IEICE TRANS. INF. & SYST., VOL.E100–D, NO.11 NOVEMBER 2017

SURVEY PAPER

# Software Analysis Techniques for Detecting Data Race

**Pilsung KANG**[†a)], ***Member***

**SUMMARY** Data races are a multithreading bug. They occur when at least two concurrent threads access a shared variable, and at least one access is a write, and the shared variable is not explicitly protected from simultaneous accesses of the threads. Data races are well-known to be hard to debug, mainly because the effect of the conflicting accesses depends on the interleaving of the thread executions. Hence there have been a multitude of research efforts on detecting data races through sophisticated techniques of software analysis by automatically analyzing the behavior of computer programs. Software analysis techniques can be categorized according to the time they are applied: static or dynamic. Static techniques derive program information, such as invariants or program correctness, before runtime from source code, while dynamic techniques examine the behavior at runtime. In this paper, we survey data race detection techniques in each of these two approaches.

***key words:*** *concurrency, program analysis, data race*

## 1. Introduction

Multithreading is very popular in today's software. Typical examples include high-concurrency Internet servers which deal with simultaneous requests from large number of clients, and GUI (Graphical User Interface) components that have to repaint itself, respond to user input events, and perform spell-checking or play a song at the same time. However, multithreaded programming is error-prone and it is very easy to make synchronization mistakes, which causes data races. These data races occur when the programmer fails to properly protect a shared variable from concurrent accesses of multiple threads. Specifically, a data race occurs when at least two concurrent threads access a shared variable, and at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Data races are hard to debug. They are difficult to reproduce since the interleaving of thread execution depends on the scheduler, and sometimes they just remain undetected and change data structure invariants. This causes program failure later in the future, which makes it hard to trace back and can sometimes result in severe consequences [1]–[3].

There has been much effort to develop automatic tools for detecting data races. The detection techniques are broadly categorized according to the time they are applied to the subject program: static and dynamic. Static techniques try to extract the program information from source code before program runtime, while dynamic techniques examine the behavior at runtime. In this paper, we survey the data race detection techniques in line with these two approaches, together with a set of evaluation points.

The remainder of this paper is organized as follows. In Sect. 2, we describe two most commonly used data race detection models: the *happens-before* relation and the *locksets*. Then, we present representative tools and techniques for data race detectors in Sect. 3 for static approaches and Sect. 4 for dynamic approaches, respectively. In Sect. 5, we present design issues in race detection techniques. Finally, we summarize the survey and make conclusions in Sect. 6.

## 2. Models of Data Race Detection

In this section, we present two major models for detecting data race: happens-before relation and locksets.

### 2.1 Happens-Before Relation

One of the common approaches used in detecting data races is Lamport's *happens-before* relation [4], which is a partial order on all events of all threads in a system. The happens-before relation was originally developed to establish causality between the events in a distributed system. The happens-before relation, denoted by $\rightarrow$, is defined as follows.

- *Definition*: The relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If $a$ and $b$ are events in the same process, and $a$ comes before $b$, then $a \rightarrow b$. (2) If $a$ is the sending of a message by one process and $b$ is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events $a$ and $b$ are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

The data race verification procedure using the happens-before relation can be described by applying this definition to multithreaded programs as follows.

1. Within a thread, order the events as they occurred using the condition (1) in the above definition. Note that the scheduling of the events is nondeterministic depending on the execution platforms.
2. Between threads, determine the happens-before relation using the condition (2) in the above definition.

Manuscript received June 9, 2017.
Manuscript publicized August 9, 2017.
[†]The author is with Department of Computer Engineering, Youngsan University, South Korea.
a) E-mail: pilsungk@ysu.ac.kr
DOI: 10.1587/transinf.2017EDR0004

In particular, consider the unlocking function call in one thread in multithreading as sending a message by one process, and also consider the locking call in another thread as receiving the message in another process. This is because unlocking by one thread in multithreaded programs should happen before another thread can grab the lock, as the message sending by one process should causally happen before another process receives the message in distributed systems.

3. Check whether there is a pair of accesses to the same memory location that are *concurrent*. Based on the ordering of events determined by the steps above, we can say that a potential race is reported if two or more threads access a shared variable and the accesses are concurrent, which indicates that the variable is not properly protected and can be accessed simultaneously.

