SCIENCE CHINA
Information Sciences

• REVIEW •

March 2017, Vol. 60 031101:1–031101:27
doi: 10.1007/s11432-015-0203-2

Surveying concurrency bug detectors based on types of detected bugs

Zhendong WU[1,2], Kai LU[1,2*] & Xiaoping WANG[1,2]

[1]*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China;*
[2]*College of Computer, National University of Defense Technology, Changsha 410073, China*

Received October 10, 2015; accepted February 3, 2016; published online October 13, 2016

Abstract Concurrency bugs widely exist in concurrent programs and have caused severe failures in the real world. Researchers have made significant progress in detecting concurrency bugs, which improves software reliability. In this paper, we survey the most up-to-date and well-known concurrency bug detectors. We categorize the existing detectors based on the types of concurrency bugs. Consequently, we analyze data race detectors, atomicity violation detectors, order violation detectors, and deadlock detectors, respectively. We also discuss some other techniques which are mostly related to concurrency bug detection, including schedule bounding techniques, interleaving optimizing techniques, path expanding techniques, and deterministic replay techniques. Additionally, we statistically analyze the reviewed detectors and get some interesting findings, for instance, nearly 86% of previous detectors focus on data races and atomicity violations, and dynamic approaches are popular (74%). We also discuss the limitations of previous detectors, finding that 91% of previous detectors suffer from false negatives and 64% of previous detectors suffer from runtime overhead. Based on the reviewed detectors and statistical analysis, we conclude some future research directions, including accuracy, performance, applicability, and integrality.

Keywords concurrency bug detection, data race, atomicity violation, order violation, deadlock

Citation Wu Z D, Lu K, Wang X P. Surveying concurrency bug detectors based on types of detected bugs. Sci China Inf Sci, 2017, 60(3): 031101, doi: 10.1007/s11432-015-0203-2

1 Introduction

Multicore technology is making concurrent programs increasingly pervasive. Unfortunately, concurrency bugs widely exist in concurrent programs and have caused severe damages such as Northeastern Blackout of 2003 [1, 2]. Concurrency bugs are particularly difficult to be detected due to their unique nondeterminism. Unlike the bugs in sequential programs, concurrency bugs are triggered due to a combination of two conditions. First, they require the right set of program inputs, which is difficult to find. Second, they require the right thread interleaving, which is hidden in huge interleaving space.

Fortunately, many researchers have focused on concurrency bug detection. They have proposed many effective detectors to detect concurrency bugs. This paper aims to review the most recent and important detectors based on the types of concurrency bugs, and tries to find some future research directions.

In this section, we first discuss several previous surveys. Then, we describe the types of concurrency bugs in detail, and discuss the limitations of concurrency bug detectors and the methodology which is used for selecting reviewed papers. Finally, we conclude the major contributions of this study.

[*] Corresponding author (email: kailu@nudt.edu.cn)

1.1 Previous surveys

Recently, many effective detectors have been proposed to detect concurrency bugs. Previous surveys [3,4] categorize the existing detectors into different types. Specifically, Abdelqawy et al. [3] categorize the detectors based on the techniques, including race free-type technique, static analysis technique, dynamic analysis technique, model checking technique, and hybrid technique. In their categorization, the static detectors, dynamic detectors, and hybrid detectors are reviewed in detail. These three types of detectors are more popular than the other two types. The race free-type technique used in detectors can prevent and detect concurrency bugs. The detectors that use model checking technique are always inefficient for large programs. Raza [4] reviews a lot of detectors that focus on data races. In the paper [4], the detectors are categorized as on-the-fly, ahead-of-time, and post-mortem techniques. Actually, the detectors are categorized into static detectors, dynamic detectors, and hybrid detectors.

Static detectors [5–11] always analyze the concurrent program without executing it. They can immediately detect the concurrency bugs in obscure code paths that are difficult to reach during dynamic execution. Therefore, they can reason about multiple program paths during analysis, and typically do not miss bugs (i.e., have low rate of false negatives). Some static detectors analyze concurrent program based on programmers' annotations. The programmers use annotations to assume correct thread schedule, and the detectors statically check whether the programmers' intention is violated. Some static detectors build a control-flow graph of the entire program and use a flow-sensitive method to analyze the program, checking whether the shared variables may be accessed by multiple threads and those accesses may lead to incorrect behaviors. However, being imprecise in nature, static detectors tend to report many false positives, which consumes manual effort to identify true concurrency bugs.

Dynamic detectors [12–22] typically monitor memory accesses and synchronization operations at runtime, and determine if the observed memory accesses can trigger concurrency bugs. Some dynamic detectors analyze program when the program is executing, and detect all concurrency bugs after the program is stopped. Some dynamic detectors record the information of program execution, and detect all concurrency bugs through analyzing the recorded information. The dynamic detectors have a precise knowledge about the runtime behavior of the program so that they can report few false positives. Also, they can detect all the concurrency bugs that are seen in the observed execution paths. However, dynamic detectors may monitor a tiny subset of a program's possible execution paths, which makes some concurrency bugs not detected.

To alleviate the limitations of static detectors and dynamic detectors, researchers try to combine static and dynamic analysis to leverage the advantages of both [23–26]. The hybrid detectors always have two main phases: static analysis phase and dynamic analysis phase. The static analysis phase can statically analyze the program and get some information to help dynamic analysis. Some researchers have tried to combine static analysis and model checking to analyze concurrent programs. The model checking dynamically analyzes all the program states to detect concurrency bugs. However, it fails to scale for large programs due to the exponential increase in state space with execution length. Researchers use static analysis to reduce the state space, which can improve the efficiency of dynamic analysis. Some hybrid detectors use static analysis to predict potential concurrency bugs. Since the static analysis cannot accurately detect concurrency bugs, the results of static analysis can be used to help dynamic analysis. The hybrid detectors dynamically verify the results of static analysis, identifying the real concurrency bugs. Benefiting from the information of static analysis, hybrid detectors can dynamically analyze fewer thread schedules, improving the efficiency. However, hybrid detectors cannot guarantee that all the potential concurrency bugs are verified in actual executions, which may produce false negatives.

Additionally, Fiedor et al. [27] provide a uniform taxonomy of concurrency bugs to help understand the bugs and develop techniques for detecting them. They also review some detectors which can detect different types of concurrency bugs. Their categorization of concurrency bugs is similar to ours. However, they only give a definition of each type of concurrency bug, and we describe the characteristics of each type of concurrency bug in detail. For each type of concurrency bug, Fiedor et al. [27] review the most important detectors that can effectively detect the specific type of concurrency bug. However, the reviewed detectors are not further categorized. Moreover, the limitations of the reviewed detectors are not discussed. To help the researchers to understand previous concurrency bug detectors more clearly, we describe each type of detector in detail and discuss the limitations of each one. We also try to statistically analyze a lot of detectors, and obtain some interesting findings for researchers and developers.

Overall, previous surveys [3, 4, 27] only analyze each concurrency bug detector. Moreover, the limitations of concurrency bug detectors are not discussed in detail. This paper tries to review the concurrency bug detectors, and statistically analyzes the methodologies and limitations of previous concurrency bug detectors. According to statistical analysis, we get some interesting findings which may provide some suggestions for future research.

1.2 Characteristics of different types of concurrency bugs

Many concurrent programs use various locks to coordinate how their threads produce program outputs. Improper sequences of lock acquisitions and releases performed by these threads may result in data races, atomicity violations, or deadlocks. In addition, the assumption of a certain order between two operations from two threads may be violated due to non-determinism. Consequently, data races, atomicity violations, order violations, and deadlocks may exist in the same concurrent programs. To avoid missing concurrency bugs, we categorize the concurrency bugs into four types: data race, atomicity violation, order violation, and deadlock.

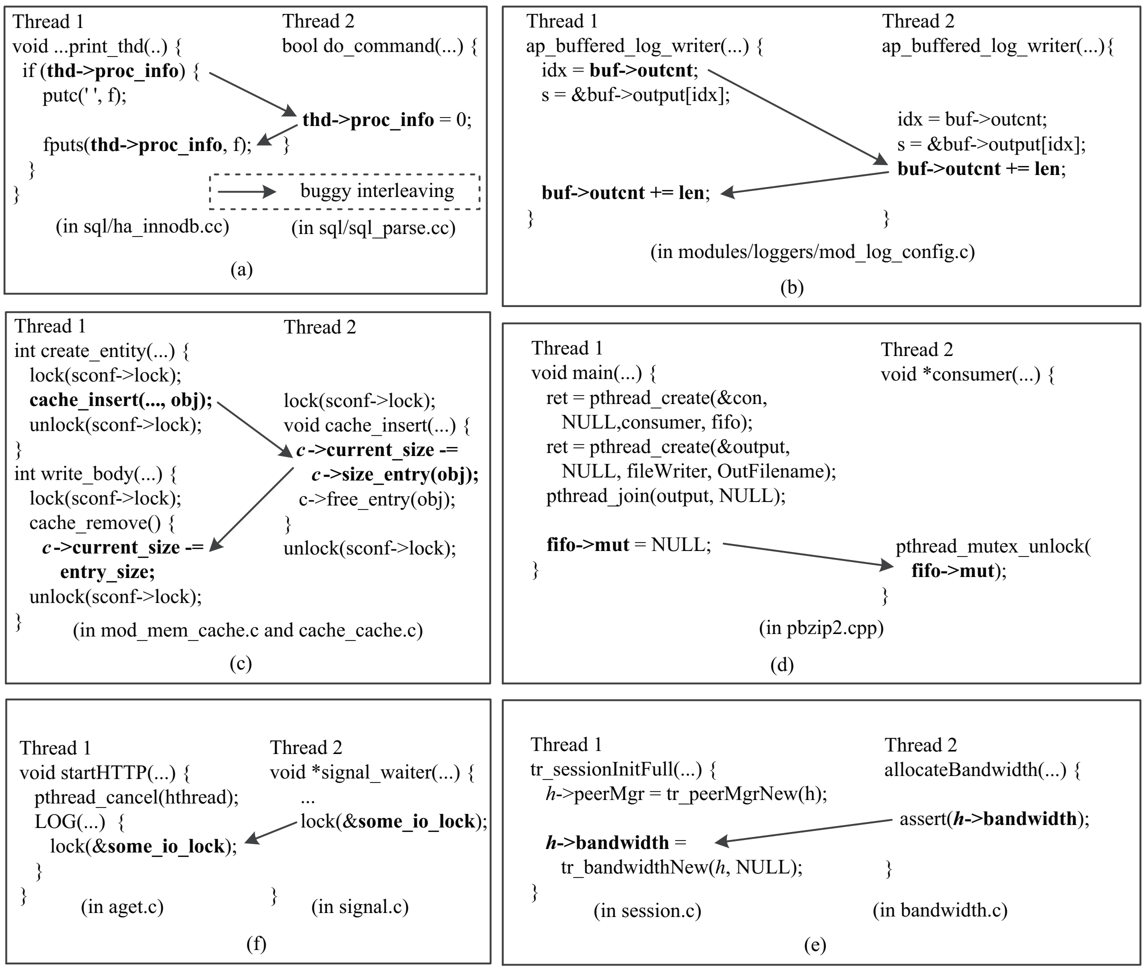
This categorization is different from [28]. Lu et al. [28] separately study two types of concurrency bugs: deadlock bugs and non-deadlock concurrency bugs. Specifically, they carefully examine the bug patterns, manifestations, fix strategies and other characteristics of concurrency bugs. Since the data races cannot reflect programmers' order intention, and benign data races always do not compromise program's correctness, they do not examine the data race bugs. Our categorization shows more detail on concurrency bug detection, and provides more knowledge about the concurrency bug detection, especially the data race detection. For instance, the data races can be classified into benign races and harmful races, and the harmful races are real bugs that should be fixed before software release. Therefore, it is useful to discuss data race detectors. Additionally, almost all (97%) of the non-deadlock concurrency bugs are actually two types: atomicity violations and order violations [28]. We separately discuss atomicity violation detectors and order violation detectors, which can help researchers to understand the targets of detectors. The goal of the categorization in [28] can help researchers to understand the characteristics of concurrency bugs and design new detectors. Our goal of categorization can help researchers to understand the targets of previous concurrency bug detectors and design new detectors that can complement previous work.

