Race Detection in Parallel Programming

Yubo Wang
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000

yubo.wang79@gmail.com

ABSTRACT

Computer systems were sequential originally. This type of system design is simple but slow. In order to get to the power limit of CPU, people introduced multiprogramming, timesharing and multi-core technologies into computer systems. At the same time, some problems that never exist in the sequential systems emerged. Race condition is one of them. Race condition is an infamous bug in parallel system. The result of race condition is nondeterministic and extremely hard to reproduce and debug, which is obviously not what we want. Is it possible to detect race condition in parallel system? Can we find all of such problems? If so, can we solve it in a more graceful way? There are so many questions need to be answered. Clearly, if we can detect all of such potential threats and avoid them, we will get a more reliable and robust system.

In this paper, I will give an introduction to race condition. What is a race condition? Why is it so difficult to solve. Also, I will explore some methods trying to solve this problem. Basically, people detect race condition problem in two big category methods. The first one is detecting in the compile time, which is also called static detection; the other is detection in the run time, which is called dynamic detecting. Both of these two have shortcomings such as coverage, speed, etc. In this paper, I will give an introduction to some methods in both categories. Also, I will show you some tools in race detection.

Keywords

Race condition detection, parallel system, static, dynamic, tools

1. INTRODUCTION

1.1 What is race condition

Race condition occurs when two or more threads have shared data and at least one of them tries to change the shared data. Because OS decides the thread schedule algorithm and it can swap between threads at any time, user has no control over the order of how threads access the shared data. Therefore, the result of change in data is nondeterministic. It all depends on which thread wins the race.

1.2 How to Detect Race Condition

Clearly, such a situation is not what we expect and unacceptable. Normally, there are two category ways to solve this problem.

One method is detecting race condition in compile time. This method is referred to as static detection. Static detection does not need to really run the program. By scanning the whole program of metadata from a complied application or annotation codes, static analysis can obtain a thorough and detailed coverage. The shortcoming of static detection is there are also many false reports.

The other method is detecting race condition at runtime that called dynamic detection. Dynamic detection is more accurate compared to static method. However, because dynamic detection only find race conditions in the path of execution it will miss some bugs that are in the paths that are not covered.

1.3 Difficulties in Detection

When using multiple semaphores in static detection is NP-hard [9], which means it is hard to find an efficient solution. If the synchronization mechanism is weaker than semaphores, an exact and efficient algorithm does exist [10]. Otherwise, only heuristic algorithms are available [3, 5]. Because heuristic algorithms will only report potential race conditions, which means there may be false alarm in that many of reporting race conditions are not real ones. However, since detecting race condition is NP-hard problem, one will never know which of them are real race conditions. And, this is the reason why it is difficult to use a tool to find race conditions accurately.

1.4 Outline

- In section 2, I will discuss static race detection, including type-based system and context sensitive models.
- In section 3, I will introduce dynamic race detection, including Happens-before relation, Lockset algorithm and the hybrid way.
- In section 4, I will give some examples of tools of race detection.
- Section 5, is the summary.

2. STATIC DETECTION

Static analysis was mainly used in compilers. Nowadays, it is also used in many other areas such as program understanding, debugging, and testing and reverse engineering.

2.1 Type-based System

Any well-typed system is free of race condition. Flanagan et al. described a race-free type-based system for a concurrent object calculus [7].

Type system assigns a type to each of methods of an object. In addition, it defines a set of locks must be hold before invoking the method and the lock must be hold before updating the method. Each of these locks may be a special lock associated with the object, see Figure 1 [1], or, totally external to the object, which will be showed later.