Figure 1 is a simple example of one possible execution ordering of a multithreaded program, where two threads execute a common code segment. Inside thread 1, three program statements are ordered sequentially and we can apply the rules (1) and (3) of the happens-before relation to get $1 \rightarrow 2$, $2 \rightarrow 3$, and $1 \rightarrow 3$. Similarly for thread 2, we get $4 \rightarrow 5$, $5 \rightarrow 6$, and $4 \rightarrow 6$. Between the two threads, we note that the locking of the mtx object for mutual exclusion by thread 2 follows the unlocking of mtx by thread 1, resulting in $3 \rightarrow 4$ when the rule (2) of the definition is applied. Considering the accesses to the shared variable x, $2 \rightarrow 5$ holds and there are no concurrent accesses. Hence, no races are detected by the happens-before relation based tools.

The key feature of the tools based on the happens-before relation is that they theoretically report no false positives [5], since when they report a data race, it means that there is at least one alternative execution schedule where the accesses happen simultaneously. Another advantage is that the happens-before relation based tools do not depend upon specific synchronization styles. Since the timing relations for detecting races are used, the tools can handle any synchronization primitives including locks and semaphores. But historically this approach has been hard to implement efficiently, because it requires per-thread information about concurrent accesses to each shared-memory location, which can be significant to store and manage (see Sect. 5.2).

Another serious drawback is that the effectiveness of the tools is highly dependent on the interleaving produced by scheduler, which causes it to miss valid data races in some cases. Figure 2 explains this with a simple example.

```
      Thread 1              Thread 2

1   lock(mtx);
2   x = x + 1;
3   unlock(mtx);
4                         lock(mtx);
5                         x = x + 1;
6                         unlock(mtx);
```

**Fig. 1** Ordering of events in multiple threads by happens-before relation.

The program execution here is a valid ordering with respect to the happens-before relation. Each statement within each thread is sequentially ordered and the locking of mtx by thread 2 happens after the unlocking of the same synchronization primitive by thread 1. However, there is a potential race with the shared variable *y* between the two threads since it is not properly protected by some locks. This potential race can be a real data race in another program execution where the two statements that access *y* occur concurrently. A major approach for overcoming this problem uses weaker or relaxed versions of the happens-before relation to explore a bigger set of event reorderings [6], [7].

## 2.2 Locksets

A lock is a simple synchronization object used for mutual exclusion of a shared variable. Locksets-based tools are based on the following simple observation: A shared variable must be protected at the time of access by a nonempty set of locks, which should supposedly protect the variable. Hence lockset-based tools maintain a set of locks, $C(x)$, for each shared variable $x$, and compares and refines $C(x)$ with the locks currently held by an accessing thread whenever it is accessed, and issues a warning if the thread does not own any locks common with $C(x)$. Figure 3 illustrates how lockset-based tools operate.

In Fig. 3, the candidate set of locks $C(x)$ is inferred during the previous execution history, and initialized to mtx1, mtx2. After the program starts and a thread $t$ grabs the lock mtx1, the set *locks_held(t)* changes from an empty set to contain mtx1. And when $t$ accesses the shared variable $x$, $C(x)$ is intersected with the current *locks_held(t)* and is re-

```
      Thread 1              Thread 2

1   y = 1;
2   lock(mtx);
3   x = x + 1;
4   unlock(mtx);
5                         lock(mtx);
6                         x = x + 1;
7                         unlock(mtx);
8                         y = y + 1;
```

**Fig. 2** Missed data race on *y* by the happens-before relation.

```
      program        locks_held      C(x)

                        { }      {mtx1,mtx2}

1   mutex mtx1,mtx2;
2   lock(mtx1);         {mtx1}
3   x = x + 1;                        {mtx1}
4   unlock(mtx1);        { }
5   lock(mtx2);         {mtx2}
6   x = x + 1;                         { }
7   unlock(mtx2);        { }
```

**Fig. 3** Refining locksets for detecting data races.

fined to mtx1. Later, when $t$ accesses $x$ again with only mtx2, the current $C(x)$ which contain only mtx1 is intersected again with *locks_held(t)* and it becomes an empty set since these two sets have no common locks. Therefore, the detector issues a warning.

The concept of locksets was first introduced by Dinning and Schonberg [8], with the name of *lock covers technique*. A major disadvantage of the lockset-based tools is that they are not very applicable to the programs that use other synchronization primitives than locks, such as semaphores. This is because the concept of locksets is based on the ownership of locks by thread, which makes it difficult for the lockset-based tools to infer which variables are supposed to be protected when other synchronization primitives are used.

## 3. Representative Cases: Static Techniques

Static techniques try to analyze the program to obtain information that is valid for possible executions. The foremost advantage of using static techniques is that actual application execution is not required. Unlike dynamic techniques where the application is actually executed and runtime overhead is unavoidable in doing so, static techniques find errors, even in hard-to-reach areas at runtime, in a program by source code analysis.

