Detecting First Races using Trace and Replay for Parallel Programs with Random Synchronization

Hee-Dong Park¹ and Yong-Kee Jun²

¹Joongbu University, Kumsan, Korea
hpark@joongbu.ac.kr
²Gyeongsang National University, Jinju, Korea
jun@gnu.ac.kr

Abstract

Debugging multithreaded or parallel programs is a hard problem due to unintended non-deterministic results of the program, called data races. Previous race detection techniques have limitations of coverage for synchronization constructs in detecting first races for parallel programs with random synchronizations. We present, in this paper, an efficient two-pass algorithm to detect the first races in a particular execution of parallel program, by eliminating independent accesses to make a performance gain and candidate accesses, and then analyzing the traced information with candidate accesses by replaying the program. Therefore, our technique makes the race detection more practical in debugging parallel programs which have random synchronizations.

Keywords: data race, race detection, first race, concurrent programming, parallel program debugging, random synchronization, multi-threaded programming, parallel environments, trace and replay

1. Introduction

With increasing of multi-core and parallel computers, multithreaded programs may introduce concurrency defects which are difficult to detect due to the non-deterministic program behavior caused by various thread inter-leavings. A data race occurs in a parallel or multi-threaded program when a number of threads can be arbitrarily interleaved and access the same memory location without proper synchronization constraints between accesses, such that at least one of the accesses is a write [1]. Incorrect synchronization leads to incorrect ordering between accesses to shared memory. Programs which have data races usually do not give system failure immediately, but resulting in inconsistent data and unpredictable behavior of execution, therefore the importance of techniques that find data races is significantly increasing.

For race detection in parallel program, it can be classified as static and dynamic analysis. Static analysis [6] can be precise but suffer from many unfeasible races because it analyzes source code for race conditions without actually executing the software, so is prone to false negatives and false positives. Dynamic technique instruments the program and monitors an execution of the program, and includes post-mortem trace-based and on-the-fly method.

On-the-fly techniques [7, 3, 18, 23] analyze the trace information as it is generated, thus the entire trace does not need to be stored, and this technique naturally focuses on those races involving the shared memory accesses reported during the execution. Post-mortem approach [4, 11] records events during program executions and analyzes them
later. This technique, that is, is based on traces where an attempt is made to determine orderings between all blocks without regard to exactly which shared memory locations were accessed. Likewise, any post-mortem approach could be done on-the-fly with a sufficiently large buffer. The monitoring process in dynamic technique reports races which occur during the monitored execution for each variable involved, but may still produce false alarms or miss races.

A lot of technique has been proposed for detecting data races dynamically in lockset, happens-before and hybrid analysis. Lockset analysis checks whether two threads accessing a shared memory location hold a common lock and can thus report the potential data races in a program efficiently. The technique is simple, and can be implemented with low overhead with relatively insensitive to execution order. The main drawback of a pure lockset-based detector is that it produces many false positives due to the fact that it ignores synchronization primitives other than locks, such as signal/wait, fork/join, and barriers [9]. Happens-before approach uses vector clock approaches to record logical times of memory accesses, this approach is precise by reporting all and only the real data races found during a particular execution of a program, but may miss data races despite repeated execution due to its sensitivity to thread inter leavings. However, it has high overhead in monitoring all memory accesses. A number of approaches have worked to reduce these weak points, such that combining happens-before relation with lockset algorithm [9, 15]. Due to the difficulty of identifying data races accurately, some multi-threaded deterministic record/replay schemes which record data races directly could bring remarkable slowdown to the production step and introduce significant efforts to replay.

The races that occur first (first races) are races between two accesses that are not causally preceded by any other accesses also involved in races, or those that were unaffected by any other race. Locating these races is complicated because in practice only partial information about how races affect one another is available. Exact information may be costly to record and may depend on the semantics of the program. Locating first races with partial information involves estimating which races may have affected later races. This estimation becomes hard when a pair of races is tangled, where each race has an event that precedes an event in the other, making it unclear which race affected the other. The first races are important in debugging because the removal of such races may make other races disappear. It is even possible that all races reported would disappear once the first races are removed.