We will check the locks that specific methods supposed to hold each time before we access the methods. By this way, one can guarantee that no two threads can try to access a shared data at the same time. As a result, race condition will not happen.

```
class Account {
  int balance = 0 guarded_by this;
  int deposit(int x) requires this {
    this.balance = this.balance + x;
  }
}

class DepositThread {
  final Account a = new Account();
  int run() {
    synchronized (a) {
      a.deposit(10);
    }
  }
}
```

Figure 1. Type system lock example

Some additional features are added to the initial system later paper [8]. These features:

1. Classes parameterized by locks, which allow the fields of a class to be protected by some lock external to the class. See Figure 2 [1].
2. Local-thread notion: The notion of objects is local to a particular thread and therefore safely accessible without synchronization. See Figure 3 [1].
3. Mechanisms for escaping the type system in place where it proves too restrictive, or where a particular race condition is considered benign.

```
class Node⟨ghost Dictionary d⟩ {
  String key = null guarded_by d;
  Object value = null guarded_by d;
  Node⟨d⟩ next = null guarded_by d;

  void init(String k, Object v, Node⟨d⟩ n)
      requires d {
    node.key = k;
    node.value = v;
    node.next = n;
  }
  void update(String k, Object v)
      requires d {
    if (this.key.equals(k)) {
      this.value = v;
    } else if (this.next != null) {
      this.next.update(k,v);
    }
  }
  ...
}
class Dictionary {
  Node⟨this⟩ head = null guarded_by this;

  void put(String k, Object v) {
    synchronized (this) {
      if (this.contains(k)) {
        this.head.update(k,v);
      } else {
        let Node⟨this⟩ node =
              new Node⟨this⟩() in {
          node.init(k,v,this.head);
          this.head = node;
        }
      }
    }
  }
  ...
}
```

Figure 2. External Lock

```
thread_local class LinkEnumerator {
  String text = null;
  int index = 0;

  void init(String t) {
    this.text = t;
  }
  boolean hasMoreLinks() { ... }
  String nextLink() { ... }
}


class Crawler {
  final Set visited = new Set();
  final Queue todo = new Queue();
  void run() {
    while (true) {
      String url = todo.dequeue();
      if (!visited.add(url)) {
        let String text = loadPageText(url) in {
          let LinkEnumerator enum = new LinkEnumerator() in {
            enum.init(text);
            while (enum.hasMoreLinks()) {
              todo.enqueue(enum.nextLink());
            }
          }
        }
      }
    }
  }
  ...
}
```

Figure 3. Local-thread Lock

There are some cases that threads can access object without synchronization. Such as:

- The object is immutable
- The object is accessed only to a single thread
- The variable contains the only reference to the object

2.2 Context Sensitive Model

In order to give a high coverage of program behavior, some tools try to explore all the possibility of threads interleaving, which has a practical problem – with the increase of threads number, the possibility will increase exponentially, which quickly increase the compute complexity. Thus, this kind of solutions limits the scalability of the analysis.

In KISS (Keep It Simple and Sequential) [15], the authors try to give a solution to avoid exponential complexity. Their solution based on a technique to transform a concurrent problem P to a sequential problem P', which will simulate the behaviors of large subset of problem P. Since problem P' is sequential, the problem of P' can be analysis by a checker, which only needs to know the semantics of sequential execution. More important is

that if problem of P' fails an assertion, then problem of P must fail the same assertion too. So, it's easy to pinpoint the error position in P.

And, it is need to note that KISS is a complete (no false errors) but unsound (may miss errors).

KISS is an assertion checker for multithreaded C program. It needs help from sequential model checkers such as SLAM. In Figure 4, KISS transforms concurrent C program to sequential C program. Using SLAM or other sequential model checkers such as PREfix [2], MC [6], ESP [4], and Blast [11], we can analyze C sequential problem. An error trace produced by SLAM can be transformed to error trace in the original concurrent program. Applying KISS to detect race conditions in Windows NT device drivers, the code sizes range from 1KLOC to 10KLOC, there are 30 total race conditions find.