The major weakness of static techniques is that the extracted properties by static analysis are only approximations of the properties that actually hold when the program runs. This imprecision means that static analysis may provide not so accurate information to be useful. In addition, static techniques suffer from false positives in general because they are inherently conservative and tend to overestimate the number of shared locations in memory. However, with the wider use of strongly typed languages and increased hardware capabilities, the research in exhaustive and rigorous static approaches is getting more active [9]–[11]. In this section, we summarize major static techniques and tools for detecting data race.

### 3.1 RacerX

RacerX [12] is a static detector that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. The system is composed of the following five phases: Retargeting with user input, control flow graph (CFG) extraction, running checkers over the CFG, post-processing and ranking likely races, and inspection. Detection method is based on the static application of lockset analysis to the extracted CFG. RacerX infers checking information such as which locks protect which operations, which code contexts are multithreaded, and which shared accesses are dangerous, by performing depth-first search (DFS) over the CFG. During the DFS traversal, RacerX adds and removes locks as needed, and calls race checkers on each statement in the graph. For efficiency, caching is used to remove redundant checking along the DFS traversal. User supplied information is substantial for effective operation of RacerX. This information is needed to capture the synchronization styles used in the test cases, such as locking/unlocking functions or enabling/disabling interrupts. Also users may provide annotator routines that mark whether routines are single-threaded, multi-threaded, or interrupt handlers. The annotation overhead is reported as modest: less than 100 lines of annotations for millions lines of checked code, which makes RacerX attractive for large programs.

RacerX seems to be the first try to detect races against large programs. They applied RacerX to three operating systems code bases, including Linux, FreeBSD, and a commercial system which they call system X. The performance measurement shows quite promising results. First of all, it is fast. They reported that it took only 2 to 14 minutes to analyze 1.8 million line system. With respect to accuracy, RacerX found 3 bugs for Linux and 7 bugs for System X, although it generated false positives too.

### 3.2 Chord

Chord [13] is a static race detector for Java programs. Starting from an initial set of all the memory access pairs, Chord applies a series of static analysis steps to narrow down potential race candidates. First, it determines if a given access pair is actually reachable from the main method. Then, the next stage prunes the reachable pairs by checking if the accesses in a pair are aliases such that they cause a conflict for the same memory location. The third stage continues pruning by checking if the data accessed by the pair is thread-shared. Finally, the last step analyzes if a pair of accesses are properly protected by a common lock. At the heart of the Chord's algorithm is a context sensitive points-to analysis [14] for computing precise aliases and locksets, which directly affects the size of the search space in pruning.

The authors report that Chord discovered tens to hundreds of previously unknown bugs when applied to a suite of multithreaded Java programs as large as 646K LOC (lines of code). However, it has been reported in other experiments that Chord suffers from false positives [15], the notorious shortcoming in static detection techniques.

### 3.3 Type-Based Race Detection

Abadi et al. [16] propose an annotation-based approach based on a formal type system for capturing common synchronization patterns. The primary design goal of the system is provide a cost-effective way of static detection by minimizing both the number of annotations required and the number of false alarms produced. In this approach, the type system is used to verify the lock-based synchronization discipline. It associates a protecting lock with each field declaration, and tracks the set of locks held at each program point. Each field declaration is annotated with guarded-by $l$, to indicate that the field is protected by the lock expression $l$. The type system then verifies that this lock is held whenever

the field is accessed or updated. In addition, each method declaration is annotated with requires *l1, . . . , ln*, to indicate that the locks *l1, . . . , ln* are held on method entry. And the type system verifies that these locks are indeed held at each call-site of the method, and checks that the method body is race-free given this assumption.

Since this approach requires annotations, it heavily relies on user-supplied input. In addition, it relies on the programmer to aid the verification process by providing additional type annotations. The type system was implemented for the full Java language and the implemented race condition checker, rccjava, supports inferring default annotations for unannotated classes and fields, thereby saving a significant amount of time for annotating large programs. The checker was applied and found races in standard Java libraries and other applications. But the annotation overhead is still considerable: about one per 50 lines of code.

### 3.4 IteRace

IteRace [11] focuses on the use of parallel language features in Java 8 [17] in application code, aiming at producing less false positives with improved performance. Like most static approaches, IteRace performs pointer analysis to produce a call graph for program execution and a control flow graph for each method, which are then used to compute locksets and potential races for each statement in the program. In contrast to other approaches that use only one abstract thread in modeling the forked threads of a parallel loop, IteRace uses two distinct threads to distinguish between thread-specific objects. This 2-Threads model effectively reduces the number of memory locations to track because thread-specific objects are modeled separately.