In multi-threaded and parallel programs, threads can be coordinated via a barrier operation, signal/wait or lock/unlock primitives and others. It is generally recognized that accurate models of parallel execution need to describe the complex interactions among concurrent activities. In case of synchronized execution, it is required that all computations running concurrently complete, before the next set of activities can proceed. The ordered synchronization directives are executed in the order in which iterations would be executed in a sequential execution of the loop, while allowing program outside the section to run in parallel. Random synchronizations in a parallel program cannot guarantee the order of synchronization events to shared memory, which could result in different races, if any, when executed repeatedly with the same input. Our primary interest is to detect first races for parallel programs which have non-deterministic sequence of synchronization events. Because random synchronization behavior encompasses ordered one, it could be applicable to the programs including mutual exclusion.
We propose a two-pass framework to find first races in shared-memory programs with random synchronization. In the first pass, our algorithm collects filtered candidate accesses in which at least one access is included in first races, and saves traced information which have total order in a particular execution of the program. In the second pass, our technique analyzes the candidate accesses by replaying the program using traced information to extract the first races.

This paper is organized as follows; Section 2 describes the necessary background on random synchronization constructs and on first race with definitions related to our technique. In section 3, we present our two-pass algorithm to collect candidate accesses that are involved in races, and to extract first races from traced information using logical program replay, and finally describes our conclusions.

2. Background

Almost all thread-based programs are inherently non-deterministic because of the lack of program code on the relative execution speed of the different threads. Cyclic debugging cannot be used for these non-deterministic programs as one cannot guarantee that the same execution will be observed during repeated executions. Dynamic data race detector finds data races that occur during a particular execution. The flow between threads for execution of shared-memory parallel program can be represented by a directed acyclic graph called POEG (Partial Order Execution Graph) [3] which captures the happen-before relation [1].

An access $a_i$ happens before another access $a_j$, denoted as $a_i \rightarrow a_j$, and it is concurrent with each other if there exist no paths between them, denoted as $a_i \parallel a_j$. An access $a_j$ is affected by another access $a_i$, if $a_i \rightarrow a_j$ and $a_j$ is involved in a race. A race $a_i$ is unaffected, if neither $a_i$ nor $a_j$ is affected by any other accesses. The race is partially affected, if only one of $a_i$ or $a_j$ is affected by another access. A tangle $T$ is a set of partially affected races such that if $a_i$ or $a_j$ is a race in $T$ then exactly one of $a_i$ or $a_j$ is affected by $a_k$ such that $a_k \rightarrow a_i$ or $a_k \rightarrow a_j$. There can be many races in an execution of parallel program, and first race to occur or simply a first race is either an unaffected race or a tangled race.

Previous race detection techniques [8, 12, 10] cannot locate the candidate accesses which contain at least one access involved in the first races for parallel programs with random synchronization. Two-pass on-the-fly algorithm [10] to detect the first races, which is focused on parallel program with ordered synchronization, extracts candidate

---

**Figure 1. Parallel Programs with Random Synchronization**

(a) Using Post/Wait

(b) Using Lock/Unlock
accesses from many access sequences comparing to the Access History (AH) maintained as global variables, and examines the logical concurrency between current access and candidate accesses in order to make final candidate races. Another on-the-fly technique [13], which maintains a small constant of access history for each shared variable, is efficient with regard to the memory space. It cannot applicable to programs with inter-thread coordination, however, and it fails to report tangle race in some cases.

Parallel programs with random synchronization could exhibit different sequence of events when executed repeatedly, that is, cannot guarantee the access sequence in one execution to that in another execution due to small timing variations in execution of synchronization event, and could result in non-deterministic event ordering for debugging. There can be a number of synchronization constructs between threads or accesses such as fork/join, post/wait, lock/unlock and others. These constructs may be observed either implicit or explicit synchronization, and could be used as deterministic or random order executions between synchronization events. It has been proved that detecting races in program executions that have synchronization powerful enough to support mutual exclusion is NP-hard [2]. Determining event ordering relationship for post/wait without clear is Co-NP hard on no loop and monotonic programs, and is NP-hard for post/wait with clear on non-monotonic programs [7]. Thus detecting actual races is of practical use in a particular execution of such parallel programs [5].

For example, Figure 1 shows two POEGs of parallel programs which contain random synchronization events or mutual exclusion. Although the threads in Figure 1(a) execute in parallel, the synchronization operations (such as S1, S2, S1* and S2*) where each one is composed of a pair of post event P(a) and wait event W(a), are executed in a sequenced pair of S1-S2 or S1*-S2* according to the execution speed of each thread. Between threads for lock/unlock mutual exclusion operations, as shown in Figure 1(b), unlocking by one thread in multithreaded programs should happen before another thread which can grab the lock, as the message sending by one process should causally happen before another process receives the message. Therefore, the order of accesses could not be the same in one execution to that in another execution when executed repeatedly.

