</table>

(2) Verification efficiency. In order to identify harmful races, this paper creates real race conditions by controlling thread scheduling. Table 2 shows the trigger probabilities of harmful races in six benchmarks, which run 100 times in natural conditions. Column 2 shows the probability that harmful races are triggered in the natural execution of multi-thread programs. It can be seen that it is very difficult to trigger harmful races if we do not use any strategies. Column 3 shows the probability that triggering harmful races causes the program failure is increased by an average of 81.3% by instrumenting codes related potential race to control thread execution order. Therefore, it is useful to control thread scheduling to identify harmful races.

<table>
<thead>
<tr>
<th>Program</th>
<th>Natural conditions (%)</th>
<th>Thread control trigger (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>0</td>
<td>74</td>
</tr>
<tr>
<td>FFT</td>
<td>0</td>
<td>95</td>
</tr>
<tr>
<td>LU</td>
<td>0</td>
<td>87</td>
</tr>
<tr>
<td>RADIX</td>
<td>4</td>
<td>89</td>
</tr>
<tr>
<td>OCEAN</td>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>3</td>
<td>66</td>
</tr>
<tr>
<td>Average</td>
<td>1.3</td>
<td>83.6</td>
</tr>
</tbody>
</table>
Because dynamic instrumentation extracts original traces and controlling thread scheduler triggers harmful races, they are all added some additional runtime overhead. To ensure that all harmful races are identified with a limit time, the paper proposes another concurrent strategy that verifies all potential races divided by the weighted Round-Robin algorithm on multiple cloud servers. We have also done some experiments to evaluate the efficiency of the verification phase. Table 3 shows the results of comparison with RaceFuzzer and RaceCheck. Columns 2, 3, and 4 show the number of the harmful races identified by three detectors respectively. In particular, the number of harmful races in RADIX is 23, which is the largest. We analyze the reasons, finding that the synchronous barrier was used incorrectly in RADIX. Columns 5, 6, and 7 show the verification time in three detectors. By the strategies of dividing the potential races according to the weighted Round-Robin algorithm and concurrently verifying on the multiple cloud server, the efficiency of our method is much better. Comparing with RaceFuzzer and ReceChecker, the runtime overhead is on average reduced by 72% and 46% respectively. For describing the scalability of our method, Figure 9 shows average runtime of verifying potential races in our method on multiple cloud servers. As we can see, the runtime is maximized when the number of servers is 1, because there will not be the divide potential races phase or concurrent verification phase. Once the number of servers is greater than 2, the runtime will be reduced greatly. Therefore, our method could scale effectively as the number of cloud servers increases.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Harmful races</th>
<th>Verification time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RF</td>
<td>RC</td>
</tr>
<tr>
<td>BA</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FFT</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LU</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RADIX</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>OCEAN</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 9. Time comparison of concurrent verification harmful races based on multiple cloud servers

7. Conclusion

This paper proposes an innovative, efficient algorithm to identify harmful races. The algorithm uses a PIN tool to extract the original traces of the programs, removing the redundant data. The original traces are classified into multiple trace sets based on the HashMap according to memory access, greatly reducing the search space for the detection phase. Then, Hadoop distributed architecture concurrently processes the trace sets to detect potential races on several computation nodes. In the verification phase, the weighted Round-Robin algorithm is used to divide potential races to multiple cloud servers, verifying the potential races concurrently by controlling thread scheduling on multiple cloud servers. The evaluation results show that our concurrent strategies significantly improve the efficiency of identifying harmful races.

Acknowledgement
This work was partly financially supported through grants from the National Natural Science Foundation of China (No. 61502123), Heilongjiang Province Science Foundation for Youths (No. QC2015084), and China Postdoctoral Science Foundation (No. 2015M571429). The authors appreciate the anonymous reviewers for their valuable work.

References


Suxia Zhu is currently an associate professor. Her research interests include parallel computing and high-performance architecture.
Yanan Wu is currently a Master’s student in the School of Computer Science and Technology at Harbin University of Science and Technology. Her research interests include parallel computing and concurrency bug detection.

Guanglu Sun graduated from the School of Computer Science and Technology at Harbin Institute of Technology with a bachelor’s degree, Master’s degree, and Ph.D. He entered the post-doctoral mobile station of the Computer Science Department at Tsinghua University as an assistant researcher from 2008 to 2011. He was a visiting scholar at Northwestern University in the United States from 2014 to 2015. He is currently a professor in the School of Computer Science and Technology and the director of the Center of Information Security and Intelligent Technology at Harbin University of Science and Technology. He is also a senior member of the China Computer Federation and a member of IEEE. His current research interests include computer networks and security, machine learning, and intelligent information processing.

Jianda Sun is currently a Master’s candidate at Harbin University of Science and Technology. His research interests include parallel computing and concurrency bug detection.