The authors report that IteRace found 6 bugs in 7 open source projects with orders of magnitude less false warnings compared to Chord. However, IteRace is specialized for only the lambda-style parallel loops in Java and cannot handle other concurrency cases like explicitly spawned threads, which limits its applicability.

## 4. Representative Cases: Dynamic Techniques

Dynamic detectors instrument the program to extract relevant information at runtime. The detection results are usually valid for the run in question, but make no guarantees for other runs. Also, dynamic monitoring requires quite heavy computations, in that it consumes significant time to run test cases. Furthermore, their dependence on invasive instrumentation typically rule out their use on low-level code such as OS kernels and device drivers, although these are the very programs where concurrency errors are most dangerous.

However, dynamic techniques have the advantage because detailed information about a single execution is usually much easier to obtain than comparably detailed information that is valid over all executions. In addition, they have more realistic and accurate views of program behavior by considering actual execution paths. In this section, we present major dynamic techniques and tools for detecting data race.

### 4.1 Eraser

Eraser [18] is widely known as the first dynamic detection tool that applied the lockset discipline, which imposes that every shared variable must be protected by some lock at program execution. Any access to a shared variable unprotected by some lock is considered an error. Specifically, Eraser keeps track of a set of all locks, *locks_held(t)*, held by a thread $t$ during each shared variable access. In the meantime, it initializes a candidate set of locks, $C(x)$, for each variable $x$ to hold all possible locks and updates $C(x)$ on each access by intersecting $C(x)$ and *locks_held(t)*. If the candidate set of locks becomes empty, this implies that the variable is not shared by appropriate locks, and a warning is issued. Eraser further refines this procedure to support common programming practices that violate the discipline, but are not real data races such as initialization, read-shared data, and reader-writer locks.

Eraser was implemented at the binary level via binary rewriting into test applications by instrumenting each load/store for maintaining $C(x)$, lock acquire/release for maintaining *locks_held(t)*, and calls to the storage allocator for initializing $C(x)$, respectively. Eraser was reported to have found many race conditions in the tested server programs, although it produced false alarms too, for instance, when the memory locations are privately recycled without communicating with Eraser system. In terms of performance, Eraser was reported to slow down the applications up to 30×, and half of the slowdown was attributed to the procedure call overhead caused by instrumenting each load/store for maintaining $C(x)$.

### 4.2 Hybrid Dynamic Detection

The detection technique proposed by O'Callahan and Choi [19] is the first hybrid form of the happens-before relation and the locksets. It uses the locksets method for its performance advantage, while trying to suppress false positives by using happens-before relation method. Since combining these two different techniques would cause significantly larger computational overhead than using only either one of the two, optimizing the performance is critical. The foremost optimization technique by the hybrid detection is the use of the limited form of the happens-before detection, rather than fully supporting it. Specifically targeting Java programs, it keeps track of only the start(), join(), wait(), and notify() methods, and the happens-before relations are constructed on the events generated by instrumented codes whenever any one of these methods is called. It excludes shared memory accesses and locking/unlocking pairs. This limited form of the happens-before relation is reported to be very useful for suppressing false positives, while greatly reducing the number of thread messages and the overhead of maintaining vector clocks [20]. And the check they perform

at runtime is just the conjunction of the locksets detection check and the limited happens-before detection check.

The system is implemented in two-phase mode to reduce the number of memory locations for possible races. First, the detector runs in *simple mode*, where only locksets detection is used to efficiently identify all Java fields for possible races. Then, the user runs the detector in *detailed mode*, which instruments accesses to only these "race-prone" fields and performs the hybrid check. Experimental results with a variety of Java programs report bugs in many of the programs as well as false and benign races. The detection overhead results were acceptable in most cases, but were intolerable in a few cases. For instance, the simple mode detection for raytracer ran about 27 times slower.

### 4.3 ThreadSanitizer

ThreadSanitizer (TSan) [21] is a hybrid race detector that employs both the happens-before relation and the locksets. It observes the memory access events and synchronization events of a running program and based on this history, it constructs and updates the state information about global lock states and happens-before orders for each memory location. By checking the state information of a running program, TSan detects races when the accesses to the same memory location are concurrent by the definition of the happens-before relation. TSan offers an annotations API for the user to specify different forms of synchronization in the program, so that false positives are effectively suppressed through the support from the user.