To describe different types of concurrency bugs clearly, we collect some real world concurrency bugs. Figure 1 shows six code snippets from open source softwares (MySQL, Apache, pbzip2, aget-devel, and transmission).

Figure 1(a) is not only a data race bug but also an atomicity violation bug. This bug is from MySQL-4.0.19. The function first checks whether thd->proc_info is NULL or not. If not, the function will write the info to output. However, the check and the use are not enforced, and it is possible that thd->proc_info is nullified by other threads between the check and the use, leading to a crash of MySQL server.

Figure 1(b) also shows not only a data race bug but also an atomicity violation bug. This bug is also from MySQL-4.0.19. In the function, buf->outcnt is the index to the end of the log buffer. It is read, checked, and then used to help append the log buffer. After the log buffer updates, the buf->outcnt is increased. However, the whole process is not protected by any lock, and thread 1's read-write sequence may falsely mingle with thread 2's read-write sequence to the same shared variable, leading to program's misbehavior.

Figure 1(c) is an atomicity violation bug (data race free). This bug is from Apache-2.0.46. The three memory accesses are protected by the common lock. Programmers assume that thread 1 will execute function write body after cache insert. However, such an atomicity can be violated. When cache_insert invokes c->free_entry to deallocate the object, the reference count is not zero since there is a reference from cache_handle. However, current_size will be decreased again if cache_insert is invoked in other



(a)

```
Thread 1                              Thread 2
void ...print_thd(..) {               bool do_command(...) {
 if (thd->proc_info) {
   putc(' ', f);
                                        thd->proc_info = 0;
   fputs(thd->proc_info, f);           }
 }
}                                     ⟶ buggy interleaving
   (in sql/ha_innodb.cc)              (in sql/sql_parse.cc)
```

(b)

```
Thread 1                              Thread 2
ap_buffered_log_writer(...) {         ap_buffered_log_writer(...){
  idx = buf->outcnt;
  s = &buf->output[idx];
                                        idx = buf->outcnt;
                                        s = &buf->output[idx];
                                        buf->outcnt += len;
  buf->outcnt += len;
}                                     }
          (in modules/loggers/mod_log_config.c)
```

(c)

```
Thread 1                              Thread 2
int create_entity(...) {
  lock(sconf->lock);
  cache_insert(..., obj);             lock(sconf->lock);
  unlock(sconf->lock);                void cache_insert(...) {
}                                       c->current_size -=
int write_body(...) {                     c->size_entry(obj);
  lock(sconf->lock);                    c->free_entry(obj);
  cache_remove() {                    }
    c->current_size -=                unlock(sconf->lock);
      entry_size;
  unlock(sconf->lock);
}    (in mod_mem_cache.c and cache_cache.c)
```

(d)

```
Thread 1                              Thread 2
void main(...) {                      void *consumer(...) {
  ret = pthread_create(&con,
    NULL,consumer, fifo);
  ret = pthread_create(&output,
    NULL, fileWriter, OutFilename);
  pthread_join(output, NULL);

  fifo->mut = NULL;                     pthread_mutex_unlock(
}                                         fifo->mut);
                                      }
              (in pbzip2.cpp)
```

(f)

```
Thread 1                              Thread 2
void startHTTP(...) {                 void *signal_waiter(...) {
  pthread_cancel(hthread);              ...
  LOG(...)  {                           lock(&some_io_lock);
    lock(&some_io_lock);
  }
}      (in aget.c)                    }    (in signal.c)
```

(e)

```
Thread 1                              Thread 2
tr_sessionInitFull(...) {             allocateBandwidth(...) {
  h->peerMgr = tr_peerMgrNew(h);
                                        assert(h->bandwidth);
  h->bandwidth =
    tr_bandwidthNew(h, NULL);         }
}
      (in session.c)                  (in bandwidth.c)
```

Figure 1 Different types of concurrency bugs from open source softwares. (a) A data race bug and also an atomicity violation bug; (b) a data race bug and also an atomicity violation bug; (c) an atomicity violation bug; (d) an order violation bug; (e) an order violation bug; (f) a deadlock bug.

threads. When cache_remove is called in write_body for the object, the size is actually decreased twice, causing negative cache size and later crashing the server.

Figure 1(d) is an order violation bug from pbzip2 which is a parallel implementation of bzip2 block-sorting file compressor. The main thread waits for the consumer threads to finish and then releases mutexes and queue data structures. However, when the main thread tries to free all the resources, it is possible that some consumer threads still have not exited yet. The main() will free fifo->mut. However, the consumer threads might still need fifo->mut. Therefore, a segmentation fault will be thrown.

Figure 1(e) is also an order violation bug from transmission which is a multi-thread BitTorrent download client. Function allocateBandwidth is called from an event callback function which is supposed to be called 500 milliseconds after t->peerMgr is created. This is the reason why programmer assumes that variable h->bandwidth is always initialized first. However, this order is not enforced by synchronization primitives, one of the two operations may be executed faster (or slower) than the programmer's assumption. As a result, assert(h->bandwidth) is likely to be executed before the initialization and read an uninitialized variable, causing an assertion failure.

Figure 1(f) is a deadlock bug due to asynchronous cancel thread from aget-devel. This bug is hard to reproduce even if delays are injected. The bug happens when the download is completed. Thread 1 (main thread) will cancel the signal handler thread. It invokes pthread_cancel(hthread) to cancel the thread 2 (signal handler thread). Since thread 2 sets its cancel type to asynchronous, it is likely that it is cancelled when it is holding a lock. Hence, thread 1 will get stuck when it tries to get the same lock.

As a result, a deadlock bug happens.

Overall, the characteristics of each type are briefly described below.

• **Data race.** A data race occurs when two threads are about to access the same piece of memory, and at least one of the two accesses is a write, and the relative ordering of the two accesses is not enforced using synchronization primitives, like mutexes.

• **Atomicity violation.** Atomicity is a property for the concurrent execution of several operations when their data manipulation effect is equivalent to that of a serial execution of them. An atomicity violation occurs when the assumed atomicity is broken if the code region is unserializably interleaved by accesses from another thread.

• **Order violation.** Programmers may assume a certain order between two operations from two threads. An order violation occurs when one operation A should execute before another operation B, but this is not enforced by the program.

• **Deadlock.** The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. A deadlock occurs when a set of threads block forever because each thread in the set is waiting to acquire a lock held by another thread in the set.

Researchers have proposed different detectors to detect different types of concurrency bugs. The detectors can complement each other to detect multiple types of concurrency bugs. For instance, programmers can first use RaceFuzzer [13] to detect data races in concurrent programs. Then, they can use Ctrigger [12] and DeadlockFuzzer [29] to detect atomicity violations and deadlocks, respectively. Finally, they can combine the results of these detectors. However, using several detectors separately to detect multiple types of concurrency bugs may increase detecting time, and understanding formats of results from separate detectors may need additional effort. Fortunately, some general detectors [15, 17–20] are proposed to detect multiple types of concurrency bugs, which can detect not only data races, but also other types of concurrency bugs.

1.3 Limitations of concurrency bug detectors

Although many concurrency bug detectors have been proposed, several fundamental limitations still exist. We summarize the limitations in four parts: false negatives, false positives, runtime overhead and manual effort. The false negative and false positive are used to evaluate the accuracy of bug detectors. For instance, some static detectors such as RELAY [5] can analyze the whole program statically and produce few false negatives. Some dynamic detectors such as RaceFuzzer [13] can expose the harmful bugs in actual executions and produce few false positives. Runtime overhead is used to evaluate the performance of bug detectors. For instance, Maple [15] is a dynamic detector which imposes 45.4x overhead when it is applied to pbzip2. Manual effort means that bug detectors need programmers to add some annotations in source code or check the results reported by the detectors. For instance, RELAY [5] reports many false positives which should be identified manually. The main reasons for each limitation are briefly described below.

• **False negatives.** Many dynamic detectors [12, 13] cannot explore obscure execution paths which may rarely occur in actual executions (some concurrency bugs may be missed), producing false negatives. Some detectors [30] use sampling strategies to improve performance, which may cause detectors to produce false negatives.

• **False positives.** Static detectors [5–11] always detect concurrency bugs without actual execution. They cannot accurately determine whether two memory accesses alias or not, and also cannot accurately infer happens-before relationship. Therefore, static detectors are notorious for reporting a large number of false positives.

• **Runtime overhead.** Dynamic detectors [12–21] require to insert additional instructions to find potential interleavings or verify potential interleavings, imposing runtime overhead (e.g., 200x for Intel ThreadChecker [31]). To trigger concurrency bugs, dynamic verification needs to re-execute the program to test potential interleavings, which imposes runtime overhead due to controlling thread schedule.

• **Manual effort.** Some detectors report many false positives (e.g., 84% of races reported by RELAY [5] are not true races) so that the results require manual inspection to identify true bugs. Several detectors require manual annotations to help detectors to detect bugs (e.g., ThreadSafety [32]). Additionally, some commercial detectors report buggy interleavings which are too difficult to explain to developers [33].

1.4 Methodology in selecting related papers

Many researchers have done a lot of work in the area of concurrency bug detection. This paper tries to survey concurrency bug detectors based on the types of detected bugs. Consequently, the methodology in selecting related papers follows two principles.

1) The selected papers should cover four types of concurrency bugs. Since we need to survey the concurrency bug detectors based on types of bugs, the selected papers should focus on detecting data races, atomicity violations, order violations, and deadlocks. To avoid missing the most important research, we even check the references of the selected papers.

2) The selected papers should be most up-to-date and well-known. We try to select the most recent papers which are related to concurrency bug detection. Moreover, we try to make the related papers selected from some well-known conferences and journals (e.g., OSDI, SOSP, ASPLOS, PLDI, FSE, ICSE, etc.). Additionally, several valuable technical reports are also selected.

