Software Analysis Techniques for Detecting Data Race

[CS 5204 OS Course Project: Fall 2004]

Pilsung Kang
Department of Computer Science
Virginia Tech
kangp@vt.edu

ABSTRACT

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 has been much effort to detect 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.

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 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.

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 runtime, while dynamic techniques examine the behavior at runtime. In this paper, we survey the data race detection techniques, along with the major design issues, in each of these two approaches.

The remainder of this paper is organized as follows. In Section 2 and 3, we describe two most commonly used approaches, *happens-before* relation and *locksets*, for detecting data races. In Section 4, we present issues in race detection techniques, including detection accuracy, overhead, scalability, and usability. After that we present case studies for data race detectors, Section 5 for static detectors and Section 6 for dynamic detectors. Finally, we summarize the survey in Section 7.

2. HAPPENS-BEFORE RELATION

One of the common approaches on detecting data races uses Lamport's *happens-before* relation [8], which is a partial order on all events of all threads in a system. This happened-before relations 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 $b \not\rightarrow a$.

We can apply this definition to multithreaded programs as follows. First, within a thread, order the events as they oc-

```
      Thread 1              Thread 2

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

Figure 1: Ordering of events in multiple threads by happens-before relation, given in [12]

curred. Now between threads, applying the condition (2) in the above definition to multithreaded programs, 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.

Now, based on this ordering of events in multithreading, 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 and 2, three program statements are ordered such that they satisfy the happens-before relation since they occur sequentially. And between the two threads, we note that the locking of *mtx* object by thread 2 follows the unlocking of *mtx* by thread 1. As described above, this satisfies the happens-before relation too because the lock can only be acquired after the previous owner releases it. Hence, this program execution ordering is valid in the view of race detectors based on happens-before relation.

The key feature of the tools based on the happens-before relation is that they basically report no false positives (explained later), since when it reports a data race, it means that there is at least one alternative execution schedule where the accesses happen simultaneously. Another advantage is that it does not depend upon specific synchronization styles. Since it uses timing relations for detecting races, it can handle any synchronization primitives including locks, semaphores, etc.

But historically this approach has been hard to implement efficiently, because it requires per-thread information about concurrent accesses to each shared-memory location. And 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    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;
```

Figure 2: Missed data race on y by happens-before relation, given in [12]

```
      program        1ocks_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);       { }
```

Figure 3: Lockset refinement, given in [12]

As with the example in Figure 1, the above program execution 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.

3. 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 $C(x)$. The example in Figure 3 shows how lockset-based tools detect a data race.

In the example in Figure 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 refined 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 in [2], with the name lock covers technique. A dynamic detection approach was proposed using the lock cover technique along with happens-before relation, but without any reports of a serious implementation. The first race detector that used locksets is Eraser [12].

A disadvantage of 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 by those semaphores.

4. ISSUES IN DATA RACE DETECTION

In this section, we present important design issues in developing and evaluating data race detectors. Specifically, they are analysis accuracy, overhead, scalability, and usability. We describe accuracy first.

4.1 Analysis Accuracy

Accuracy is one of the most important 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, or do not effectively detect real races.

Tools based on happens-before relation theoretically do not issue false alarms, but 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 [11].

4.1.1 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 `pthread` interface that Eraser implements.

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.

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

Figure 4: Benign race in 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.

4.1.2 False Negatives

False negatives are undetected 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 [3].

Some tools based on either of happens-before relation or lockset refinement are known to be vulnerable to the false negatives caused by interleavings of scheduler [12, 13]. For some tools, there is trade-off of what to effectively suppress between false positives and negatives [12].

4.2 Analysis Overhead

Dynamic tools suffer from time and space overhead at runtime. Especially, it is known that race detection is *NP-hard* in general [10]. Therefore, how and where to focus detection efforts on the given programs is the key to realizing efficient detector.

4.2.1 Time Overhead

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. For instance, applications that used Eraser were reported to slow down by a factor of 30 at maximum [12].

Although static detector do not suffer from runtime overhead, some tools which are geared for complete rigorous detection consume much more time during analysis. For instance, the proposed detector in [15] uses constraint solver for model checking, of which the performance is yet to be verified.

