First Race Detection in Parallel Program with Random Synchronization using Trace Information

Hee-Dong Park¹, Yong-Kee Jun²
¹Joongbu University, Kumsan, Korea
hdpark@joongbu.ac.kr
²Gyeongsang National University, Jinju, Korea
jun@gnu.ac.kr

Abstract

Detecting data races in multi-threaded programs is a challenging problem in debugging, because the races could reveal nondeterministic program behavior in execution of such programs. Nondeterministic runtime effects of a race are hard to identify and it is difficult to decide whether the reported data races can appear or not in the actual program executions. Previous techniques for detecting races cannot provide method to locate first races in parallel programs with random synchronization including lock/unlock mechanism. This paper presents an algorithm which extracts first races by replaying the program and checking concurrency between sequenced traced data and candidate accesses, which are from a particular execution of parallel programs. We also present the correctness of our algorithm by showing that all the first races are included in the traced accesses which are composed of key accesses.

Keywords: first race, race detection, post-mortem analysis, parallel program debugging, concurrent programming, random synchronization

1 Introduction

Concurrent programs are difficult to debug because of potential nondeterminant conditions which leads to the problem of data races. Data races may exist in multi-threaded programs with concurrency errors, where the program fails to use proper synchronization when accessing the same location in shared memory, such that at least one of the accesses is a write.[1, 14] The data races are considered to be harmful and could exhibit unpredictable results in execution of parallel programs, and cause the behavior of programs to be different from users expectation on the same input, therefore must be detected and fixed for debugging parallel programs. Currently, multicore processors have become the default configuration not only for high performance servers, but also for personal computers and mobile computing devices. With increased using of multicore computers, the techniques that detect data races becomes significantly important.

Data race detection, where finding conflicting accesses and determining whether the conflicting events could have executed concurrently can be usually performed statically, or with the aid of run-time information. Static race detectors can be precise but currently suffer from an excess of false positives. Dynamic detection technique instruments the program...
to be debugged, and monitors an execution of the program. The monitoring process reports races which occur during the monitored process for each variable involved, but may still produce false warnings or miss races. Post-mortem [4, 12] and on-the-fly methods [3, 8, 11, 19, 21] improve race detection accuracy by collecting information from actual executions. This information is either analyzed after (post-mortem techniques) or during (on-the-fly methods) execution. The monitored programs provide complete information about memory locations accessed, so data conflicts can be detected accurately.

Determining whether the conflicting events executed concurrently can be accomplished either through happens-before relation or lockset algorithm at runtime. Happens-before approach uses vector clock approaches to record logical times of memory accesses. This approach is precise and results no false positives. However, it has high overhead in monitoring all memory accesses. Lockset-based approaches assume that every access to shared data must be protected by locks in order to make data race freedom. This approach generally performs with low overhead than happens-before approach, but is still imprecise and can report false alarms. A number of approaches have worked to reduce these weak points, such that combining happens-before relation with lockset algorithm [9, 15] and using more efficient labeling approaches [20]. 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. Although it is NP-hard to identify accurately the existence of data races between a pair of memory operations, there are many techniques to reduce the false positives [18].

The races that occur first (first races) are the ones between two accesses that are not causally preceded by any other accesses also involved in races. The first races are important in debugging because the removal of such data races may make other races disappear. It is even possible that all races reported would disappear once the first races are solved. Previous techniques to detect the first races have focused on the parallel programs without synchronizations [10, 20], with ordered synchronizations [11] or just collect accesses involved in first races [21]. These techniques still have limits to the scope of parallel programs in detecting first data races, or cannot provide method to find the final first races from the information which have logical execution order of accesses in a parallel program.

To detect the first races for parallel programs with synchronizations, huge amount of information could be collected, stored and managed to decide the order and concurrency relationship between data. In parallel programs with ordered synchronization, any pair of the corresponding coordinations are on ordered procedure, and synchronization constructs are executed in the order in which iterations would be occurred in a sequential form of the loop. 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. It has been proved that detecting races in program executions that have synchronizations 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 practical in a particular execution of parallel programs with weaker synchronization. The main result of our technique is to detect the occurred first races by analyzing candidate accesses and traced information which were from on-the-fly execution of parallel programs.

The rest of the paper is organized as follows: Section 2 describes the background on pro-
grams with synchronization and on first race with some definitions related to our technique. We present, in section 3, our algorithm to extract first races from trace information and candidate sets and finally summarize our work.

2 Background

The concurrency relation among threads in an execution of shared-memory parallel program can be represented by a directed acyclic graph called POEG (Partial Order Execution Graph) [3] which captures the happens-before relation [1]. We will illustrate, in this section, the existing dynamic race detection methods, the notion of random synchronization in parallel programs with POSIX threads model [13] and some terms for first races.