1.5 Contributions of this survey

In this paper, we make the following contributions.

1) We survey concurrency bug detectors based on the types of concurrency bugs. We categorize the existing detectors into data race detectors, atomicity violation detectors, order violation detectors, and deadlock detectors. This categorization can help programmers to debug and test concurrent programs. For instance, programmers can understand the concurrency bugs more easily if they know the target of the detector. Moreover, researchers can also know what types of concurrency bugs are studied and get some findings to design new detectors which can complement previous detectors.

2) We statistically analyze the targets and methodologies of previous concurrency bug detectors, getting some interesting findings. Unlike previous surveys [3, 4, 27] which only describe each concurrency bug detector, we statistically analyze the previous detectors, finding that 86% of the reviewed detectors focus on data races and atomicity violations. We also find that a large portion (74%) of the reviewed detectors use dynamic analysis techniques. These findings may guide researchers to do some future work which may complement existing detectors.

3) We statistically analyze the limitations of previous concurrency bug detectors in detail. Unlike previous surveys [3, 4, 27] which only point out the limitations of each detector simply, we statistically analyze the limitations of the reviewed detectors, finding that most (91%) of the reviewed detectors suffer from false negatives and 64% of the reviewed detectors suffer from runtime overhead. These findings may guide researchers to do some future work to alleviate the primary limitations. Also, the findings can help the programmers to use multiple detectors that complement each other to test concurrent programs.

4) We conclude several future research directions in concurrency bug detection. Based on the survey of previous concurrency bug detectors and statistical analysis, we discuss the future research of concurrency bug detection, including accuracy, performance, applicability, and integrality. The discussion of future research may guide researchers to design new algorithms or new detectors.

In the remainder of this paper, we review the previous data race detectors (Section 2), atomicity violation detectors (Section 3), order violation detectors (Section 4), and deadlock detectors (Section 5). We discuss some techniques which are related to concurrency bug detection (Section 6). We statistically analyze the targets and methodologies of previous detectors, and discuss the limitations of previous detectors (Section 7). We also discuss the future research directions in concurrency bug detection (Section 8). Finally, we conclude this paper (Section 9).

2 Data race detectors

In this section, we categorize the data race detectors into three parts: static detectors [5, 7, 10, 11, 34–41], dynamic detectors [13, 21, 42–47], and hybrid detectors [25, 26, 48–50].

2.1 Static detectors

Static race detectors do not require actual executions, which can provide significant advantages for large code bases. Therefore, they may be suitable for operating systems where a large part of code never executes [5]. Additionally, static race detectors can easily detect races in obscure code paths which are difficult to reach in actual executions. This feature can make static detectors produce few false negatives. We review the static race detectors based on type, lockset, and model checking, respectively.

2.1.1 *Type-based detectors*

The type-based detectors focus on specifying synchronization discipline by means of types [36, 38, 40]. For instance, EPAJ is a type system for specifying and verifying data races [36]. Although EPAJ is completely automatic, manual effort is still required to check the results when it is applied to large programs. Additionally, many type-based detectors suffer from the expressiveness of implemented type system and the burden of annotations.

ThreadSafety [32] is an annotation-based race detector which can detect data races caused by inconsistent protection of a shared variable. However, ThreadSafety can only detect data races that are on annotated variables, which may produce false negatives.

2.1.2 *Lockset-based detectors*

The lockset-based detectors use inter-procedural analysis to detect data races [5, 6, 11, 39]. RELAY is a lockset-based race detector [5]. It performs its analysis in a bottom-up way through program's control flow graph while computing function summaries that summarize which variables are accessed and which locks are held in each function. It detects a data race whenever it sees at least two accesses to the same memory location, and at least one of the accesses is a write, and the accesses are not protected by at least one common lock. In most cases, RELAY is complete (i.e., does not miss data races). However, being imprecise in nature, RELAY reports many false positives (e.g., 84% of the races are not true races). Therefore, manual effort is required to identify true races from false positives.

2.1.3 *Model-checking-based detectors*

The model-checking-based detectors are typically path-sensitive so that they may be not suitable for large programs. KISS is an automated checker for multi-threaded C programs [7]. It transforms a concurrent program into a sequential program that simulates the execution of behaviors. The transformed sequential program is then analyzed by a tool that only needs to understand the semantics of sequential program. KISS uses model-checking technique to analyze a subset of the behaviors of concurrent program. However, the problem of huge state space of a model of large concurrent program exists in KISS.

2.2 Dynamic detectors

Unlike static race detectors, dynamic race detectors typically monitor memory accesses and synchronization operations at runtime, and check if the observed accesses race with each other [13, 21, 42–47, 51–56]. Due to limited execution paths, some data races which are not exposed in actual executions are missed. In addition, high runtime overhead is also imposed due to instrumentation. We review five kinds of dynamic detectors in detail. These five kinds of detectors target detecting data races, detecting harmful races, classifying data races, improving the performance of detection, and detecting races for smartphones, respectively.

2.2.1 *Detecting data races*

ThreadSanitizer [32] (TSAN) is developed by Google and has two versions: TSAN v1 and TSAN v2. TSAN v1 is implemented as a binary instrumentation tool. It is based on lockset and happens-before relation, trying to improve coverage and may report false positives. TSAN v2 is a compiler-based instrumentation pass in LLVM [57]. TSAN v2 imposes roughly 5-15x runtime overhead, which is much lower than TSAN v1 (20-300x overhead). TSAN v2 is based on happens-before relation so that it has few false positives. ThreadSanitizer can effectively detect data races, including benign races and harmful races.

2.2.2 *Detecting harmful races*

RaceFuzzer [13]. The benign races would not affect the behavior of program execution. The harmful races which may lead to program failures are real bugs that should be fixed before software release. RaceFuzzer [13] is a randomized dynamic race detector that can separate harmful races from false warnings without manual effort. It first uses an imprecise race detector to compute a set of pairs of statements that could potentially race in a concurrent execution (potential races). It then executes the program with an active thread scheduler, creating real race conditions, and checking whether the potential races could lead to program failures. However, being dynamic in nature, RaceFuzzer cannot dynamically check all the potential races in limited execution paths, which makes RaceFuzzer produce false negatives.

RaceChecker [58]. Since RaceFuzzer dynamically verifies one data race in one execution to check whether this race is harmful, the efficiency may be affected. To improve efficiency, RaceChecker can dynamically verify multiple data races in one execution. RaceChecker first uses previous techniques to detect data races, including benign races and harmful races. Then, it groups all the data races into N parts according to the logic clocks of two elements of each data race. The data races in the same group do not interfere with each other so that RaceChecker can verify all the data races in one group during one execution. By grouping, the number of executions during race verification phase is reduced significantly, which makes RaceChecker detect harmful races more efficiently.

2.2.3 *Classifying data races*

To improve the accuracy of classifying data races, Portend [42] uses multi-path multi-schedule analysis to dynamically check all the data races. Previous tools [59] that can classify data races are not accurate. To check whether a race is harmful or benign, they allow the primary and alternate executions to run independently and check the consequences of these two executions. If either of these two executions crashes, the race is determined as a harmful race. Kasikci et al. observe that it is not certain that the race is benign if the primary execution and alternate execution behave identically. Therefore, they propose Portend which can explore more paths and more schedules to achieve good accuracy. Comparing with previous tools, Portend can classify data races more accurately. However, Portend uses symbolic execution which is an inefficient technique, so the efficiency of identification may be affected due to a large number of paths and schedules. Therefore, it is a tradeoff between efficiency and accuracy.

2.2.4 *Improving performance of race detection*

To speed up data race detection, Wester et al. [43] propose two parallel race detectors (lockset-based and happens-before-based) which can spread the process of detection across multiple cores. These two parallel detectors rely on uniparallelism and execute time intervals of a program (called epochs) in parallel. They execute all threads from a single epoch on a single core to eliminate locking overhead. The process of detection is divided into three phases: the epoch-sequential phase predicts a subset of analysis state, the epoch-parallel phase carries out the bulk of analysis, and the commit phase resolves symbolic expressions logged during the epoch-parallel phase. The parallel race detectors significantly speed up race detection. However, due to instrumenting many instructions, runtime overhead is a challenge for these parallel detectors.

2.2.5 *Detecting data races for smartphones*

Currently, smartphones have provided programming environments for efficient and feature-rich applications which combine multi-threading and asynchronous event-based dispatch. Therefore, thread interleavings with non-deterministic reorderings of asynchronous tasks can lead to errors in the applications. Some detectors are designed to detect data races for smartphones [60, 61]. Maiya et al. propose a dynamic race detector called DroidRacer to detect data races for Android applications [60]. They formalize the concurrency semantics of Android programming model. DroidRacer generalizes happens-before relations for multi-threaded programs and single-threaded event-driven programs. It runs unmodified binaries on an instrumented Dalvik VM and instrumented Android libraries. One execution trace of the application is analyzed offline for data races by computing the happens-before relation. DroidRacer can effectively detect data races in popular Android applications.

2.3 Hybrid detectors

To alleviate the limitations of static detectors and dynamic detectors, many hybrid race detectors are proposed [25, 26, 48–50]. These detectors combine static and dynamic analysis to leverage the advantages of both. We review two hybrid detectors [25, 26] of them.

RaceMob [25]. To achieve both good accuracy and low runtime overhead, Kasikci et al. propose RaceMob which uses real-user crowdsourcing to dynamically verify data races. For a given program, RaceMob first uses RELAY [5] to statically analyze source code and find a large number of potential data races. Then, it uses dynamic information to prune false positives. Finally, to efficiently verify all the data races, it crowdsources the verification of data races to user machines that are anyway running the program. Due to the combination of static analysis and dynamic analysis, RaceMob can accurately detect data races and identify harmful races. RaceMob also imposes low runtime overhead due to infrequent instrumentation. By crowdsourcing race verification, RaceMob amortizes the cost of verification and gains access to real user executions. However, the user machines are not under control so that some obscure code paths may not executed (some potential races would not be verified).

ColFinder [26]. This is our previous work which can detect race-directed bugs by collaborative techniques. It first uses static analysis (RELAY [5]) to find the potential data races in source code. With respect to the shared variables accessed by the potential data races, it then uses static program slicing to extract many small programs (called slices [62]). Finally, it actively controls thread scheduler to create race conditions to verify whether program failure occurs in the executions of slices. It reports a harmful bug if the potential data race could lead to program failure. The major advantage of ColFinder is that it uses slicing to improve the efficiency of bug detection. However, the capability of dynamic verification may be affected by the limited execution paths. (Several potential races may hide in obscure code and not be verified in actual executions).

2.4 Summary of data race detectors

Overall, we have categorized the data race detectors and reviewed some major detectors. Many other race detectors [10, 34, 44, 45, 63] have contributed to this direction too. However, some data race detectors suffer from false positives (e.g., RELAY, Eraser, etc.) due to imprecise static information, or limited synchronization information. For instance, many ad-hoc synchronizations [64–66] are not considered by detectors so that false positives are reported. Finally, we expect to see significant progress in the area of race detection as concurrent programs are increasingly popular.