As today's software becomes huge and complex, and as most users want to quickly spot serious errors in their programs, careful consideration between detection accuracy and performance is needed.

4.2.2 Space Overhead

Space overhead is one of the hard challenges for detectors based on the happens-before relation, since it requires to maintain large amount of per-thread information, including memory location, access time, and locks. For instance, a data structure for read access in [2] needs to store $N * 2^K$ entries, where N is the number of threads and K is the number of locks.

In contrast, lockset-based detectors enables simpler implementation, in that they need only information about set of locks for thread and shared memory. So [11] proposes 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 [13].

Exhaustive static analysis needs huge amount of space. For instance, the model based approach that uses constraint solver [15] deals with large adjacency matrices and consumes large amount of memory for the search space over the matrices.

4.3 Analysis Scalability

Scalability is emerging as a new important issue in data race detectors, and is getting more and more attention. This is mainly due to the rapid growth of software programs, which the race detectors have to deal with, in terms of both size and complexity.

4.3.1 Number of Threads and Locks

Internet-scale server programs are highly concurrent with lots of multiple threads and locks. The good detectors need to maintain high detection accuracy, and also their performance needs to slow down gracefully as the number of threads and locks increases.

Lockset-based detection is reported to be insensitive to the number of threads in terms of accuracy [12]. 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.

4.3.2 Code Size

Today's race detectors are faced with very large and complex software programs. For instance, the test program used in [3] had millions of lines of code. Therefore, efficiently scalable approaches for detecting races are needed accordingly. [3] and [11] are nice examples with regard to this issue, in that they effectively pick up and focus on potentially dangerous or error-prone program points in large programs.

4.4 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.

4.4.1 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 [12, 11].

Annotation is more than crucial for annotation-based tools. A type-based race detector presented in [5] 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. For instance, [5] measured an overhead of one annotation per 50 lines of code at a cost of one programmer hour per thousand lines of code. In order to avoid the burden of manual annotations, there has been work on automating the annotation process [6].

4.4.2 User Input about Program Information

Some tools need the user to supply program information, in order to capture the synchronization styles used in the programs. For instance, in RacerX [3], the user supplies a table specifying functions used to acquire and release locks as well as those that disable and enable interrupts. In this way, the user can get faster, more relevant, and more precise results for his program.

4.4.3 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 [3]. This allows the user to quickly focus on dangerous points in large programs. The hybrid method proposed in [11] improved usability by reporting more information about detected races, which eases debugging process.

5. STATIC TECHNIQUES: CASE STUDIES

Static techniques try to analyze the program to obtain information that is valid for possible executions. And they can provide significant advantages for large code bases. That is, unlike a dynamic approach, static analysis does not require executing code. It immediately finds errors in obscure code paths that are difficult to reach at runtime. However, since they occur offline, they can also do analysis impractical at runtime. In other words, the extracted properties by static analysis are only approximations of the properties that actually hold when the program runs. This imprecision means that a static analysis may provide not so accurate information to be useful.

However, with the wider use of strongly typed languages and increased hardware capabilities, exhaustive and rigorous static approaches are getting more attractive [7, 3].

5.1 RacerX

RacerX [3] is a static detector developed by Dawson Engler at Stanford. It uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. The system

is composed of the following five phases: Retargetting with user input, control flow graph (CFG) extraction, running checkers over the CFG, post-processing and ranking likely races, and inspection.

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.

Detection method is based on static application of lockset analysis to the extracted control flow graph (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.

One of the key features of RacerX is the use of heuristics to sort out potentially significant races from trivial violations. In order to identify and rank likely races, it uses scoring functions to add or subtract points with regard to the following criteria: *Is the lockset valid? Is code multithreaded? Does X need to be protected?* The author reports that this ad hoc ranking does not seem overly sensitive to scoring, which sounds not very convincing.

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.

5.2 Type-based Race Detector

This is a static annotation-based approach, based on a formal type system that is capable of capturing many common synchronization patterns [5]. The system is designed to provide a cost-effective way of static detection by minimizing both the number of annotations required and the number of false alarms produced.