2.1 Dynamic race detection

There has been a number of work on dynamic race detection techniques using replay systems [12, 18], whose purpose is to enable deterministic replay debugging. Our technique 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.

The techniques based on the happens-before relation [11, 10] is that they primarily report no false positives, it means if it reports a data race, then there exists at least one access where the other accesses execute concurrently. Since we use timing relations for detecting races, it can handle any synchronization primitives including post/wait, lock/unlock, semaphores, etc [1]. A disadvantage of lockset-based techniques [9] is that they are not applicable to the parallel programs which use other synchronization primitives than locks, such as semaphores. This is because lockset algorithm is based on the ownership of locks by thread, which makes it difficult for the lockset-based techniques to infer which variables are supposed to be protected by those semaphores.

Dynamic techniques suffer from time and space overhead at runtime. Especially, it is known that race detection is NP-hard in general [5]. Therefore, how and where to focus detection efforts on the given programs is the key to realizing efficient detector. Space overhead is one of the hard challenges for detectors based on the happens-before relation, since it requires to maintain large amount of per-thread information, including memory location, access time, and locks. In contrast, lockset-based detectors enables simpler implementation, in that they need only information about set of locks for thread and shared memory. So there are many proposed hybrid race detector [15, 16] which combines lockset-based detection with a limited form of happens-before detection.

Dynamic techniques typically instrument existing binary programs and this incurs runtime overhead [17]. They usually instrument each load and store of shared memory locations, each call to locking and unlocking calls, and each initialization and allocation of memory. And this causes significant overhead for dynamic techniques.

2.2 Program with Random Synchronization

A lot of mechanisms are used for synchronization in parallel program written for shared memory systems. There are locks, semaphores, event variables and others for synchronization constructs. We consider parallel programs with nested/non-nested fork-join parallelism and synchronization events, in which the order of corresponding coordination pairs would
or would not occurred in a guaranteed sequence. For detecting first races in post-mortem analysis, tracing the order in which threads interact is sufficient to replay according to the total order of saved accesses in trace file for a parallel program. Our goal is to extend the coverage of existing technique for detecting occurred first races to the wide range of parallel programs with random synchronization events.

If two threads are coordinated by synchronization primitives such as post and wait, the thread that issues wait may not execute beyond the coordination point until the thread that issues post reached the corresponding point. On the other hand, the posting thread may proceed immediately. For programs that use post/wait synchronization with clear, the problem of detecting guaranteed ordering is NP-complete [6]. So we are considering simple post/wait without clear, where wait suspends the calling thread until the state of the corresponding post changes to posted, and do not reuse an event variable for synchronization. If more than one threads send signals to the other waiting threads, it will depend on the scheduling of which one of the post events will wakeup the waiting thread. Similarly, if multiple threads are waiting with the same priority, the waiting threads are released according to their scheduling priority [13].

For example, Fig. 1(a) shows ambiguity of matching random synchronization events, which one of the synchronization events, $P(post(a))$ or $W(wait(a))$, could trigger or be triggered by the corresponding event, $wait(a)$ or $post(a)$, respectively. In this figure, although the threads execute in parallel, the synchronization operations (such as S1, S2, S1*, and S2*) 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 Fig. 1(b), consider the unlocking function call in one thread in multithreading as sending a message by one process, and also consider the locking call in another thread as receiving the message in another process. This is because 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 in distributed systems. Therefore, the order of accesses could not be the same in one execution to that in another execution when executed repeatedly.

Fig. 2 shows a parallel program with lock/unlock mutual exclusion, as an example for random synchronizations, represented by a POEG, among which there are lock/unlock operations and some read and write accesses for a shared variable. Although the threads in Fig. 2 execute in parallel, the implicit synchronization operations (such as lock/unlock) are

Figure 1. Parallel programs with random synchronization
executed in a sequenced coordinated order according to the execution speed of each thread. Thus from each particular execution of the program, we can see the different results of races due to the inconsistency of happens-before between accesses along with the execution order of events.

POSIX threads (Pthreads) library [13], a particular threading implementation, can be found on many modern POSIX-compliant operating systems. The synchronization mechanisms found in the Pthreads library can be used to synchronize the execution of threads belonging to a process, taking into account the fact that the threads access in a concurrent way the common resources. The synchronization mechanisms presented are: the mutexes, the condition variables and the mechanism which ensures that some functions are executed only once. Each process can create as global variables its own synchronization mechanisms from the ones enumerated before. Because they are declared as global variables, these synchronization mechanisms are visible in all the threads belonging to the process, but they cannot be accessed by the threads of another process; that is why they cannot be used to synchronize the execution of processes. The mutexes are used to ensure mutual exclusion access to a certain resource of some threads. The condition variables are a specialized mechanism, which waits for a certain event inside the mutual exclusion region.