The original version of TSan was implemented as a Valgrind [22] tool for binary instrumentation, resulting in very slow performance with almost 20× to 300× slowdown. In later versions, the implementation changed as a compiler instrumentation with a redesigned runtime library, which improved the performance significantly. For instance, TSan included in the Go language [23] reports 2× to 20× slowdown for typical programs. TSan has become very popular to be bundled with major compilers including Clang [24] and the GCC (GNU Compiler Collection) C/C++ front ends.

### 4.4 DataCollider

DataCollider [5] is a dynamic technique for kernel data race detection. It utilizes the hardware breakpoint mechanisms in detecting races, which allows for neutrality over complex architecture-specific synchronization protocols and different locking primitives that can co-exist in the kernel code. Unlike the usual user-mode programs, directly applying either the happens-before or the lockset model is difficult for the kernel code because synchronization abstractions such as threads are not clearly defined due to frequent handlings of hardware events like interrupts and DMA (direct memory access). To overcome the runtime overhead issue in dynamic techniques, DataCollider randomly samples instructions that access memory, instead of monitoring all memory accesses, from disassembled program binary. DataCollider inserts code breakpoints at sampled instructions and checks for conflicting accesses by other threads when the breakpoints are fired. To detect conflicts, DataCollider checks the change in value at the memory location by using a data breakpoint (or hardware watchpoint), a debugging facility provided by modern architectures.

DataCollider has been implemented for the Windows 7 kernel on the x86 architecture. The authors report that 25 race bugs were found from various drivers and the core kernel while running kernel stress tests with 1000 code breakpoint samples per second, which incurred only 5% runtime overhead.

## 5. Issues in Data Race Detection

In this section, we present four important design issues in developing and evaluating data race detectors: detection accuracy, overhead, scalability, and usability.

### 5.1 Accuracy

Accuracy is one of the most fundamental aspects of data race detection tools. Ideally, the tools must detect every real race while not issuing a warning on valid codes. However most of current tools, both static or dynamic, suffer from false alarms of races (incomplete), or do not effectively detect real races (unsound). Furthermore, even true data races can be benign and would not do any harm to the system operation. It has been reported that the amount of such harmless data races can be so big as 76%–90% of the true data races reported by the modern detectors [25], which calls for a sophisticated detection mechanism for distinguishing harmful races over harmless ones [26].

Tools based on the happens-before relation "theoretically" do not issue false alarms, but in practice, their detection accuracy depends on the thread interleavings generated by schedulers. On the other hand, locksets-based tools catch races irrespective of actual thread interleavings, but these tools suffer from false alarms because perfectly valid race-free code can violate the lockset requirement. Hence effective suppression of false alarms is an important issue for these tools. Note that there is an interesting relation between happens-before detection and locksets-based detection, which shows that the races reported by a full happens-before detector are a subset of the races reported by lockset-based detection [19].

#### False Positives

False positives can be broadly divided into two categories by their nature. One type of false alarms is those generated when it is not a true data race but warnings are issued because detectors failed to get enough information about this. For instance, Eraser reports alarms for private implementation of multiple reader, single writer locks, since these are not part of the POSIX thread (or `pthread`) [27] interface that Eraser implements.

```
if (x == NULL) { //tests x without locks
  lock(mutex);
  if (x == NULL) {
    set(x);
  }
  unlock(mutex);
}
```

**Fig. 4** Benign race in double-checked locking.

The other type is benign races, which are true data races but do not affect the program correctness. These races are usually intentional for performance reasons which try to avoid synchronization overhead. Typical examples include double-checked locking and lazy initialization. Figure 4 illustrates an example of double-checked locking.

Since the null test of the shared variable $x$ is executed by every thread, it would be costly for every thread to get the lock just to see that $x$ is not null in most cases. Hence, the null test is doubled such that the first check is done without costly locking operations and just passes the whole *if*-block when $x$ is not null. Otherwise, if $x$ is null, the check is done again seriously with a proper lock. The locksets-based detectors will issue a warning for the first null check since the check is done without protecting the shared variable with proper locks.

**False Negatives**

False negatives are undetected races and a *sound* detector is one that guarantees to find all races. Certainly, there is no perfect detector and it is also usually hard to tell how many real data races the detector failed to discover unless you have much information about the test program. False negatives are more serious for static detectors because any analysis errors that cause false negatives just go silent. Therefore, significant emphasis should be made on detecting such silent failures [12]. Some tools based on either of the happens-before relation or the lockset refinement are known to be vulnerable to the false negatives caused by interleavings of the scheduler [18], [28]. In addition, there can be a trade-off of what to effectively suppress between false positives and false negatives [18].