Since this is annotation-based, it heavily relies on user-supplied input. In addition, it relies on the programmer to aid the verification process by providing a small number of additional type annotations, which can also be used as documentation of the locking strategies. 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.

The previous version of the type system supported only Java classes with only internal synchronization [4]. They extended it to support patterns for external locks for client-side synchronization, and thread-local classes.

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. 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.

In order to allow classes to be parameterized by external locks, it uses ghost variables in class definitions. Thread-local classes require no synchronization and should not need to have locks guarding their fields. To indicate this information, the `thread-local` modifier is used on class definitions.

The type system was implemented for the full Java language. It was built on top of an existing Java front-end and required 5K lines of new code into existing code base. A nice feature of the race condition checker, *rccjava*, is its ability to infer default annotations for unannotated classes and fields by using heuristics. Though the heuristics are not guaranteed to produce the correct annotations, for about 90% for their test programs, it is reported to save a significant amount of time for annotating large programs.

They applied the checker to Java libraries including the `Hashtable` and `Vector` classes, `java.io`, and other test cases such as Ambit (the mobile object calculus), and WebL (the language for automating web based tasks). The detector reported race conditions in three of the five cases. But the annotation overhead looks quite much: about one annotation per 50 lines of code at a cost of one programmer hour per thousand lines of code. This overhead makes it not very practical for large, complex programs.

5.3 Memory-Model-Sensitive Race Detection

The detection technique proposed by Yang [15] is a formal approach to detecting races by specifying the memory model, which is a specification for thread semantics, and converting this into input constraints for constraint logic solver. Although current hardware technology has grown very powerful and its cost has significantly improved, logic solvers or theorem provers usually consume huge amount of computation time and resources since the search space covers all possible paths of program execution. Specifically, rigorous semantic analysis such as race detection is known to be NP-hard in general.

Therefore, the proposed method is not comparable to conventional race detection techniques in terms of performance and scalability. The authors aim to establish a sound basis for rigorous and precise race detection, which can be a basis for more efficient methods.

The basic idea is to specify threads' memory access rules into equivalent formal constraints and use them as an input problem for an existing constraint solver to automate the analysis. Then the challenge is how to accurately specify

the access rules, along with program semantics. In order to tackle this issue, the authors target Java as it provides built-in support for threads, and they extract the executable specification of race conditions from the Java memory model [1] which adopts a weaker form of sequential consistency [9] and offers formalization of concurrent memory accesses based on Lamport's happens-before order.

The specification format is a predicate logic, but the operators and notations in the predicate logic are specified using their own specification framework for memory models [14]. Also, they use a modestly extended predicate logic to effectively express a complex model, and try to construct a complete specification by being fully explicit about all ordering properties such as totality, transitivity, and circuit-freedom. Their model supports normal read/write, use of local variables, computation operations, control branches, and synchronization operations.

The constraint solving algorithm can be briefly described as follows.

1. Derive a program execution, which is a set of symbolic operation instances generated by program instructions, *ops* from the program text in a preprocessing phase
2. Suppose there are n *ops*. Then construct n by n adjacency matrix M, where the element $M(i, j)$ indicates whether operations i and j should be ordered
3. Go through each requirement in the specification and impose the corresponding propositional constraints with respect to the elements of M
4. If there is a binding of the free variables in *ops* such that it satisfies the conjunction of all specification requirements, then it is a data race
5. If not, no data race

The prototype was implemented straightforward using a constraint logic programming languages, Prolog. It was found that providing domain information significantly reduced the solving time and the search order among the constraints also impacts the performance. The experimental results for simple were reported, but it is unclear how to apply the proposed model-based approach to large programs.

6. DYNAMIC TECHNIQUES: CASE STUDIES

Dynamic detectors instrument the program to extract program 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 a 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 analyses have the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions. By operating at runtime they only visit feasible paths and have accurate views of the values of variables and aliasing relations [7, 3].

6.1 Eraser

Eraser [12] 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.

But at first, this simple scheme was too strict and did not work well. It failed to capture common programming practices that violate the discipline, but are not real data races. Three cases were reported: initialization, read-shared data, and reader-writer locks. For instance, it is very common that shared variables are frequently initialized without holding a lock. Hence, the lockset algorithm had to be refined to support these cases, and this was done mainly with the design of state transition diagram for shared variables. The following is the description of the state diagram.