3 Atomicity violation detectors

Atomicity violations are one of the most common and important concurrency bugs as a number of serializability criteria have been proposed for concurrent programs. Some detectors are specifically proposed to detect atomicity violations [12, 14, 16, 22, 67–74]. Some detectors can detect not only atomicity violations, but also other types of bugs (e.g., order violations and data races), and survive atomicity violations [15, 30, 75–78]. We review the atomicity violation detectors according to the techniques or goals. For instance, AVIO [69] is designed based on invariants, and PECAN [17] is designed based on interleaving patterns, and ICfinder [68] is designed to detect library-related atomicity violations.

3.1 Invariant-based detectors

AVIO [69]. By the observation of access interleaving invariants, Lu et al. propose AVIO which is a good indication of programmers' assumptions about the atomicity of certain code regions. By automatically extracting access interleaving invariants through a set of correct executions and detecting violations of these invariants at run time, AVIO can effectively detect a variety of atomicity violations.

DefUse [20]. DefUse is also an invariant-based bug detector which can detect not only concurrency bugs (including atomicity violations and order violations) but also memory and semantic bugs. Since many bugs occur as violations to programmers' data flow intentions, DefUse uses three different types of definition-use invariants which commonly exist in both sequential and concurrent programs. DefUse can automatically extract such invariants from programs, which are used to detect bugs. However, due to instrumentation, the runtime overhead imposed by DefUse is a challenge.

3.2 Multi-variable atomicity violation detectors

Some atomicity violations consist of more than two variables and cannot be detected by most atomicity violation detectors (e.g., AVIO). Lu et al. propose MUVI [67] to detect multi-variable atomicity violations. In concurrent programs, some correlated variables are not updated in a consistent way and correlated accesses are not protected in the same atomic sections so that detecting such violations is difficult. MUVI combines source code analysis and data-mining techniques to automatically infer variable access correlations and detect multi-variable atomicity violations.

3.3 Pattern-based detectors

Ctrigger [12]. To reduce the interleaving space, Ctrigger focuses on four interleaving patterns which are inherently correlated to atomicity violations. Ctrigger first uses trace analysis to systematically identify feasible unserializable interleavings with low occurrence probability. Then, it uses minimum execution perturbation to exercise low-probability interleavings and expose atomicity violations. Due to ranking mechanism, Ctrigger can expose the deeply hidden atomicity violations more efficiently.

AssetFuzzer [16]. Atomic-set serializability is another criterion that assumes a consistency property exists between memory accesses [16,22,72]. AssetFuzzer is an active randomized detector which can detect atomic-set serializability violations. It works in two phases. First, it uses a hybrid analysis technique (lockset and happens-before relation) to infer potential violations from a relaxed partial order execution trace. Then, it controls thread schedule to explore only these potentially violating interleavings. Due to violation inference and thread manipulation, AssetFuzzer can detect real atomicity violations effectively. However, similar to Ctrigger, being dynamic in nature, AssetFuzzer may miss atomicity violations which cannot be exposed with the given test inputs due to limited execution paths. To analyze more execution paths, PathExpander [79] attempts to increase the code path coverage with no programmer involvement, which may be used in Ctrigger and AssetFuzzer.

PECAN [17]. In addition to considering atomicity violation pattern, Huang et al. consider general interleaving patterns which include data race pattern and atomicity violation pattern, and propose a persuasive bug prediction detector called PECAN. PECAN can detect not only atomicity violations but also data races and atomic-set violations. It takes the bytecode of an arbitrary Java program and produces two versions: the record version and the replay version. The record version is executed for trace collection, and the replay version is used to create a concrete execution which can expose the predicted interleavings. It executes the record version to collect trace, analyzes and predicts potential interleavings. To find true concurrency bugs, it controls thread schedule to enforce a deterministic execution order of all the events and to create one potential interleaving, checking whether an exception occurs. The interleaving patterns defined in Falcon [18] and Unicorn [19] are similar to PECAN, which can also detect multiple types of concurrency bugs.

Maple [15]. To expose untested interleavings as much as possible, Yu et al. propose a new interleaving coverage-driven testing tool called Maple. Maple memorizes tested interleavings and actively seeks to expose untested interleavings for a given test input to increase interleaving coverage. It also defines a set of interleaving patterns which can be used to detect multiple types of concurrency bugs (e.g., atomicity violations and data races). Maple works in two phases. First, it uses an online technique to predict untested interleavings that can potentially be exposed for a given test input. Second, the predicted untested interleavings are exposed by controlling thread schedule. Maple avoids testing the same interleavings across different test inputs, improving the efficiency of concurrency bug detection. However, due to a large number of dynamic instrumentations, Maple imposes high runtime overhead (e.g., 45.4x overhead when it is applied to pbzip2).

3.4 Library-related atomicity violation detectors

In object-oriented concurrent programs, the libraries provide a convenient way for programmers to write parallel code. However, programmers are easy to misuse parallel libraries [80]. For instance, the library may implement its own atomic APIs correctly, and client code may incorrectly use the library APIs that cannot keep atomic. Liu et al. propose an automatic approach called ICfinder to detect such atomicity violations [68]. ICfinder works in four phases. First, it infers atomic sets of related fields in a given program. Second, it finds libraries that encapsulate them and the clients that use the libraries elsewhere in the program. Third, it infers atomicity requirements of these uses. Finally, it attempts to exhibit executions that violate these atomicity requirements. ICfinder helps programmers to use library APIs to keep atomic correctly.

3.5 Preventing atomicity violations

Several bug detectors have tried to use hardware to improve performance. Kivati uses hardware watchpoints to efficiently detect and prevent atomicity violations [75]. Kivati works in two phases. First, it combines watchpoints with a simple static analysis technique that annotates regions of codes that are likely required to be executed atomically. Second, it uses watchpoints to monitor these regions for accesses that may lead to an atomicity violation. Unlike previous detectors, Kivati can also prevent atomicity violations. It dynamically reorders the accesses when an atomicity violation is detected. Due to the usage of hardware, the average runtime overhead imposed by Kivati is only 19%, which makes it practical to deploy on software in production environments. However, since Kivati uses imprecise static analysis technique to annotate atomic region, several atomicity violations may be missed.

3.6 Sampling-based detectors

Cooperative bug isolation (CBI) project aims to automatically detect bugs in production-run software with low runtime overhead [77, 78, 81]. It statically instruments a program at particular program points. During program execution, it gathers feedback. Then, it statically analyzes the aggregated feedback data to identify program misbehaviors which are correlated with bugs. CBI imposes low runtime overhead due to its sampling strategy. However, CBI only focuses on one thread at a time, which makes CBI cannot detect concurrency bugs. Fortunately, cooperative crug isolation (CCI) enhances CBI. It is a low-overhead dynamic detector for detecting bugs in concurrent programs [30]. CCI analyzes specific thread interleavings at runtime, and uses statistical models to identify strong failure predictors among these. Concurrency bugs such as atomicity violations can be detected by CCI. To keep runtime overhead low, CCI uses variant random sampling strategies that suit different types of predicates, making it practical for use in production environments. However, due to sampling strategies, CCI may miss several concurrency bugs.

3.7 Summary of atomicity violation detectors

Overall, we have reviewed the atomicity violation detectors based on the techniques they use or the goals they target. Additionally, many other atomicity violation detectors like PENELOPE [82], and AtomFuzzer [14] can detect atomicity violations effectively too. Similar to Kivati, Atom-aid [76] can not only detect atomicity violations, but also prevent atomicity violations. As atomicity property is increasingly popular in parallel programming, we expect to see significant progress in the area of detecting atomicity violations.

4 Order violation detectors

An order violation occurs when the desired order between two memory accesses is flipped. According to the study in [28], 32% (24 out of 74) of the examined non-deadlock concurrency bugs are order violations. There are no specific detectors which only detect order violations. Therefore, we review some concurrency bug detectors that can detect order violations [15, 17–20, 30, 83–87] (they also can detect other types of concurrency bugs). If the detectors have been reviewed above, we briefly describe them.

4.1 General detectors

As we described above, Maple is an interleaving coverage-driven testing tool [15]. It defines a set of interleaving patterns, including order violation pattern. Therefore, Maple can detect order violations successfully. Other detectors like Falcon [18], Unicorn [19] and PECAN [17] which define general interleaving patterns can also detect order violations. Since general detectors define multiple types of interleaving patterns, they may report repeated results. Griffin [88] is proposed to optimize the results of these detectors. It groups the memory access patterns which are responsible for the same concurrency bug, and outputs the grouped patterns. By Griffin, programmers can understand the root of concurrency bug more easily.

4.2 Effect-oriented detectors

ConMem [84]. Unlike many concurrency bug detectors which focus on bugs caused by specific interleavings (e.g., atomicity violations), ConMem targets concurrency bugs which can lead to severe effects. ConMem is designed to predicatively catch concurrency-memory errors and to report fatal interleavings. It discovers buggy interleavings that lead to two types of common order violations: dangling resources and uninitialized reads. ConMem can detect concurrency bugs in two steps. First, it detects the basic ingredients of concurrency-memory errors which are mostly insensitive to interleavings. Second, it checks the timing condition (e.g., deallocating a memory object before another thread accesses it) among the basic ingredients. Since ConMem does not focus on specific type of concurrency bug, it can detect many types of bugs, including order violations. However, ConMem can only detect the order violations which would lead to program crashes.

ConSeq [85]. Fortunately, Zhang et al. enhance ConMem. They propose ConSeq which is an effect-oriented backward-analysis concurrency bug detector [85]. Unlike ConMem, which focuses on program crash, ConSeq focuses on five types of severe effects: infinite loop, assertion violations, memory errors, incorrect outputs, and error messages. ConSeq combines static analysis and dynamic analysis. It uses static analysis to identify potential error sites (e.g., assert function). It then uses static slicing to identify critical-read instructions that are likely to impact potential error sites through data/control dependence. After static analysis, ConSeq monitors several correct executions to identify potential interleavings. Finally, it exercises potential interleavings to find real concurrency bugs. Similar to ConMem, ConSeq does not focus on specific interleavings, it can successfully detect order violations. Since ConSeq is a backward, effect-oriented concurrency bug detector, it can complement many traditional concurrency bug detectors.

4.3 Summary of order violation detectors

Overall, we have reviewed some concurrency bug detectors which can detect order violations effectively. Additionally, other concurrency bug detectors can also detect order violations, such as CCI [30] and PCT [89]. We expect to see some order violation detectors which are designed accurately and efficiently.

5 Deadlock detectors

Concurrent programs always rely on locks to avoid race conditions. Unfortunately, programs with locks may deadlock when threads attempt to acquire locks held by other threads. The concurrency bug detectors which we reviewed above focus on non-deadlock concurrency bugs. In this section, we review some deadlock detectors which use dynamic analysis or static analysis [6, 29, 90–99].