A condition variable in Pthread model can be used when it needs to schedule the way in which multiple threads access some shared memory, and can be used to synchronize threads among processes when they are allocated in memory that is shared by the cooperating processes. The **wait** event, `pthread_cond_wait()`, unlocks the mutex and blocks the thread. The **signal** event, `pthread_cond_signal()`, signals one thread out of the possibly sleeping threads to wakeup. Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable.

2.3 The First Races

There can be many races in an execution of a parallel program. The first races must be detected because they can potentially affect other races that occur later.
Definition 2.1 An access \( a_i \) happens before another access \( a_j \), denoted by \( a_i \rightarrow a_j \), if there exists a path from \( a_i \) to \( a_j \) in a POEG. It is concurrent with each other if there exist no paths between them, denoted by \( a_i \parallel a_j \).

Definition 2.2 If two accesses, \( a_i \) and \( a_j \) in which at least one of them is a write access, are concurrent, then the two events constitute a race denoted by \( a_i - a_j \).

Definition 2.3 An access \( a_j \) is affected by another access \( a_i \), if \( a_i \rightarrow a_j \) and \( a_i \) is involved in a race. A race \( a_i - a_j \) is unaffected, if neither \( a_i \) nor \( a_j \) is affected by any other accesses.

Definition 2.4 A tangle \( T \) is a set of partially affected races such that if \( a_i - a_j \) is a race in \( T \) then exactly one of \( a_i \) or \( a_j \) is affected by \( a_k \) such that \( a_k - a_l \) is also in \( T \).

Definition 2.5 An occurred first race or simply a first race is either an unaffected race or a tangled race.

Fig. 3(a) shows a parallel program with random synchronizations, among which there are synchronization events and accesses for a shared variable. Although the threads in Fig. 3(a) execute in parallel, the synchronization operations, where each one is composed of a pair of \( P \) and \( W \), are executed in a sequenced pair of S1-S2 or S1*-S2*. Thus from each particular execution of the program, we can see the different results of races due to the inconsistency of happens-before between accesses along with the execution pair of synchronization events.

3 Algorithm for First Race Detection

We first define, in this section, some terminologies to present our technique for extracting first races in an execution of parallel programs. The detection algorithm uses two sets of information, candidate accesses and trace information, where the candidate accesses are the filtered and key accesses collected using the algorithm in [21], and have at least one access which is involved in the first races. We replay the program using traced accesses and analyze it with candidate accesses to extract accesses of first races.

Definition 3.1 A write (read) access \( a_i \) is a key access if there does not exist any other write (read or write) access \( a_j \) within a block such that \( a_j \rightarrow a_i \).
**Definition 3.2** Block Keys for thread identifier $T$, denoted by $BK(T)$ is a set of key accesses in a block, and each has maximum two accesses (read and/or write).

Every $BK(T)$ is maintained as local variable of corresponding thread $T$, which can be free from mutual exclusion with shared memory of other threads. Every access in a block is compared to the accesses in the corresponding block keys by checking happens-before relation whether it can be a key access or not. If it is key access, it will be saved to corresponding block keys, otherwise returns.

**Definition 3.3** 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 with random synchronization, each key access within a block is stored in a consecutive order to a trace file. Therefore each access in trace file can be used to replay the parallel programs with total order.

**Definition 3.4** A read (write) access $a_i$ 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. A write access $a_i$ is a read-write candidate, if $a_i$ is involved in a race and there exists a read candidate $a_h$ which happened before $a_i$, and there exists no other write access $a_x$ such that $a_x \rightarrow a_i$ and $a_h \rightarrow a_x$.

**Definition 3.5** Candidate Set for a shared variable $X$, denoted by $CS(X)$, is a set of candidate accesses 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.

Fig. 3(b) shows a particular execution of parallel program as shown in Fig. 3(a), represented as POEG only with traced accesses because it is sufficient to represent the order dependencies of accesses during replay. Consider a case where synchronization operation $S1$ occurs first prior to $S1*$ at the point of $P$ in thread $T1$ (thus synchronization operations

---

**Figure 4. Structure of first race detection algorithm**
S1* and S2* will not be occurred). This particular execution of this program results in the sequenced pair of synchronization events S1-S2 as shown in the Fig. 3(b). We can get the candidate accesses as \((w_4, r_5)\) from the algorithm of [21], and can also get the traced accesses as \((r_0, r_2, r_3, w_4, r_5, w_6)\) which have total order of a particular program execution. On the other hand, for a sequenced pair of synchronization events S1*-S2*, we can get the candidate accesses as \((w_4, r_5, w_6)\) with the same traced accesses in the case of S1-S2.