### 5.2 Analysis Overhead

Dynamic tools often suffer from time and space overhead at runtime. In fact, it is known that the problem of precise race detection is *NP-hard* in general [29]. Therefore, how and where to focus detection efforts on the given programs is the key to realizing an efficient detector. Dynamic tools typically instrument existing binary programs and this incurs runtime overhead. They usually instrument each load and store of shared memory locations, each call to locking and unlocking calls, and each initialization and allocation of memory. And this causes significant overhead for dynamic tools, where 10× or 100× performance slowdown is not unusual [21], [30], [31]. Hence, there has been a great amount of work to improve the runtime overhead in dynamic techniques. Some approaches sample and examine a very small portion of the entire memory accesses, sacrificing accuracy with increased false negatives [5], [32], [33]. Other approaches exploit the support from custom hardware [34], [35] or commodity hardware [36], [37] like transactional memory [38]. Another approaches include parallelizing detection efforts [39], removing unnecessary information about access orderings [40], and combining the lockset algorithm with happens-before reasoning [41].

**Space Overhead**

Space overhead is one of the hard challenges for detectors based on the happens-before relation, since it requires maintaining a large amount of per-thread information including memory location, access time, and locks. For instance, a data structure for read access used by Dinning and Schonberg [8] needs to store $N * 2^K$ entries, where $N$ is the number of threads and $K$ is the number of locks. In contrast, the lockset-based detectors allow for simpler implementation, in that they need only information about the set of locks for threads and shared memory. So O'Callahan and Choid [19] proposes a hybrid race detector which combines lockset-based detection with a limited form of happens-before detection. Moreover, another locksets-based detector reports significant improvement in memory overhead, as well as runtime overhead, by shifting the granularity level of race detection to objects [28].

### 5.3 Analysis Scalability

Scalability is emerging as a new important issue in data race detectors, due to the sheer size and complexity of modern software programs that race detectors have to deal with. In addition, today's highly concurrent programs like Internet-scale server applications are programmed with lots of multiple threads and locks. The good detectors need to maintain high detection accuracy, and also their performance should slow down gradually rather than abruptly as the number of threads and locks increases.

Static techniques tend to be more advantageous as the size of the code base grows than dynamic techniques that suffer from performance overhead. One major approach of dynamic techniques for dealing with the scalability issue is to sample only a small percentage of memory accesses instead of analyzing the entire behavior, as described in Sect. 5.2. In a similar manner, static techniques often make trade-offs between accuracy and efficiency in order to scale, such as using less program information [10], focusing on potentially dangerous program points only [12], or not tracking function pointer aliasing across files [9]. Lockset-based detection is reported to be insensitive to the number of threads in terms of accuracy [18]. And the happens-before approach will suffer from performance degradation as the number of threads and locks increases, since it requires more and more space for storing information as described above.

### 5.4 Detector Usability

After all, race detectors are a tool. Most users want easy-to-use and fast, while effective, detectors. Since most detectors need user input to capture program information and need to constantly communicate with users for analysis, effective detectors try to extract as much as information with as little user input as possible.

Some detection tools are specialized for applications written in a certain concurrent programming model or languages. This kind of tools can be very effective in finding races in terms of accuracy and performance because they can utilize innate characteristics of the used programming model or language. For instance, for OpenMP [42] applications, on-the-fly detection mechanisms have been developed [43]. However, these techniques cannot be directly applied for other parallel programming models like `pthreads`.

#### Program Annotations

Program annotations are a way of communication between detectors and test programs. Dynamic race detectors usually use program annotations to effectively suppress false positives [18], [19]. Annotations are frequently used in static techniques too. Warlock [44] implements annotations as compiler directives and use them in tracing execution paths for lock-based concurrent programs. IteRace [11] relies on the programmer to specify which part of the code is thread safe or not in order to filter true races.

Annotation is more than crucial for annotation-based tools. A type-based race detector by Abadi et al. [16] is based on programmer annotations to specify which lock should be held to access a variable. Hence, effective use of annotations while maintaining accuracy is a major challenge for them. In order to avoid the burden of manual annotations, the annotation process can be automated to some extent [45].