5.1 Dynamic detectors

DeadlockFuzzer [29]. Joshi et al. propose DeadlockFuzzer which can detect deadlocks without false positives. DeadlockFuzzer combines an imprecise dynamic analysis technique with a randomized thread scheduler which can create real deadlocks with high probability. DeadlockFuzzer first uses an informative and simple variant of Goodlock algorithm called iGoodlock to discover potential deadlock cycles in a concurrent program. It then executes the program with a random scheduler to create a real deadlock corresponding to a cycle reported in the previous phase. The second phase prevents DeadlockFuzzer from reporting any false positives.

CheckMate [90]. CheckMate is an effective dynamic detector that can detect a broad class of deadlocks, including resource deadlocks, communication deadlocks, and deadlocks involving both locks and condition variables. CheckMate first observes a concurrent program execution and generates a trace that only records operations (e.g., lock acquire and release, wait and notify, thread start and join) observed during the execution that are deemed relevant to detecting deadlocks. Then, it uses a model checker to explore all possible interleavings of the trace and check if any of them can trigger a deadlock. The major advantage of CheckMate is that it discards most of the program logic which usually causes state-space explosion in model checking. However, being dynamic in nature, CheckMate is not complete because the trace is constructed by observing a single execution.

WOLF [97]. In order to determine whether the deadlock is feasible in a real execution, many dynamic analysis techniques require manual effort to reason about deadlocks. WOLF is a dynamic detector that can intelligently leverage execution traces, and replay the program by making the execution following a schedule derived based on the observed trace. Without manual effort, WOLF can automatically verify the feasibility of each deadlock through checking whether the reply causes the execution to deadlock.

5.2 Static detectors

Additionally, a Java extension called AJ uses locks to implement synchronization. Marino et al. present an algorithm for detecting possible deadlocks in AJ programs through ordering the locks associated with atomic sets [91]. They extend a type-based static analysis technique to handle recursive data structures by considering programmer-supplied lock ordering annotations. The algorithm relies on two properties of AJ: (1) all locks are associated with atomic sets, and (2) the memory locations associated with different atomic sets will be disjoint unless they are explicitly merged by the programmer. The algorithm computes a partial order on atomic sets. A program is deadlock-free if such an order can be found. Due to static analysis, this algorithm reports few false negatives. However, because AJ is just a research language without real users, the applicability of this approach may be limited.

5.3 Summary of deadlock detectors

Overall, we have reviewed several recent and major deadlock detectors. Additionally, many other deadlock detectors such as Dreadlocks [92] and Dimmunix [93] can also detect deadlocks effectively. We expect to see great progress in the area of deadlock detection.

6 Other techniques for concurrency bug detection

We have reviewed the most recent and important concurrency bug detectors based on the types of bugs. These detectors have made great progress in the area of concurrency bug detection. However, one concurrency bug may be triggered by a specific interleaving which is hidden in huge interleaving space. To alleviate the challenge of huge interleaving space, schedule bounding techniques and interleaving optimizing techniques are proposed. We discuss how these techniques work in concurrency bug detection.

Since dynamic analysis techniques cannot explore all the execution paths in actual executions, dynamic detectors are easy to produce false negatives. To face this problem, researchers use symbolic execution to expand execution paths. Therefore, we discuss several path expanding techniques that use symbolic execution to detect more concurrency bugs.

To detect harmful concurrency bugs, some dynamic detectors need to verify potential concurrency bugs in actual executions, checking whether the potential concurrency bugs can lead to program failure. Some dynamic detectors use deterministic replay techniques to control thread schedule to trigger concurrency bugs [17]. Moreover, the deterministic replay techniques are important for debugging concurrent program. Therefore, we discuss some deterministic replay techniques.

6.1 Schedule bounding techniques

The systematic techniques repeatedly execute a concurrent program, controlling thread schedule so that a different schedule is explored during each execution [100–106]. These techniques do not stop until all schedules have been explored, or until a time or schedule limit is reached. Therefore, they are inefficient because the number of schedules increases exponentially with execution steps.

To reduce the number of schedules, schedule bounding techniques have been proposed. These techniques are effective in concurrency bug detection. They can be classified into two parts: preemption bounding techniques and delay bounding techniques. Preemption bounding techniques bound the number of preemptive context switches [107], and delay bounding techniques bound the number of times a schedule can deviate from the scheduling decisions of a given deterministic scheduler [104]. Most of the dynamic detectors we have reviewed above use delay bounding techniques. Previous work has shown that delay bounding techniques improve on preemption bounding techniques in most cases [104, 108].

6.2 Interleaving optimizing techniques

Exposing a concurrency bug requires two conditions: right set of inputs and right thread interleaving. However, a concurrent program can take many different inputs, and follow many different interleavings while executing each input. Therefore, the interleaving space is too massive to be explored exhaustively. The systematic techniques [100–106] try to explore all the interleavings, which is inefficient. To avoid analyzing repeated interleavings, Bergan et al. constrain multithreaded execution to a small set of input-covering interleavings [109]. This approach uses a small set of interleavings instead of the huge interleaving space. Therefore, the approach can detect concurrency bugs by thoroughly testing each interleaving in this small set, improving the efficiency.

Additionally, Deng et al. observe that existing concurrency bug detectors always analyze the same interleavings under different inputs if a set of inputs is given to test concurrent programs [110]. To improve the efficiency of concurrency bug detection, they propose a new metric called concurrent function pair (CFP) to guide multi-input concurrency bug detection, which can avoid analyzing repeated interleavings across inputs [111]. However, if the inputs are randomly selected, unpredictable false negatives may be incurred. Maple [15] also speeds up bug verification by exercising each potential interleaving only once across different inputs. However, Maple incurs high runtime overhead.

To improve the effectiveness and efficiency of concurrency bug detection, inputs and interleavings are both important. Currently, to automatically find many inputs that may trigger concurrency bugs is a challenge. Many existing dynamic bug detectors require special inputs constructed by programmers. In future, this burden would be alleviated by automatic generation.

6.3 Path expanding techniques

Table 1 shows that most of the concurrency bug detectors use dynamic analysis techniques. However, dynamic analysis techniques always suffer from limited execution paths. Fortunately, symbolic execution is an effective technique for generating high-coverage test suites and for finding deep errors in complex software [42, 79, 100, 120–124]. It can explore more execution paths in a given amount of time. PathExpander [79] uses symbolic execution to increase path coverage so that more concurrency bugs are detected. Portend [42] uses symbolic execution to improve the accuracy of data race classification. Farzan et al. systematically test concurrent programs by con2colic testing [100] which is based on concrete and symbolic executions of a program. Currently, most of the dynamic concurrency bug detectors have not used symbolic execution to alleviate the problem of false negatives.

6.4 Deterministic replay techniques

Deterministic replay [125–133] is an important technique which can make the concurrent program execute the same thread interleaving in multiple executions. PECAN [17] is a general detector which uses deterministic replay to dynamically trigger concurrency bugs under specific thread interleavings. Moreover, some detectors [12–14] use deterministic replay to manifest concurrency bugs again and again, which can help programmers to understand and debug the bugs. LEAN [126] and LEAP [127] are two techniques of deterministic replay which can substantially assist debugging concurrent programs by making the program execution reproducible. ODR [125] is a replay system which does not need to generate a replica of the original execution, producing any execution that exhibits the same outputs as the original execution. Consequently, deterministic replay techniques can be used in concurrency bug detectors to trigger bugs or reproduce bugs.

7 Discussion

In this section, we statistically categorize the concurrency bug detectors, getting some interesting findings. We then statistically analyze the limitations of previous detectors. The goal of statistical analysis is computing the percentage of each type, approach or limitation, helping researchers to understand the previous work clearly. For instance, we select many concurrency bug detectors and statistically analyze the approaches used in these detectors, getting some useful findings to help researchers to do future work. If researchers are going to design new dynamic approaches to detect concurrency bugs, they can conveniently refer to the previous detectors from the results of statistical analysis. Currently, the statistical analysis is simple, since it only needs to compute the percentage. For instance, we compute the percentage of detectors that can detect atomicity violations, the percentage of detectors that use dynamic approaches, and the percentage of detectors that suffer from false negatives.

7.1 Statistical categorization of concurrency bug detectors

Previous surveys [3, 4] categorize the concurrency bug detectors based on the methodologies. In Table 1, we categorize the detectors into a two-dimension space (● means the detector can detect this type of concurrency bug or use this approach). For instance, Ctrigger is not only an atomicity violation detector, but also a dynamic detector. We do not categorize all the existing concurrency bug detectors, but the most up-to-date and well-known detectors are categorized. As we can see in Table 1, 55 existing detectors are selected and categorized. Then, we get the following interesting findings:

1) Most of the existing concurrency bug detectors (86%) focus on data races and atomicity violations. This high percentage indicates that data races and atomicity violations may occur more

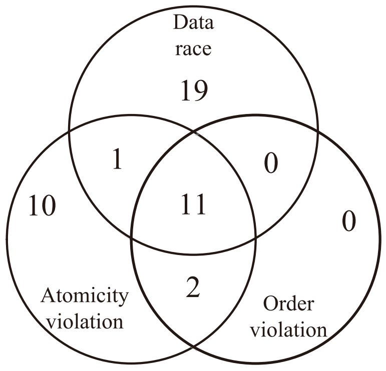


Figure 2 Venn diagram showing number of different types of non-deadlock detectors.

frequently in concurrent programs. Therefore, detecting these two types of concurrency bugs is more important. On the other hand, order violations and deadlocks require more researchers to focus on.

2) Dynamic analysis techniques are popular (74%) in concurrency bug detection. The detection results reported by static detectors always have a large number of false positives so that manual inspection is required to separate the true bugs, which can avoid debugging false positives. Although dynamic detectors may produce false negatives, they report few false positives so that programmers can fix the concurrency bugs more quickly.

3) Only 14% of the reviewed detectors use static analysis to detect concurrency bugs. This low percentage indicates that static analysis may be limited in concurrency bug detection, since the results reported by several static detectors include a large number of false positives that may consume a lot of unnecessary effort. On the other hand, the accuracy of static detectors needs to be improved. Additionally, the static analysis techniques can complement dynamic analysis techniques, which makes researchers propose some hybrid detectors.

The Venn diagram in Figure 2 gives a concise summary of the non-deadlock concurrency bug detectors (the detectors selected in Table 1). For instance, there are 10 detectors which are designed for atomicity violation specifically, 2 detectors can detect both atomicity violations and order violations, and 11 detectors can detect all the three types of concurrency bugs.

7.2 Statistically analyzing limitations of concurrency bug detectors

We statistically analyze the limitations based on four factors: false negatives, false positives, overhead, and manual effort. The main reasons of these four limitations are described in Section 1. It is difficult to apply different detectors to the same concurrent programs, since different detectors analyze different programming languages, for instance, RELAY [5] analyzes standard C programs and PECAN [17] analyzes Java programs.