To present our two-pass algorithm, in this section, for detecting first races using trace and replay in parallel programs with random synchronization, we define a set of terms. Then, we introduce a new algorithm which collects subset of candidates and trace information in the first pass, and then completes with analyzing set of candidates and traced accesses in the second pass.
2.1. Candidate Set and Traced Accesses

In a parallel program, a synchronization block or a block is an access sequence between inter-thread coordination events (such as post/wait, fork/join), and does not include thread or coordination operations.

We define a write (read) access as a key access if there exist no other write (read or write) access within a block such that. Key accesses are not always involved in the race, and maintaining all the key accesses to detect first races is also inefficient. All other accesses other than key accesses will not be involved in the first races because any races involving these accesses will be affected by the races containing key accesses.

Effective Keys for a shared variable $X$ and thread identifier $T$, denoted by $Ek(X,T)$ is a set of effective key accesses in a block of thread $T$. Every $Ek(X,T)$ is maintained as local variable of corresponding thread $T$, which can be free from mutual exclusion with shared memory of other threads or processors. An $Ek(X,T)$ has maximum two accesses(read and/or write) and is cleared at each synchronization event.

**Definition 2.1** Traced Accesses(TA) is a sequence of key accesses saved in a trace file with total order in a particular execution of parallel programs.

In an execution of parallel program, the first key accesses within a block (that is, trace information) will be stored in a consecutive order to a trace file. Therefore each access in trace file could be used to replay the parallel programs with total order afterward. From Figure 2, for example, we can see that effective keys can hold at most two accesses between synchronization events for each thread.

**Definition 2.2** A read(write) access $a_i$ in the corresponding access history(AH) is a read(write) candidate, if $a_i$ is involved in a race and there exists no other access $a_h$ such that $a_h \rightarrow a_i$ and $a_h$ is involved in a race.
Definition 2.3 Candidate Set for a shared variable X, denoted by CS(X), is a set of candidates which are involved in the race for a shared variable X: CS(X,R) for a set of read candidates, CS(X,W) for a set of write candidates and CS(X,RW) for a set of read-write candidates.

All these candidates, however, are not always involved in the first races. Accesses in candidate set can be removed or added, and read-write candidates will be collected or moved from CS(X,W) to CS(X,RW) in the second pass.

3. First Race Detection using Trace and Replay

Figure 3(a) shows a particular execution of Figure 2, where an execution in which the first wait event, W(a) of thread $T_{21}$, first matches to the post event, P(a) of thread $T_1$. This execution causes another P(a) of thread $T_{22}$ will trigger the second W(a) of thread $T_{21}$ afterward, represented as solid arrow lines(S1 and S2). In this particular execution, we can get access histories, candidate sets and traced access as shown in Table 2. In this particular execution, we can get access histories, candidate sets and traced access as shown in Table 2.

3.1. Algorithm

Our detecting algorithm collects a subset of candidate accesses and saves sequence of traced accesses into a trace file in the first pass, and then completes the set of candidates by replaying using the accesses from trace file in the second pass. The algorithm is as shown in Table 1.

In the first pass, we monitor all memory operations executed during a particular execution and filter out the accesses to get key accesses. For each access, if it is a key then saves or appends it into a trace file. All the accesses which are not key will be excluded in the processing of the first race determination, which makes our algorithm more efficient in time and space usage. From the Figure 3(a), all the accesses $(r_3, r_5, r_7, r_{12}, r_{13})$ which are not key accesses will be returned, so only key accesses will be considered in our first pass. We update the corresponding access history which contains mutually concurrent accesses with the key access. This means that every key
access is compared the logical concurrency with the accesses in AH(X) in an execution of program to get a subset of candidate accesses, CS(X). In this step, the key access will be saved or returned according to the concurrency or happens-before relation.

**Table 1. Algorithm for First Race Detection**

<table>
<thead>
<tr>
<th>Detection Algorithm</th>
<th>Detection Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process_1st_Pass</strong>(X,current)</td>
<td><strong>Process_2nd_Pass</strong>(X)</td>
</tr>
<tr>
<td>1 if ¬IsKey(X,current) then</td>
<td>1 while ¬EOF(trace_file)</td>
</tr>
<tr>
<td>2 return</td>
<td>2 current := trace_file(X)</td>
</tr>
<tr>
<td>3 else save current to trace_file</td>
<td>3 Check_2nd_pass(X,current)</td>
</tr>
<tr>
<td>4 Check_1st_pass(X,current)</td>
<td>4 End_while</td>
</tr>
</tbody>
</table>