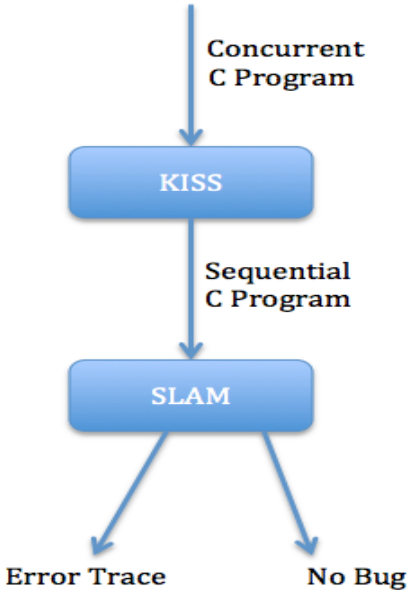

Figure 4. KISS Structure

In a concurrent program, each thread has its own stack. But, there is only one single stack for the unique thread. How to use only one stack to generate behaviors that simulating the interleaving between multithreads is a huge challenge. Using global variable access that takes value from set {0, 1, 2} as a flag to r. When access = 0, there is no access to r; when access = 1, read access happens to r; when access = 2, write access to r happens. Two functions $check_r$ and $check_w$ are used to check if there is race condition due to read or write access to r.

```
checkr(x) {
if (x == &r)
{assert (¬(access == 2)); access = 1; }
}

checkw(x) {
if (x == &r)
{assert (access == 0); access = 2; }
}
```

The test result of KISS on Windows NT device drivers is as Figure 5:

Driver	KLOC	Fields	Races	No Races
tracedrv	0.5	3	0	3
moufiltr	1.0	14	7	7
kbfiltr	1.1	15	8	7
imca	1.1	5	1	4
startio	1.1	9	0	9
toaster/toastmon	1.4	8	1	7
diskperf	2.4	16	2	14
1394diag	2.7	18	1	17
1394vdev	2.8	18	1	17
fakemodem	2.9	39	6	31
gameenum	3.9	45	11	24
toaster/bus	5.0	30	0	22
serenum	5.9	41	5	21
toaster/func	6.6	24	7	17
mouclass	7.0	34	1	32
kbdclass	7.4	36	1	33
mouser	7.6	34	1	27
fdc	9.2	92	18	54
Total	69.6	481	71	346

Figure 5. KISS Experimental Results

3. DYNAMIC DETECTION

Dynamic race detection happens at run time. As opposed to static race detection, which happens at compile time, dynamic race detection doesn't generate false assertion. Dynamic race detection is more accurate compare to static way. I will introduce some popular dynamic race detection methods, especially Happens-before and Lockset. And of course, dynamic race detection has its own drawbacks such as: slow speed and miss errors.

3.1 Happens-before

Happens-before defines a partial order for events in a set of concurrent threads.

- If there is only one thread, happens-before reflects the temporal order of events.
- If there are multiple threads, then thread A happens before Thread B if they both obey the rule when trying to access the lock and Thread A access the lock before B.
- Data race is possible if accesses to shared data are not ordered by happens-before.

When a thread tries to access a shared data, it must follow the rules:

[1] Acquire the lock.
[2] Do some operations.
[3] Release the lock.

Then, if thread a acquires the lock before thread b, there is a Happens-before relation between thread a and thread b.

In Figure 6, Thread 1 acquires the lock before Thread 2, so Thread 1 happens before Thread 2. Since Thread 1 releases the lock before Thread 2 tries to acquires the lock, there is no race condition. We will see different result in Figure 7 & 8, where the threads don't obey the rule of operating locks. And, race condition will exist in these two examples.

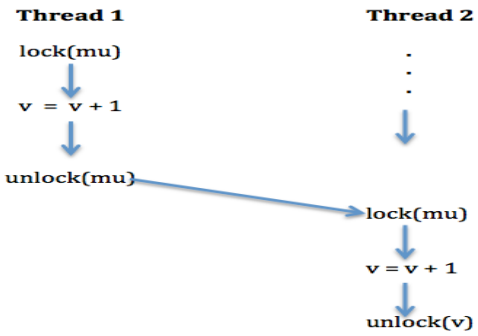

Figure 6. Happens-before Example

In Figure 7, Thread 1 acquires the lock before Thread 2, there is Happens-before relationship of Thread 1-> Thread 2. Accessing to variables of both y and v is ordered by Happens-before, so there is no threat of race condition.