We have installed the detectors which are publicly available and apply them to some real world programs, finding some advantages and disadvantages of them. For instance, RELAY reports all the data races but reports a large number of false positives, Maple can effectively detect concurrency bugs but imposes high overhead, and PECAN reports real concurrency bugs but consumes large memory when instrumentation. We cannot install all the reviewed detectors to evaluate them on real world programs since most of them are not publicly available. Consequently, we analyze the limitations of concurrency bug detectors based on the experiments that have been done by researchers if the detectors are not publicly available, and some analysis results are based on the experiments that we have done (if the detectors are publicly available).

We statistically summarize the limitations of many detectors (33 most up-to-date and well-known detectors are selected). The developers can directly know the main limitations of each detector and use multiple detectors to complement each other. Table 2 shows the results (● means the detector suffers from this limitation). Columns 2 and 3 show that whether the detectors report false negatives and false positives, respectively. Column 4 shows whether the detectors impose high overhead (we assume high

Table 1 Categorization of concurrency bug detectors

	Types of detected concurrency bugs				Methodologies		
Detector	Data race	Atomicity vio.	Order vio.	Deadlock	Static	Dynamic	Hybrid
RELAY [5]	●				●		
KISS [7]	●				●		
EPAJ [36]	●				●		
RacerX [6]	●			●	●		
Locksmith [10]	●				●		
CoBE [39]	●				●		
IteRace [112]	●				●		
RaceFuzzer [13]	●					●	
Portend [42]	●					●	
Parallel race detector [43]	●					●	
IFRit [44]	●					●	
FastTrack [45]	●					●	
RD2 [113]	●					●	
Maple [15]	●	●	●			●	
PECAN [17]	●	●				●	
Falcon [18]	●	●	●			●	
Unicorn [19]	●	●	●			●	
Eraser [21]	●					●	
DataCollider [34]	●					●	
LiteRace [46]	●					●	
CCI [30]	●	●	●			●	
ConMem [84]	●	●	●			●	
Con2colic [100]	●	●		●		●	
jCUTE [114]	●	●		●		●	
CHESS [101]	●	●	●	●		●	
IMUnit [115]	●	●	●	●		●	
RaceTrack [116]	●					●	
FUSION [102]	●	●	●	●		●	
Inspect [103]	●	●	●	●		●	
PACER [117]	●					●	
RaceMob [25]	●						●
ConSeq [85]	●	●	●				●
MultiRace [118]	●						●
ColFinder [26]	●	●	●				●
Ctrigger [12]		●				●	
DoubleChecker [119]		●				●	
AtomFuzzer [14]		●				●	
AssetFuzzer [16]		●				●	
MUVI [67]		●				●	
AVIO [69]		●				●	
Kivati [75]		●				●	
Atom-aid [76]		●				●	
PENELOPE [82]		●				●	
DefUse [20]		●	●			●	
PCT [89]		●	●	●		●	
ICfinder [68]		●					●
HAVE [73]		●					●
Daniel's algorithm [91]				●	●		
DeadlockFuzzer [29]				●		●	
Dreadlocks [92]				●		●	
CheckMate [90]				●		●	
Dimmunix [93]				●		●	
MagicFuzzer [94]				●		●	
Pulse [95]				●		●	
WOLF [97]				●		●	

Table 2 Limitations of concurrency bug detectors

Detector	False negative	False positive	Overhead	Manual effort
EPAJ	●	●		●
RELAY		●		●
KISS	●			●
Locksmith		●		●
IteRace		●		●
RaceFuzzer	●		●	
Portend	●		●	
FastTrack	●	●	●	
RD2	●		●	
Eraser	●	●	●	●
Maple	●		●	
PECAN	●		●	
ConMem	●		●	
CCI	●			●
Ctrigger	●		●	
AssetFuzzer	●		●	
AVIO	●	●		●
MUVI	●	●		●
PENELOPE	●		●	
AtomFuzzer	●		●	
Kivati	●	●		●
DefUse	●	●	●	
Falcon	●		●	
PCT	●	●	●	●
ColFinder	●			
ConSeq	●		●	●
CheckMate	●	●	●	
DeadlockFuzzer	●		●	
Dimmunix	●			
RaceMob	●			
ICfinder	●		●	
Unicorn	●	●	●	
RaceTrack	●		●	
Rate	91%	39%	64%	36%

overhead means larger than 100%). Column 5 shows whether the results reported by detectors need manual inspection or manual annotation to help detecting. From Table 2, we get the following findings:

1) Most of the existing concurrency bug detectors (91%) suffer from reporting false negatives. Due to limited execution paths, dynamic detectors often miss the bugs hidden in obscure code. Some static detectors may report false negatives due to the simplification of detecting process.

2) Many detectors (64%) suffer from imposing high runtime overhead. Most of the detectors use binary instrumentation to predict and verify potential interleavings, imposing high runtime overhead. However, some dynamic detectors can achieve low overhead with hardware support, or some special strategies.

3) Only a few concurrency bug detectors (36%) are not completely automatic. Although many detectors can automatically analyze concurrent programs and detect concurrency bugs, a few detectors are not completely automatic due to manual annotations or manual inspection of detection results.

Table 2 shows the limitations of each concurrency bug detector. However, some detectors suffer from some limitations more severely. It is unfair to give a same result for different detectors. For instance, the

Table 3 Quantitative analysis on overhead

Detector	Average overhead	Range of overhead
PECAN	110%	0%–784%
RaceTrack	138%	30%–198%
RaceFuzzer	187%	1%–1947%
DeadlockFuzzer	200%	26%–498%
RD2	201%	60%–2658%
ConMem	262%	3%–1556%
AssetFuzzer	272%	45%–1060%
FastTrack	335%	100%–4430%
AtomFuzzer	376%	23%–1601%
Portend	381%	110%–4990%
DefUse	430%	103%–2026%
Unicorn	723%	20%–3540%
ConSeq	798%	21%–3846%
CheckMate	988%	112%–1733%
Falcon	1012%	10%–6040%
Eraser	2000%	1000%–3000%
Maple	3709%	250%–18330%
PENELOPE	10306%	1%–116215%

average overhead imposed by DefUse [20] is nearly 5X, and the average overhead imposed by Maple [15] is nearly 40X. Therefore, we try to do some quantitative analyses on detectors' limitations.

As we can see from Table 2, most of the detectors suffer from false negatives. However, it is hard to quantitatively evaluate the false negatives of each detector. There are no detectors that can detect all the true concurrency bugs (without any false positives). Researchers propose a concurrency bug detector and detect some concurrency bugs in one program. Since researchers do not know all the concurrency bugs in one program, they cannot evaluate the false negatives of their proposed detector. Based on the design of their detector, they discuss the scenarios that cause the detector to report false negatives. Also, we identify the false negatives of each detector based on its design.

Some concurrency bug detectors may report several false positives. The researchers discuss the reasons for reporting false positives in their papers. For instance, RELAY [5] may report false positives due to six scenarios, and the authors design some filters to prune false positives as many as possible. However, they do not evaluate the false positives in detail. To determine whether a reported concurrency bug is false positive, researchers need to check it manually. If the number of reported concurrency bugs is huge, it is hard to check all the reported concurrency bugs manually. Therefore, we identify the false positives of each detector based on its design and the researchers' analysis.

Additionally, manual effort is one of the limitations, which is discussed by many researchers in their papers. It is also hard to do some quantitative analyses. For instance, EPAJ [36] requires manual annotations, which is hard to evaluate quantitatively. We also identify the manual effort of each detector based on its design.

To evaluate the efficiency of concurrency bug detector, runtime overhead is an important factor. The 4th column of Table 2 shows that 21 detectors suffer from high runtime overhead. To give a quantitative analysis on those 21 detectors' runtime overhead, we evaluate the average overhead and the range of overhead. It is hard to implement those 21 detectors and apply them to concurrent programs. We directly use the results from their papers. Among those 21 detectors, three detectors' overhead is not evaluated (Ctrigger, PCT, ICfinder). Therefore, we show the overhead of 18 detectors in Table 3. According to the average overhead of each detector in Table 3, researchers can understand the overhead of those detectors in detail.

8 Future research

In this section, we discuss the future research in concurrency bug detection. Based on the reviewed detectors and statistical analysis of many detectors, we conclude several future research directions, including accuracy, performance, applicability, and integrality.

8.1 Accuracy

Most of the static detectors can analyze the whole source code and produce few false negatives. However, being imprecise in nature, static detectors always report a large number of false positives. If programmers want to debug and fix concurrency bugs, they may consume much time to understand the false positives. Consequently, it is urgent for researchers to design some new algorithms to make the detectors produce fewer false positives.

Many dynamic detectors trigger concurrency bugs in actual executions so that few false positives are reported. However, being dynamic in nature, some obscure code may not be executed in multiple actual executions, which makes some concurrency bugs missed. Although previous detectors have tried to use symbolic execution to expand execution paths, the capability is limited since several deeply hidden bugs are still hard to be detected. Moreover, symbolic execution consumes much time to expand execution paths in large programs. As we have reviewed above, dynamic detectors are popular. Accordingly, researchers may use con2colic testing techniques to reduce the false negatives produced by dynamic detectors. To handle the problem of inefficiency, researchers may make a tradeoff between accuracy and efficiency.

Additionally, the accuracy may be affected by compiler. If the detector analyzes source code to detect bugs, several false negatives may be produced since the compiled code may have bugs due to the optimization by compiler. On the other hand, if the detector analyzes compiled code to detect bugs, several unrelated bugs may be produced since these bugs may be caused by libraries. It is hard for programmers to fix such bugs which are not in source code. Consequently, researchers may design some new detectors to detect real bugs in source code and show bugs that are not in source code.

8.2 Performance

As we have reviewed above, some dynamic detectors impose high runtime overhead, which affects the efficiency of concurrency bug detection. The main reason is that they need to instrument concurrent program and collect execution information. The instrumented instructions make the program executed slower. To alleviate this challenge, researchers may use some strategies such as sampling strategy to reduce the number of instrumentation. Also, researchers may use virtual machine to monitor program execution, which can replace instrumentation.

Moreover, dynamic detectors always need to execute the instrumented program multiple times. In those executions, some are redundant, which may affect the efficiency of bug detection. Researchers may design some strategies to avoid redundant dynamic analysis.

Researchers also have tried to deploy the detectors in production environments, which indicates that low performance would greatly affect the usability of software. Accordingly, researchers may design specific hardware or new algorithms to reduce runtime overhead imposed by dynamic detectors.

Additionally, the interleaving space of large program is always huge. It is difficult to find specific interleavings which make the program produce undesired results. Fortunately, deterministic execution techniques can eliminate the non-determinism, making the threads executed deterministically. Researchers have proposed many deterministic techniques to improve software reliability. These techniques also can be used in concurrency bug detection. Since concurrency bugs always hide in a small quantity of code, to improve performance, it is unnecessary to analyze all the source code. Researchers may design some predictive analysis to determine which codes are safe (even multiple threads may execute these codes). Then, deterministic execution techniques can make the predicted codes executed deterministically so that huge interleaving space may be reduced significantly, making fewer interleavings analyzed.