1. *Virgin* : indicates a variable is newly allocated
2. *Exclusive* : enters this state once the variable is accessed. As long as it is accessed by only one thread, there is no change in state and $C(x)$. This takes care of the initialization issue
3. *Shared* : Enters from *Exclusive* state by a read access from another thread. $C(x)$ is updated in this state but data races are not reported even if $C(x)$ becomes empty. This takes care of the read-shared data issue
4. *Shared-Modified* : Enters this state from *Exclusive* or *Shared* state by a write access from another thread. $C(x)$ is updated in this state and races are reported if $C(x)$ becomes empty

Eraser was implemented at the binary level using code instrumentation into test applications on Digital Unix system. Eraser instruments each load/store for maintaining $C(x)$, lock acquire/release for maintaining $locks\_held(t)$, and calls to the storage allocator for initializing $C(x)$. It treats each 32bit word in heap or global data as a possible shared variable. Every memory word has a shadow word for storing lockset information and variable states.

Eraser was not optimized and reported to slow down the applications by a factor of 10 to 30, and half of the slowdown was attributed to the procedure call overhead. It reported

to have found many race conditions in the tested server programs. It produced false alarms too, for instance, when the memory locations are privately recycled without communicating with Eraser system. To effectively suppress this kind of false alarms without harming the real warnings, the authors devised appropriate program annotations, such as *EraserReuse(address, size)* which resets the shadow memory to *Virgin* state to indicate that the memory has been privately reused.

6.2 Hybrid Dynamic Detection

The detection technique proposed by O'Callahan and Choi [11] is a hybrid of happens-before relation and locksets for Java programs. 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, optimization is more than necessary. As described in Section 2, the overhead incurred by using happens-before relation alone is quite large in itself.

Therefore, the first optimization technique by the hybrid detection is the use of the limited form of the happens-before detection, rather than fully supporting it. Specifically, it keeps track of only $start()$, $join()$, $wait()$, and $notify()$ methods in Java programs, 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 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. And the check they perform at runtime is just the conjunction of the locksets detection check and this limited happens-before detection check.

Another optimization for efficient hybrid detection is to find redundant events and remove them. Redundant events are those that can be safely ignored without affecting the accuracy of race detector. For instance, suppose that there is a set of recorded events, E_m, for a specific memory location, and also suppose that a new event e is generated. Then if we can prove that for every future event e_f that races with e, e_f must race with at least one of previous events in E_m, the event e is redundant.

Two heuristics for detecting redundant events are used in the hybrid detection technique: Lockset-subset condition and oversized lockset condition checks. The first one is based on their vector clock implementation that only increments a thread's timestamp after it has sent a message. It checks if the thread, access type, and timestamp of a new event e matches those of one of previously recorded events. If there is one such event e_i, it also checks if e_i's lockset is a subset of e's lockset. If it is, the new event e is considered redundant.

The oversized lockset condition check is based on the observation that the number of locks held by a thread at any one time is very small and it infers a priori bound on the number of locks a thread can hold. It checks the number of locksets intersected by a new event e's locks with those of a set of previous events. If the number exceeds the bound, the new event e is redundant. The basic idea here is that for a new event e to be nonredundant, a future event must have a lockset which does not intersect e's lockset but does intersect the locksets of the prior events.

The system is implemented in two phases for performance reasons. 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. In fact, the simple mode run is not necessary and the user can specify the fields of his interest for the detailed mode run. But the two-phase mode to reduce the number of memory locations for possible races is recommended by the authors.

Experimental results with a variety of Java programs, including web-application servers Resin (67K lines) and Tomcat (54K lines), reports 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. The results also showed that two-phase mode run of the detector, where the detailed mode run is supported by the simple mode run, is essential. The test results where only the detailed mode run alone was used showed unacceptable overhead.

6.3 Object Race Detection

The detection technique proposed by Praun and Gross [13] is a specialized application of Eraser's locksets analysis to Java multithreaded programs at the level of objects, in the hope of reducing the detection overhead by exploiting specific properties of object-oriented programs, such as data encapsulation.