As shown in Fig. 4, from both set of candidate and traced accesses, where candidate set have at least one access which is involved in the first races, we will extract and add the actual corresponding accesses which will construct final candidate set of first races, if exist. We process and analyze the candidate accesses, therefore, with replaying the program according to traced accesses in a trace file to make the candidate set. The algorithm is as follows:

Algorithm 1 We use two sets of candidates(read candidate set CS(X,R) and write candidate set, CS(X,W)) and ordered accesses from Traced Accesses(current), and update candidate sets while creating read-write candidate set, CS(X,RW) if needed:

1. Determine candidate accesses: (1) For all accesses in CS(X), if there exists no race between accesses in candidate set and current access, then return; (2) If current is read access, for all accesses in CS(X,R), if there exists any read candidate in CS(X,R) which happens-before current access, then return. If current access is happens-before any access \(e_r\) in in CS(X,R), then delete the access \(e_r\) in CS(R). (3) In case of the write current, determine if the current is a read-write candidate by checking CS(X,R), and for all accesses in CS(X,W), delete the access \(e_w\) if current is equal to the write access \(e_w\) in CS(W).

2. Update candidate set: If the current is a new candidate, then add the current to the corresponding set of CS(X), or if the current is a read-write candidate, then move the current from CS(X,W) to CS(X,RW).

3. Skip: Skip the current, if the current is a write or a read-write candidate.

In step 1, 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. Step 2 updates the candidate sets which have the first races to occur. 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. Recall that accesses in candidate set are \(w_4, r_5\), and traced accesses are \(r_0, r_2, r_3, w_4, r_5\) and \(w_6\) from Fig. 3(b). The first read access \(r_0\) from traced accesses will be returned because there is not any access in candidate set which is concurrent with this access \(r_0\). Next read access \(r_2\) will be saved into CS(R) because it is concurrent with \(w_4\) which was already in CS(W). The access \(r_3\) will be returned, and write access \(w_4\) will be still in CS(RW) because it is concurrent with \(r_2\) and \(r_5\). Next access \(r_5\) is also in CS(R) because of the concurrency with \(w_4\). The last write access \(w_6\) from traced accesses is returned because there already exist a write candidate \(w_4\) which is happened before this access. After this, we can see three candidate accesses in candidate set \((r_2, w_4, r_5)\) for first races as \(r_2-w_4\) and \(w_4-r_5\). If we replay and analyze the other case for a pair of synchronization events S1*-S2* in Fig. 3(a), we can get the first races as \(r_2-w_4\), \(r_3-w_6\) which are tangled races.
Theorem 1 If there exists any first race, all reported accesses in the first races are also included in the traced accesses.

[Proof] Each access in candidate set is an effective key access which can be involved in race during program execution, and the first race is unaffected or tangled. If there exist any races, all 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. Therefore, the accesses which are not key have been discarded in the processing of the first race determination. This means that only the key accesses will be involved in the first races. And traced accesses(TA) is a sequence of key accesses saved in trace file by Def. 3.3. Therefore, any access in the first races should be the subset of traced accesses. Q.E.D.

4 Conclusion

Synchronization is important for a parallel or multithreaded programming that uses dependence analysis as the basis for the correctness of parallel constructs. Parallel programs with random synchronization events cannot guarantee sequence of events in one execution to that in another execution when executed repeatedly, which could make parallel program debugging more difficult.

In this paper, we present an algorithm which detects occurred first races using the trace information and candidate accesses from a particular execution of parallel programs with random synchronization. Our technique checks concurrency of the candidate accesses with the traced information by replaying the parallel program and update them to extract occurred first races. We also present the correctness of our algorithm by showing that all the first races are included in the traced accesses which are composed of key accesses. Future work includes improvement of detection efficiency and showing practical use of our technique by experiment and evaluation.

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 his Ph.D. degree in Computer Science from Seoul National University. He was a research associate in Computer Science at Univ. of California, Santa Cruz, and a research staff in Electronics and Telecommunications Research Institute (ETRI). Dr. Jun is now a full professor of Gyeongsang National University (GNU) where he served as the founding professor of Dept. of Informatics. He is now the director of GNU Embedded Software Center for Avionics which is a national IT Research Center of South Korea. His research interests include parallel/distributed computing, embedded systems, and systems software. His research has focused primarily on reliability of concurrent software.