8.3 Applicability

We have reviewed several concurrency bug detectors which need programmers to add annotations to help detectors. The manual annotations may affect the applicability of detectors if the concurrent programs which need to be analyzed are large and complex. Accordingly, researchers need to make the detectors completely automatic.

Some detectors produce a large number of false positives, causing programmers to consume much time to understand the wrong detection results. The real bugs are hidden and may not be fixed if time is limited. Such detectors may ensure there is no false negatives, but they may be not applicable. Therefore, improving accuracy can also improve applicability. Researchers may first use imprecise detectors to predict many potential concurrency bugs. Then, they can use precise detectors to analyze potential concurrency bugs. This process of detection is similar to the reviewed hybrid detectors. However, hybrid detectors may affect the applicability due to complex usage. Researchers may improve the convenience to make the detectors more applicable.

Additionally, most of the concurrency bug detectors are designed to test C/C++ or Java programs. However, some other programming languages such as Objective-C and Python are also popular. Researchers may design some new detectors to test more types of programming languages.

8.4 Integrality

The goal of concurrency bug detection is to improve software reliability. However, it is just a start. After detecting concurrency bugs, programmers need to fix them. Since the concurrency bugs are complex, fixing them is time-consuming and error-prone [134–136]. It is not integrated if concurrency bugs are not fixed during in-house testing.

Fortunately, researchers have proposed many tools to fix concurrency bugs automatically [137–140]. Most of them rely on concurrency bug detectors to fix bugs. The fixing strategies are simple. For instance, AFix [137] inserts lock and unlock operations to protect atomic regions, avoiding atomicity violations. Through inserting synchronization operations, many simple concurrency bugs are fixed effectively. However, some complex bugs that involve multiple shared variables are hard to be fixed effectively. Although the tools can fix concurrency bugs automatically, the performance of program may be reduced since these tools insert many synchronization operations. Consequently, fixing concurrency bugs is important. To make the testing integrated, researchers may design some new tools to effectively fix bugs which are detected by concurrency bug detectors.

9 Conclusion

We have reviewed the most up-to-date and important concurrency bug detectors. We categorize the concurrency bug detectors based on the types of concurrency bugs. Researchers can know what types of concurrency bugs are studied by previous detectors and find some implications to design new algorithms which can complement previous detectors. We also briefly discuss some techniques which may help concurrency bug detection, including schedule bounding techniques, interleaving optimizing techniques, path expanding techniques, and deterministic reply techniques. These techniques are used to improve the effectiveness and efficiency of concurrency bug detection. Researchers can design new concurrency bug detectors based on these techniques.

We statistically analyze many concurrency bug detectors, including the types of bugs they target and methodologies they use. From statistical analysis, we get some interesting findings, for instance, dynamic analysis techniques are popular in concurrency bug detection. Based on these findings, we discuss some future research directions which may guide researchers in the area of concurrency bug detection.

Additionally, we statistically analyze the limitations of different concurrency bug detectors. We focus on four types of limitations (false negative, false positive, runtime overhead, and manual effort) which are evaluated in previous detectors. From statistical analysis, we also get some interesting findings. For instance, most of the existing detectors suffer from false negatives. These findings may guide researchers to do some future work to alleviate the limitations of concurrency bug detection. Overall, we expect to see significant advancement in the area of concurrency bug detection.

Acknowledgements This work was supported by National High Technology Research and Development Program of China (863 Program) (Grant Nos. 2012AA01A301, 2012AA010901) and Program for New Century Excellent Talents in University and National Natural Science Foundation of China (Grant Nos. 61402486, 61402492, 61272142). Moreover, it is a part of Innovation Fund Sponsor Project of Excellent Postgraduate Student (Grant No. B130608).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Leveson N G, Turner C S. An investigation of the therac-25 accidents. Computer, 1993, 26: 18–41
2 Godefroid P, Nagappan N. Concurrency at Microsoft an Exploratory Survey. Technical Report, Microsoft Research, MSR-TR-2008-75. 2008
3 Abdelqawy D, Kamel A, Omara F. A survey on testing concurrent and multi-threaded applications tools and methodologies. In: Proceedings of the International Conference on Informatics and Applications, Kuala Terengganu, 2012. 458–470
4 Raza A. A review of race detection mechanisms. In: Proceedings of the 1st International Conference on Computer Science Theory and Applications. Berlin: Springer, 2006. 534–543
5 Voung J W, Jhala R, Lerner S. RELAY: static race detection on millions of lines of code. In: Proceedings of the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2007. 205–214
6 Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York: ACM, 2003. 237–252
7 Qadeer S, Wu D. KISS: keep it simple and sequential. In: Proceedings of the 25th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2004. 14–24
8 Flanagan C, Freund S N. Type-based race detection for Java. In: Proceedings of the 21st Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2000. 219–232
9 Grossman D. Type-safe multithreading in cyclone. In: Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation. New York: ACM, 2003. 13–25
10 Pratikakis P, Foster J S, Hicks M. Locksmith: practical static race detection for C. ACM Trans Program Lang Syst, 2011, 33: 1–55
11 Sterling N. WARLOCK-A static data race analysis tool. In: Proceedings of USENIx Winter, San Diego, 1993. 97–106
12 Park S, Lu S, Zhou Y Y. CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2009. 25–36
13 Sen K. Race directed random testing of concurrent programs. In: Proceedings of the 29th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2008. 11–21
14 Park C S, Sen K. Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2008. 135–145
15 Yu J, Narayanasamy S, Pereira C, et al. Maple: a coverage-driven testing tool for multithreaded programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2012. 485–502
16 Lai Z, Cheung S C, Chan W K. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 235–244
17 Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2011. 144–154
18 Park S, Vuduc R W, Harrold M J. Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York: ACM, 2010. 245–254
19 Park S, Vuduc R, Harrold M J. A unified approach for localizing non-deadlock concurrency bugs. In: Proceedings of IEEE 5th International Conference on Software Testing, Verification and Validation, Montreal, 2012. 51–60
20 Shi Y, Park S, Yin Z, et al. Do I use the wrong definition? DefUse: definition-use invariants for detecting concurrency and sequential bugs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2010. 160–174
21 Savage S, Burrows M, Nelson G, et al. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans Comput Syst, 1997, 15: 391–411
22 Hammer C, Dolby J, Vaziri M, et al. Dynamic detection of atomic-set-serializability violations. In: Proceedings of the 30th ACM/IEEE International Conference on Software Engineering. New York: ACM, 2008. 231–240

23 Brat G, Visser W. Combining static analysis and model checking for software analysis. In: Proceedings of the 16th Annual International Conference on Automated Software Engineering, San Diego, 2001. 262–269

24 Chen J, MacDonald S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. New York: ACM, 2008. 8

25 Kasikci B, Zamfir C, Candea G. RaceMob: crowdsourced data race detection. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. New York: ACM, 2013. 406–422

26 Wu Z D, Lu K, Wang X P, et al. Collaborative technique for concurrency bug detection. Int J Parall Program, 2015, 43: 260–285

27 Fiedor J, Krena B, Letko Z, et al. A Uniform Classification of Common Concurrency Errors. Technical Report, Brno University of Technology, FIT-TR-2010-03. 2010

28 Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2008. 329–339

29 Joshi P, Park C S, Sen K, et al. A randomized dynamic program analysis technique for detecting real deadlocks. In: Proceedings of the 30th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2009. 110–120

30 Jin G, Thakur A, Liblit B, et al. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2010. 241–255

31 Corp I. Parallel Inspector. http://software.intel.com/en-us/articles/intel-parallel-inspector. 2012

32 Sadowski C, Yi J. How developers use data race detection tools. In: Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools. New York: ACM, 2014. 43–51

33 Bessey A, Block K, Chelf B, et al. A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM, 2010, 53: 66–75

34 Erickson J, Musuvathi M, Burckhardt S, et al. Effective data-race detection for the kernel. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Berkeley, 2010. 1–16

35 Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2006. 308–319

36 Agarwal R, Sasturkar A, Wang L, et al. Optimized run-time race detection and atomicity checking using partial discovered types. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2005. 233–242

37 Flanagan C, Freund S. Detecting race conditions in large programs. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM, 2001. 90–96

38 Abadi M, Flanagan C, Freund S N. Types for safe locking: static race detection for Java. ACM Trans Program Lang Syst, 2006, 28: 207–255

39 Kahlon V, Sinha N, Kruus E, et al. Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2009. 13–22

40 Flanagan C, Freund S N. Type inference against races. In: Static Analysis. Berlin: Springer, 2004. 116–132

41 Kidd N, Lammich P, Touilli T, et al. A static technique for checking for multiple-variable data races. Softw Tools Tech Transfer, 2010

42 Kasikci B, Zamfir C, Candea G. Data races vs. data race bugs: telling the difference with portend. In: Proceedings of the 17th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2012. 185–198

43 Wester B, Devecsery D, Chen P M, et al. Parallelizing data race detection. In: Proceedings of the 18th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2013. 27–38

44 Effinger-Dean L, Lucia B, Ceze L, et al. IFRit: interference-free regions for dynamic data-race detection. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2012. 467–484

45 Flanagan C, Freund S N. FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 30th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2009. 121–133

46 Marino D, Musuvathi M, Narayanasamy S. LiteRace: effective sampling for lightweight data-race detection. In: Proceedings of the 30th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2009. 134–143

47 Callahan R, Choi J D. Hybrid dynamic data race detection. In: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2003. 167–178

48 Choi J D. Efficient and precise data race detection for multithreaded object-oriented programs. In: Proceedings of the 23rd Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2002. 258–269

49 Helmbold D P, McDowell C E. A Taxonomy of Race Detection Algorithms. Technical Report UCSC-CRL-94-35. 1994

50 Ronsse M, de Bosschere K. RecPlay: a fully integrated practical record/replay system. ACM Trans Comput Syst, 1999, 17: 133–152

51 Huang J, Meredith P O, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proceedings of the 35th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 337–348

52 Eslamimehr M, Palsberg J. Race directed scheduling of concurrent programs. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2014. 301–314

53 Wood B P, Ceze L, Grossman D. Low-level detection of language-level data races with LARD. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014. 671–686

54 Sen K, Rosu G, Agha G. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Formal Methods for Open Object-Based Distributed Systems. Berlin: Springer, 2005. 211–226

55 Yu T, Srisa-an W, Rothermel G. SimRT: an automated framework to support regression testing for data races. In: Proceedings of the ACM/IEEE 36th International Conference on Software Engineering. New York: ACM, 2014. 48–59

56 Wood B P, Ceze L, Grossman D. Low-level detection of language-level data races with LARD. In: Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014. 671–686

57 Serebryany K, Potapenko A, Iskhodzhanov T, et al. Dynamic race detection with LLVM compiler. In: Runtime Verification. Berlin: Springer, 2011. 110–114

58 Lu K, Wu Z, Wang X, et al. RaceChecker: efficient identification of harmful data races. In: Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turku, 2015. 78–85