**Check_1st_pass**(X,current)

1 for all a in AH(X)
2 {
3 if (a → current) then
4 if race_bit(a) then return
5 else delete a
6 endif
7 }
8 add current to AH(X)
9 for all a in AH(X)
10 {
11 if (a ↔ current) then
12 set race_bit(current and a)
13 endif
14 }
15 if ¬race_bit(current) then return
16 add current to CS(X)
17 if (current is write) then halt
EndCheck_1st_pass

**Check_2nd_pass**(X,current)

1 for all c in CS(X)
2 {
3 if (a ↔ current) then
4 race := true;
5 }
6 if ¬race then return
7 for all a in CS(X,R)
8 {
9 if (a → current) then return
10 if (current → a) then delete a
11 }
12 if (current is rw) then
13 add current to CS(X,RW)
14 else
15 add current to CS(X)
16 endif
17 if (current is write) then skip
EndCheck_2nd_pass

The algorithm is similar with the race verification protocol using access histories, but it uses a special bit, called race bit, for each access stored in access history. The race bit is set if its access get involved in a race. Race bits are shown in Table 2 which is coded with asterisks attached to the accesses. The algorithm halts the current thread if the current access is a write candidate. It is of no use to proceed more than the current point in this case because a write candidate does not happen before any other candidate. This step can reduce additional detection time and makes to report less races than expected. After first pass, we can see that access histories, AH(X) contain only mutually concurrent accesses, and there exists at least one access which is included in the first races. [20]
In the second pass, we only analyze the traced information with the candidate accesses instead of re-executing the parallel program. Our algorithm to detect first races is based on the observation that it is not necessary to replay all the accesses, but it suffices if we just enforced those dependencies during replay. We do not need to re-execute the synchronization events during replay: it suffices if we just enforced the appropriate dependencies during replay using traced accesses. We only focused on the logical view of execution because it is enough for our algorithm to detect the first races, as shown in Figure 3(b).

We update all candidate sets while creating read-write candidate set, if needed, using two sets of candidates, read candidate set and write candidate set which were collected in the first pass, and also using ordered accesses from trace file. We replay and check the concurrency of current access from trace file with the accesses in candidate set, and determine whether the access is affected or become a read-write candidate, and then update the candidate sets which have the first races to occur. For example, from Table 2 and Figure 3(a), we can get the candidate accesses \( r_4^*, r_8^*, r_11 \) and traces accesses as \( r_1, r_2, r_4, w_6, r_8, r_9, w_10, r_11 \) in the first pass. In the 2nd pass, new candidates \( r_1, r_2, r_4, w_6, r_8, r_9, w_10, r_11 \) are added and one access, \( r_8 \) is removed, while write candidates \( (w_9, w_{10}) \) are moved to read-write candidate set.

Besides the space to store traced accesses into a trace file, the space complexity of our algorithm depends on the number of monitored shared variables \( V \), and the maximum parallelism \( T \) of the parallel program. The access history and effective keys require \( O(T) \) space for each shared variable, because each of them has at most two accesses for each thread. And the required space for candidate set is also \( O(T) \), so the total space complexity of our algorithm is therefore \( O(VT) \).

### 4. Conclusion

We present, in this paper, an efficient two-pass algorithm to detect first races in an particular execution of parallel program with random synchronization, by eliminating some dependent accesses to make a performance gain and analyzing the traced information by replaying the program. In the first pass, we collect filtered accesses involved in first races on a particular execution of a program, by filtering some affected accesses and collecting candidate sets which are subset of traced accesses. And then, we check concurrency of the candidate sets with the accesses from traced information and update them to extract occurred first races.

It could be required a lot of space and time to detect first race in one monitored execution of parallel programs, so our technique to detect first races is more practical with regard to the parallel programs with random synchronization.
References

Authors

Hee-Dong Park, he received M.S. degree in Computer Engineering from Pohang Institute of Science and Technology (POSTECH), and Ph.D. degree in Computer Science from Gyeongsang National University. He was a research staff in Electronics and Telecommunications Research Institute (ETRI). He is now an associate professor in Department of Information & Communications, JoongBu University. His research interests include computer network, parallel and distributed computing, embedded systems, and system software.

Yong-Kee Jun, he received the BS degree in Computer Engineering from Kyungpook National University, and the MS and PhD degree in Computer Science from Seoul National University. He is now a full professor in the Department of Informatics, Gyeongsang National University, where he had served as the first director of GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of GNU Virtual College. He is now the head of GNU Computer Science Division and the director of the GNU Embedded Software Center for Avionics (GESCA), a national IT Research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Prof. Jun is a member of Association for Computing Machinery (ACM) and IEEE Computer Society.