However, there is a hidden thread in Happens-before relationship. Because Happens-before only detect race condition when the incorrect order of execution shows up, it will miss some errors. A totally different order of execution could bring up another result.

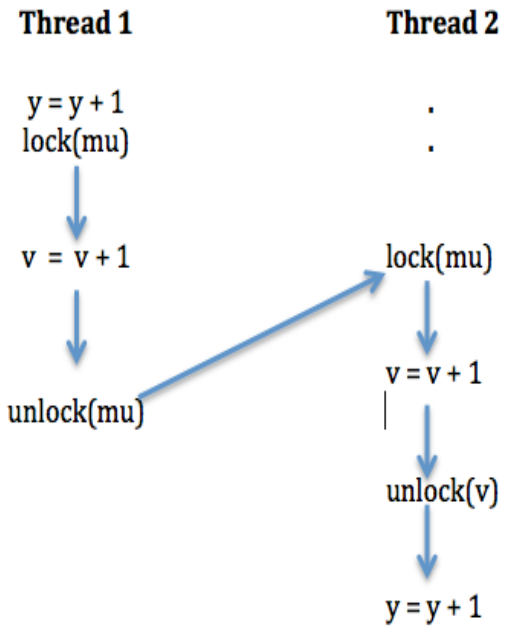

Figure 7. Happens-before with Two Variables Example

In Figure 8, Thread 2 executes before Thread 1. Since there is no Happens-before relationship of either Thread 1 -> Thread 2 or Thread 2 -> Thread 1, accessing to variable y is concurrent in the two threads. Variable y is not protected. There is race condition threat in this code.

There are many possible interleaving. Happens-before only finding race condition when dangerous schedule is executed. In the next section, we will go through a different method called Lockset. Compared to Happens-before, Lockset will find race condition in both path.

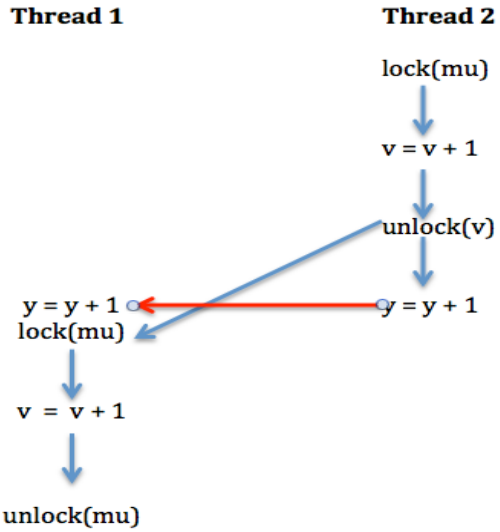

Figure 8. Happens-before not Hold

3.2 Lockset Algorithm

The basic idea of Lockset algorithm is that for every shared data, there must be a lock to protect it. Whenever a thread accesses the shared data, it must hold the lock. Eraser [16] will monitor all reads and writes to guarantee this discipline. Since Eraser has no idea of which lock to protect which data, it will infer the protection relation from the execution history.

3.2.1 First Version

For each v, initialize C (v) to the set of all locks.

On each access to v by thread t,

Set C (v) = C (v) ∩ Locks_held (t);

If C (v) = {}, then issue a warning.

V -- Shared data

C (v) – Candidate locks set for v

Locks_held (t) -- Locks hold be thread t

Figure 9 [16] illustrates how a potential data race is discovered through lockset refinement.

Program	*locks_held*	*C(v)*
	{}	{mu1,mu2}
lock(mu1);		
	{mu1}	
v := v+1;		
		{mu1}
unlock(mu1);		
	{}	
lock(mu2);		
	{mu2}	
v := v+1;		
		{}
unlock(mu2);		
	{}	

Figure 9. Eraser Lockset Algorithm Race Detection

3.2.2 Improving the locking Discipline

Clearly, the above locking discipline is too strict. There are three common situations that violate the discipline but free from data race condition.