Consider an example case when all of the shared instance variables are accessed through one instance method. Traditional detectors that employ low-level view on data races will check every access to each of all shared variables, which incurs considerable overhead. But note that these access checks are equivalent to checking only the accesses to the instance method, which is an only way of accessing the shared variables. In order to take advantage of this observation, the proposed method chooses an object as the granularity of race checks. Since the accesses to an object's instance variables, which can be shared by and protected from concurrent threads, must be done through the object's reference, it is reasonable to try to detect a conflict at the object level.

Hence the authors extended the state transition model of shared variables in Eraser system into similar model at object level, which they call ownership model. The locksets are associated with objects, rather than variables. Thus, the detector keeps track of threads that have accessed a shared object, and the accesses change object states according to the model in the same way as in Eraser, and the locking discipline is checked against shared objects at runtime.

But in treating objects as the unit of protection, there can be more false alarms because the accesses to different instance variables cannot be distinguished. The authors claim that

"This tradeoff can be justified if objects play a significant role in the data space of a program."

In order to further reduce runtime overhead, static analysis is applied to identify references that only refer to thread-local data, and only accesses through non-thread-local variables are instrumented for runtime checks.

The performance results are quite promising. The authors reported the runtime overhead of 16–129% in time and less than 25% in space for typical benchmark applications, compared to the previous dynamic detectors which were reported to slow down applications by a factor of 2–80 in time with up to twice as much memory space.

7. SUMMARY

This paper has 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. Happens-before relation exploits causality among the events generated by multiple threads. For example, if two events are not related by causality, these two events are regarded as concurrent and will cause a data race eventually. Locksets principle imposes that a thread which tries to access a shared variable must hold some locks which are supposed to protect the variable. If this principle is violated, a locksets-based detector will issue a data race warnings.

We presented four issues in data race detectors: accuracy, overhead, scalability, and usability. Accuracy is the most important goal in designing race detectors. Tools based on happens-before relation theoretically 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. Dynamic tools that instrument existing binary program typically suffer from runtime overhead, hence some dynamic tools perform static analysis to focus their detection efforts. Space overhead is a major challenge for detectors based on happens-before relation, because they have to maintain a large amount of per-thread information.

Scalability and usability are getting more attention as software becomes increasingly large and complex these days. Many 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. They want easy-to-use tools. Hence, for example, annotation-based tools try to capture the synchronization styles and the program information with an effective use of user input.

We presented six different data race detectors, half of which for static and another half for dynamic approach, as a case study. We described each detector's basic ideas, detection methods, main features, and performance results. Each detector has its own strengths and weaknesses in terms of the four design issues described in Section 4. Some are good at accuracy but bad at performance perspective. Some others target at large programs while sacrificing accuracy. Some tools need exact formalization of program information to bootstrap accuracy while sacrificing performance.

Much work has been done for detecting data races in the past. And at present, researchers are working for better detectors. It will be nice if we can have a perfect detector eventually. But in the meantime, we should carefully consider different design principles in developing race detectors, for today's software is rapidly huge, complex, and domain-specific.

8. REFERENCES

[1] JSR133: Java memory model and thread specification. http://www.cs.umd.edu/~pugh/java/memoryModel.

[2] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, May 1991.

[3] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, October 2003.

[4] C. Flanagan and M. Abadi. Object types against races. In *Proceedings of CONCUR*, August 1999.

[5] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, June 2000.

[6] C. Flanagan and K. Leino. Houdini, an annotation assistant for esc/java. *Symposium of Formal Methods Europoe*, pages 500–517, Mar. 2001.

[7] D. Jackson and M. Rinard. Software analysis: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 133–145, June 2000.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. of the ACM*, 21(7):558–565, 1978.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[10] R. H. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[11] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, volume 38, pages 167–178, June 2003.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[13] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, October 2001.

[14] Y. Yang, G. Gopalakrishnan, G. Lindstrom, , and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[15] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. *To appear in 6th International Conference on Formal Engineering Methods (ICFEM'04)*, November 2004.