59 Narayanasamy S, Tigani J, Edwards A, et al. Automatically classifying benign and harmful data races using replay analysis. In: Proceedings of the 28th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2007. 22–31

60 Maiya P, Kanade A, Majumdar R. Race detection for android applications. In: Proceedings of the 35th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 316–325

61 Hsiao C H, Yu J, Narayanasamy S, et al. Race detection for event-driven mobile applications. In: Proceedings of the 35th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 326–336

62 Mao X G, Lei Y, Dai Z Y, et al. Slice-based statistical fault localization. J Syst Softw, 2014, 89: 51–62

63 Park C S, Sen K, Iancu C. Scaling data race detection for partitioned global address space programs. In: Proceedings of the 27th International ACM Conference on Supercomputing. New York: ACM, 2013. 47–58

64 Jannesari A, Tichy W F. Identifying ad-hoc synchronization for enhanced race detection. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing, Atlanta, 2010. 1–10

65 Xiong W, Park S, Zhang J, et al. Ad hoc synchronization considered harmful. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Vancouver, 2010. 163–176

66 Tian C, Nagarajan V, Gupta R, et al. Dynamic recognition of synchronization operations for improved data race detection. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2008. 143–154

67 Lu S, Park S, Hu C, et al. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles. New York: ACM, 2007. 103–116

68 Liu P, Dolby J, Zhang C. Finding incorrect compositions of atomicity. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013. 158–168

69 Lu S, Tucek J, Qin F, et al. AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2006. 37–48

70 Muzahid A, Otsuki N, Torrellas J. AtomTracker: a comprehensive approach to atomic region inference and violation detection. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM, 2010. 287–297

71 Flanagan C, Freund S N. Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2004. 256–267

72 Kidd N, Reps T, Dolby J, et al. Static Detection of Atomic-Set-Serializability Violations. Technical Report #1623. University of Wisconsin-Madison, 2007

73 Chen Q, Wang L, Yang Z, et al. HAVE: detecting atomicity violations via integrated dynamic and static analysis. In: Fundamental Approaches to Software Engineering. Berlin: Springer, 2009. 425–439

74 Ye C, Cheung S, Chan W, et al. Detection and resolution of atomicity violation in service composition. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2007. 235–244

75 Chew L, Lie D. Kivati: fast detection and prevention of atomicity violations. In: Proceedings of the 5th European Conference on Computer Systems. New York: ACM, 2010. 307–320

76 Lucia B, Devietti J, Strauss K, et al. Atom-aid: detecting and surviving atomicity violations. In: Proceedings of the 35th Annual International Symposium on Computer Architecture, Beijing, 2008. 277–288

77 Liblit B, Naik M, Zheng A X, et al. Public deployment of cooperative bug isolation. In: Proceedings of the 2nd International Workshop on Remote Analysis and Measurement of Software Systems, 2004. 1: 57–62

78 Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. In: Proceedings of the 26th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 15–26

79 Lu S, Zhou P, Liu W, et al. PathExpander: architectural support for increasing the path coverage of dynamic bug detection. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM, 2006. 38–52

80 Okur S, Dig D. How do developers use parallel libraries? In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York: ACM, 2012. 54

81 Liblit B, Aiken A, Zheng A X, et al. Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2003. 141–154

82 Sorrentino F, Farzan A, Madhusudan P. PENELOPE: weaving threads to expose atomicity violations. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2010. 37–46

83 Lucia B, Ceze L. Finding concurrency bugs with context-aware communication graphs. In: Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture. New York: ACM, 2009. 553–563

84 Zhang W, Sun C, Lu S. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In: Proceedings of the 15th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2010. 179–192

85 Zhang W, Lim J, Olichandran R, et al. ConSeq: detecting concurrency bugs through sequential errors. In: Proceedings of the 16th Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2011. 251–264

86 Gao Q, Zhang W, Chen Z, et al. 2ndStrike: toward manifesting hidden concurrency typestate bugs. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2012. 239–250

87 Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. In: Proceedings of the 36th Annual International Symposium on Computer Architecture. New York: ACM, 2009. 325–336

88 Sangmin P, Mary J H, Richard V. Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2013. 134–144

89 Burckhardt S, Kothari P, Musuvathi M, et al. A randomized scheduler with probabilistic guarantees of finding bugs. ACM Sigplan Notices, 2010, 45: 167–178

90 Joshi P, Naik M, Sen K, et al. An effective dynamic analysis for detecting generalized deadlocks. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2010. 327–336

91 Marino D, Hammer C, Dolby J, et al. Detecting deadlock in programs with data-centric synchronization. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 322–331

92 Koskinen E, Herlihy M. Dreadlocks: efficient deadlock detection. In: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures. New York: ACM, 2008. 297–303

93 Jula H, Tralamazza D, Zamfir C, et al. Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 295–308

94 Cai Y, Chan W. MagicFuzzer: scalable deadlock detection for large-scale applications. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 606–616

95 Li T, Ellis C S, Lebeck A R, et al. Pulse: a dynamic deadlock detection mechanism using speculative execution. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. Berkeley: USENIX Association, 2005. 31–44

96 Naik M, Park C S, Sen K, et al. Effective static deadlock detection. In: Proceedings of the 31st International Conference on Software Engineering, Vancouver, 2009. 386–396

97 Samak M, Ramanathan M K. Trace driven dynamic deadlock detection and reproduction. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2014. 29–42

98 Nistor A, Luo Q, Pradel M, et al. Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 727–737

99 Cai Y, Wu S, Chan W K. ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In: Proceedings of the ACM/IEEE 36th International Conference on Software Engineering. New York: ACM, 2014. 491–502

100 Farzan A, Holzer A, Razavi N, et al. Con2colic testing. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013. 37–47

101 Musuvathi M, Qadeer S, Ball T, et al. Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. New York: ACM, 2008. 267–280

102 Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. In: Proceedings of the 33rd International Conference on Software Engineering, Honolulu, 2011. 221–230

103 Yang Y, Chen X, Gopalakrishnan G. Inspect: a Runtime Model Checker for Multithreaded C Programs. University of Utah, Technology Report UUCS-08-004. 2008

104 Emmi M, Qadeer S, Rakamari Z. Delay-bounded scheduling. In: Proceedings of the 38th Annual ACM SIGPLAN-

SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2011. 411–422

105 Godefroid P. Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 1997. 174–186

106 Sandeep B, Sorav B, Akash L. Variable and thread bounding for systematic testing of multithreaded programs. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2013. 145–155

107 Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2007. 446–455

108 Thomson P, Donaldson A F, Betts A. Concurrency testing using schedule bounding: an empirical study. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2014. 15–28

109 Bergan T, Ceze L, Grossman D. Input-covering schedules for multithreaded programs. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2013. 677–692

110 Deng D, Zhang W, Wang B, et al. Understanding the interleaving-space overlap across inputs and software versions. In: Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism. Berkeley: USENIX Association, 2012. 17

111 Deng D, Zhang W, Lu S. Efficient concurrency-bug detection across inputs. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2013. 785–802

112 Radoi C, Dig D. Practical static race detection for Java parallel loops. In: Proceedings of the International Symposium on Software Testing and Analysis. New York: ACM, 2013. 178–190

113 Dimitrov D, Raychev V, Vechev M, et al. Commutativity race detection. In: Proceedings of the 35th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 305–315

114 Sen K, Agha G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Hardware and Software, Verification and Testing. Berlin: Springer, 2007. 166–182

115 Jagannath V, Gligoric M, Jin D, et al. Improved multithreaded unit testing. In: Proceedings of the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2011. 223–233

116 Yu Y, Rodeheffer T, Chen W. Racetrack: efficient detection of data race conditions via adaptive tracking. ACM SIGOPS Oper Syst Rev, 2005, 39: 221–234

117 Bond M D, Coons K E, McKinley K S. PACER: proportional detection of data races. In: Proceedings of the 31st Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2010. 255–268

118 Pozniansky E, Schuster A. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. Concurr Comput Pract Exper, 2007, 19: 327–340

119 Biswas S, Huang J, Sengupta A, et al. DoubleChecker: efficient sound and precise atomicity checking. In: Proceedings of the 35th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2014. 28–39

120 Cadar C, Sen K. Symbolic execution for software testing: three decades later. Commun ACM, 2013, 56: 82–90

121 Cadar C, Dunbar D, Engler D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 209–224

122 Ciortea L, Zamfir C, Bucur S, et al. Cloud9: a software testing service. ACM SIGOPS Oper Syst Rev, 2010, 43: 5–10

123 Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death. ACM Trans Inf Syst Secur, 2008, 12: 322–335

124 Liu W W, Wang J, Chen H W, et al. Symbolic model checking APSL. Front Comput Sci China, 2009, 3: 130–141

125 Altekar G, Stoica I. ODR: output-deterministic replay for multicore debugging. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York: ACM, 2009. 193–206

126 Huang J, Zhang C. LEAN: simplifying concurrency bug reproduction via replay-supported execution reduction. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York: ACM, 2012. 451–466

127 Huang J, Liu P, Zhang C. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2010. 207–216

128 Veeraraghavan K, Lee D, Wester B, et al. DoublePlay: parallelizing sequential logging and replay. ACM Trans Comput Syst, 2012, 30: 3

129 Weiss Z, Harter T, Arpaci-Dusseau A C, et al. ROOT: replaying multithreaded traces with resource-oriented ordering. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. New York: ACM, 2013. 373–387

130 Huang J, Zhang C, Dolby J. CLAP: recording local executions to reproduce concurrency failures. In: Proceedings of the 34th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2013. 141–152

131 Devietti J, Lucia B, Ceze L, et al. DMP: deterministic shared memory multiprocessing. In: Proceedings of the 14th

Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2009. 85–96
132 Zhou X, Lu K, Wang X, et al. Exploiting parallelism in deterministic shared memory multiprocessing. J Parall Distrib Comput, 2012, 72: 716–727
133 Burnim J, Sen K. Asserting and checking determinism for multithreaded programs. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. New York: ACM, 2009. 3–12
134 Yin Z, Yuan D, Zhou Y, et al. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2011. 26–36
135 Sidiroglou S, Ioannidis S, Keromytis A D. Band-aid patching. In: Proceedings of the 3rd Workshop on Hot Topics in System Dependability. Berkeley: USENIX Association, 2007. 102–106
136 Gu Z, Barr E T, Hamilton D J, et al. Has the bug really been fixed? In: Proceedings of ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, 2010. 55–64
137 Jin G, Song L, Zhang W, et al. Automated atomicity-violation fixing. In: Proceedings of the 32th Annual ACMSIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2011. 389–400
138 Jin G, Zhang W, Deng D, et al. Automated concurrency-bug fixing. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2012. 221–246
139 Liu P, Zhang C. Axis: automatically fixing atomicity violations through solving control constraints. In: Proceedings of the International Conference on Software Engineering, Zurich, 2012. 299–309
140 Deng D D, Jin G L, de Kruijf M, et al. Fixing, preventing, and recovering from concurrency bugs. Sci China Inf Sci, 2015, 58: 052105