- Initialization: Frequently, shared data do not need a lock when initializing.
- Read-Shared Data: Some data are read only after initialization, which do not need locks.
- Read-Write Locks: Read-write locks allow multiple readers to access a shared data, but only a single writer.

If there is no other threads can have the possibility to reference the being accessed data, there is no need for a thread to hold a lock, which is the situation of initialize data.

To avoid the false alarms caused by the writes in initialization. Eraser only considers the threads after the first one, where Eraser believes that the initialization has been finished. As long as the shared data is still accessed by the first thread, Eraser will not change the candidate locks set.

Simultaneous reads of a shared data by multiple threads are also not race. So there is no need to report a race when the data is race only. Figure 10 [16] illustrates the state transitions that show the timing of refinement and race reports.

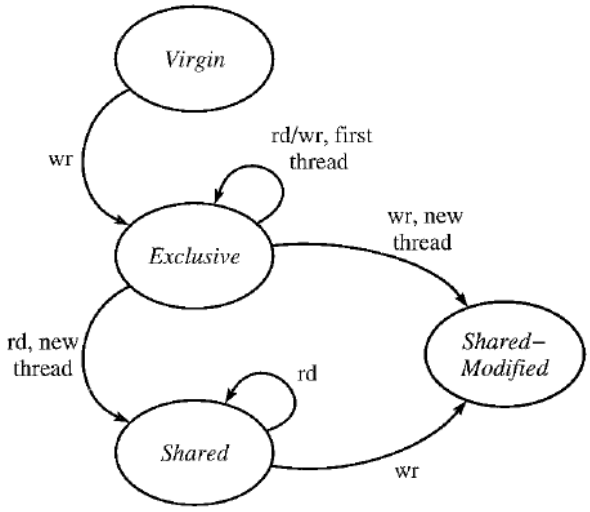

Figure 10. Lockset Algorithm States Chart

Read-Write locks refinement: If lock m protects data v, then m is held in write mode for every write of v, and m is held in some mode for every read of v. Locks held in exclusive read mode are removed from the candidate set when write happen.

On each read of v by thread t:

Set C (v) = C (v) ∩ locks_held (t)

On each write of v by thread t:

Set C (v) = C (v) ∩ write_locks_held (t)

If C (v) = {}, then issue a warning

3.3 Hybrid

Happens-before detector is precise but too conservative. It is too sensitive to thread interleaving to detect some data races. In opposite, lockset detector can find more races but are not precise. There will be false reports. By combining these two methods, hybrid detector can get better performance. We will learn how it works by Figure 11 [13]. A race condition exists when ChildThread sets main.childThread = null while main thread executes statement L. It is a very rare and hard to test.

Lockset will discover this bug for there is no common lock held to childThread field. Unfortunately, a similar potential race condition of main.gloalFlag will also be report.

However, a happens-before detector will know that there must be a happens-before relation to globalFlag in main thread and child thread, so it's not a race condition.

Starting with a lockset detector and refine the model with some limited happens-before checking.

```
// MAIN THREAD
class Main {
  int globalFlag;
  ChildThread childThread;
  void execute() {
    globalFlag = 1;
    childThread = new ChildThread(this);
    childThread.start();
    ...
    synchronized (this) {
      if (childThread != null) {

L:      childThread.interrupt();
      }
    }
  }
}


// CHILD THREAD
class ChildThread extends Thread {
  Main main;



  ChildThread(Main main) { this.main = main; }
  void run() {
    if (main.globalFlag == 1) ...;
    ...

    main.childThread = null;

  }
}
```

Figure 11. A Program With A Potential Race

4. RACE DETECTION TOOLS

There are many concurrent tools in the market to help you find race condition. Here are some on them [14].

4.1 CHESS

- Created by Microsoft Research
- User-mode scheduler
- Concurrency unit tests
- Every run takes a different thread schedule
- Reproducible

4.2 Intel Thread Checker VS. Intel Parallel Inspector

Table 1 [12] shows the comparison of the two tools.