#### User Input about Program Information

Some tools need the user to supply program information or relevant hints in order to narrow the scope of the analysis, thus accelerating the detection performance. For instance, in RacerX [12], the user is supposed to supply a table specifying the functions used to acquire and release locks as well as those that disable and enable interrupts (i.e., synchronization styles). In this way, the user can get faster, more relevant, and more precise results for his or her program. Similarly, a fast dynamic detector with less than 5× overhead has been reported with the support from the user specifying the parts of the program for analysis [46].

#### User-Friendly Race Detection

RacerX has the ability of sorting out potentially significant races from trivial violations by using heuristics to identify and rank likely races [12]. This allows the user to quickly focus on dangerous points in large programs. The hybrid method [19] improved usability by reporting more information about detected races, which eases the debugging process.

## 6. Summary and Conclusions

This paper discussed static and dynamic techniques for detecting data races. We described the happens-before relation and locksets method as the most common approaches for data race detection. The happens-before relation exploits causality among the events generated by multiple threads, whereas the locksets principle imposes that a thread trying to access a shared variable must hold some locks which are supposed to protect the variable. We presented four aspects in evaluating data race detectors: accuracy, overhead, scalability, and usability. Accuracy is the most important goal in designing race detectors. Tools based on the happens-before relation are theoretically sound and do not generate false alarms but their accuracy depends on thread interleaving. On the other hand, locksets-based tools do not depend on thread interleaving, but they suffer from false alarms. On-the-fly dynamic tools that instrument existing binary program typically suffer from runtime overhead, hence some dynamic tools perform static analysis in advance to focus their detection efforts. Space overhead is a major challenge for detectors based on the happens-before relation, because they have to maintain a large amount of per-thread information.

Scalability is getting more critical as software becomes more complex on modern computing systems with parallel applications getting increasingly widespread. We foresee that software-only solutions are not sufficient for tackling the hard performance and scalability requirements of the modern data race detectors. In this respect, recent hardware-assisted and hardware-software co-design [47] approaches are promising to accomplish more perfection.

Last but not least, usability is an important aspect of the race detector. Most users want to find a small number of serious errors quickly for their large programs, rather than having a large number of trivial errors with slow tools. Therefore, making effective use of the user input as in the annotation-based tools for quickly capturing the synchronization styles and essential program information will be a key to realizing successful race detection tools on modern computing environments.

## Acknowledgements

This work was supported by a 2017 research grant from Youngsan University, Republic of Korea. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP; Ministry of Science) (No. 2017R1C1B2009361).