Table 1. Intel Thread Checker VS. Parallel Inspector

	Intel® Parallel Inspector	Intel® Thread Checker
Threading errors - Data races and Deadlocks	√	√
Does not require special build or source code	√	√
Memory errors	√	
Easier to learn and reuse	√	
Low overhead analysis	√	
Improved scalable analysis without serializing the app	√	
Windows* standalone		√
Linux* support		√
Licensing	Single User	Single User & Floating
Support	forum support premier support option	unlimited premier support & 1 year product updates

4.3 RaceX

- Flow-sensitive static analysis tool
- Light burden to source system
- First round builds CFG (Control Flow Graph)
- Second round runs race checker and deadlock checker
- Last round generates reports based on priority

4.4 KISS

See section 2.2

4.5 Go Race Detector

- Integrated with go tool chain
- Compiler instruments all memory accesses with code
- Runtime library monitors unsynchronized accesses to shared data
- Ten-fold burden to CPU and memory
- Load test and integration test

Figure 12 gives an example how to use Race detector in command line.

```
$ go test -race mypkg    // test the package
$ go run -race mysrc.go  // compile and run the program
$ go build -race mycmd   // build the command
$ go install -race mypkg // install the package
```

Figure 12. Go Race Detector

5. SUMMARY

Race conditions are the most insidious bugs in parallel computing, because they are nondeterministic and hard to reproduce or debug. In this paper, I reviewed several common methods to solve race condition. People either solve it with static methods (in compile time), or with dynamic methods (in run time). Also, gives some examples of popular race detection tools.

6. REFERENCES

[1] Abadi, M., Flanagan, C. and Freund, S. N. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28, 2 2006), 207-255.

[2] Bush, W. R., Pincus, J. D. and Sielaff, D. J. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30, 7 2000), 775-802.

[3] Choi, J.-D. and Min, S. L. Race Frontier: reproducing data races in parallel-program debugging. *SIGPLAN Not.*, 26, 7 1991), 145-154.

[4] Das, M., Lerner, S. and Seigle, M. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Not.*, 37, 5 2002), 57-68.

[5] Dinning, A. and Schonberg, E. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26, 12 1991), 85-96.

[6] Engler, D., Chelf, B., Chou, A. and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Proceedings of the 4th conference on Symposium on Operating System Design \& Implementation - Volume 4* (San Diego, California, 2000). USENIX Association, [insert City of Publication],[insert 2000 of Publication].

[7] Flanagan, C. and Abadi, M. Object Types against Races. In *Proceedings of the Proceedings of the 10th International Conference on Concurrency Theory* (1999). Springer-Verlag, [insert City of Publication],[insert 1999 of Publication].

[8] Flanagan, C. and Freund, S. N. Type-based race detection for Java. In *Proceedings of the Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (Vancouver, British Columbia, Canada, 2000). ACM, [insert City of Publication],[insert 2000 of Publication].

[9] Ghosh, R. H. B. N. S. On the Complexity of Event Ordering for Shared-Memory Parallel Program Execution. *International Conference on Parallel Processing*1990), II93-II97.

[10] Ghosh, R. H. B. N. S. Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization. *International Conference on Parallel Processing*, II1992), 242-246.

[11] Henzinger, T. A., Jhala, R., Majumdar, R., Gr, #233 and Sutre, g. Lazy abstraction. *SIGPLAN Not.*, 37, 1 2002), 58-70.

[12] Intel http://software.intel.com/en-us/articles/intelr-parallel-inspector-comparison-with-intelr-thread-checker/.

[13] O'Callahan, R. and Choi, J.-D. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38, 10 2003), 167-178.

[14] Patil, R. V. and George, B. http://msdn.microsoft.com/en-us/magazine/cc546569.aspx.

[15] Qadeer, S. and Wu, D. KISS: keep it simple and sequential. *SIGPLAN Not.*, 39, 6 2004), 14-24.

[16] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15, 4 1997), 391-411.