#### References

[1] N.G. Leveson and C.S. Turner, “An investigation of the Therac-25

accidents," Computer, vol.26, no.7, pp.18–41, 1993.
[2] K. Poulsen, "Tracking the blackout bug," 2004, http://www.securityfocus.com/news/8412
[3] J. Jackson, "Nasdaq's facebook glitch came from 'Race Conditions'," 2012. http://www.computerworld.com/article/2504676/financial-it/nasdaq-s-facebook-glitch-came-from--race-conditions-.html
[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol.21, no.7, pp.558–565, 1978.
[5] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," Proc. 9th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA, pp.151–162, 2010.
[6] K. Sen, G. Roşu, and G. Agha, "Detecting errors in multithreaded programs by generalized predictive analysis of executions," Proc. 7th IFIP WG 6.1 International Conference on FMOODS, Lecture Notes in Computer Science, vol.3535, pp.211–226, Springer-Verlag, Berlin, Heidelberg, 2005.
[7] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," Proc. 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.387–400, 2012.
[8] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pp.85–96, May 1991.
[9] J.W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE '07, pp.205–214, 2007.
[10] P. Pratikakis, J.S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," ACM Trans. Program. Lang. Syst., vol.33, no.1, pp.3:1–3:55, Jan. 2011.
[11] C. Radoi and D. Dig, "Practical static race detection for Java parallel loops," Proc. 2013 International Symposium on Software Testing and Analysis, pp.178–190, 2013.
[12] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," Proc. Nineteenth ACM Symposium on Operating Systems Principles, pp.237–252, Oct. 2003.
[13] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," Proc. 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pp.308–319, 2006.
[14] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," ACM Trans. Softw. Eng. Methodol., vol.14, no.1, pp.1–41, Jan. 2005.
[15] M. Eslamimehr and J. Palsberg, "Race directed scheduling of concurrent programs," Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pp.301–314, 2014.
[16] M. Abadi, C. Flanagan, and S.N. Freund, "Types for safe locking: Static race detection for Java," ACM Trans. Program. Lang. Syst., vol.28, no.2, pp.207–255, March 2006.
[17] "State of the lambda: Library edition," http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," ACM Trans. Computer Systems, vol.15, no.4, pp.391–411, Nov. 1997.
[19] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," Proc. 9th ACM Symposium on Principles and Practice of Parallel Programming, pp.167–178, June 2003.
[20] F. Mattern, "Virtual time and global states of distributed systems," Parallel and Distributed Algorithms, vol.1, no.23, pp.215–226, 1989.
[21] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," Proc. Workshop on Binary Instrumentation and Applications, pp.62–71, 2009.
[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp.89–100, 2007.
[23] A.A. Donovan and B.W. Kernighan, The Go Programming Language, 1st ed., Addison-Wesley Professional, 2015.
[24] "clang: A C language family frontend for LLVM," http://clang.llvm.org/
[25] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," Proc. 28th ACM Conference on Programming Language Design and Implementation, pp.22–31, 2007.
[26] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: Telling the difference with portend," Proc. 17th Conference on Architectural Support for Programming Languages and Operating Systems, pp.185–198, 2012.
[27] IEEE, POSIX.1c, Threads extensions, IEEE Std 1003.1c, 1995.
[28] C. von Praun and T.R. Gross, "Object race detection," Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.70–82, Oct. 2001.
[29] R.H.B. Netzer and B.P. Miller, "What are race conditions? Some issues and formalizations," ACM Letters on Programming Languages and Systems, vol.1, no.1, pp.74–88, March 1992.
[30] P. Sack, B.E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, "Accurate and efficient filtering for the Intel thread checker race detector," Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability, pp.34–41, 2006.
[31] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, "IFRit: Interference-free regions for dynamic data-race detection," Proc. International Conf. on Object Oriented Prog. Syst. Lang. and Apps., pp.467–484, 2012.
[32] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," Proc. 30th ACM Conference on Programming Language Design and Implementation, pp.134–143, 2009.
[33] M.D. Bond, K.E. Coons, and K.S. McKinley, "PACER: Proportional detection of data races," Proc. 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pp.255–268, 2010.
[34] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: Hardware-assisted lockset-based race detection," Proc. IEEE 13th International Symposium on High Performance Computer Architecture, pp.121–132, 2007.
[35] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "SigRace: Signature-based data race detection," Proc. 36th Annual International Symposium on Computer Architecture, pp.337–348, 2009.
[36] T. Zhang, D. Lee, and C. Jung, "TxRace: Efficient data race detection using commodity hardware transactional memory," Proc. 21st International Conference on Archi. Support for Prog. Lang. and Oper. Syst., pp.159–173, 2016.
[37] Y. Jiang, Y. Yang, T. Xiao, T. Sheng, and W. Chen, "DRDDR: A lightweight method to detect data races in linux kernel," Journal of Supercomputing, vol.72, no.4, pp.1645–1659, 2016.
[38] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," Proc. 20th International Symposium on Computer Architecture, pp.289–300, 1993.
[39] B. Wester, D. Devecsery, P.M. Chen, J. Flinn, and S. Narayanasamy, "Parallelizing data race detection," Proc. Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.27–38, 2013.
[40] C. Flanagan and S.N. Freund, "FastTrack: Efficient and precise dynamic race detection," Proc. 30th Conference on Prog. Lang. Design and Implementation, pp.121–133, 2009.
[41] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," Proc. 12th Symp. on Operating Systems Principles, pp.221–234, 2005.

[42] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," IEEE Computational Science and Engineering, vol.5, no.1, pp.46–55, 1998.
[43] O.-K. Ha, I.-B. Kuh, G.M. Tchamgoue, and Y.-K. Jun, "On-the-fly detection of data races in OpenMP programs," Proc. 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, pp.1–10, 2012.
[44] N. Sterling, "Warlock: A static data race analysis tool," Winter USENIX, San Diego, California, pp.97–106, Jan. 1993.
[45] C. Flanagan, K.R.M. Leino, "Houdini, An annotation assistant for ESC/Java," Symposium of Formal Methods Europe, pp.500–517, March 2001.
[46] M. Metzger, X. Tian, and W. Tedeschi, "User-guided dynamic data race detection," International Journal of Parallel Programming, vol.43, no.2, pp.159–179, 2015.
[47] J. Devietti, B.P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "RADISH: Always-on sound and complete race detection in software and hardware," 39th Annual International Symposium on Computer Architecture (ISCA), pp.201–212, June 2012.

**Pilsung Kang** is an Assistant Professor in the Department of Computer Engineering at Youngsan University, South Korea. His research interests include computational science, parallel systems, and high-performance software. Kang has a Ph.D. in computer science from Virginia Tech.