# Thinking Beyond Race Conditions

Aoun Raza

Universität Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
*aoun.raza*@informatik.uni-stuttgart.de

## Abstract

Multi-threaded parallel programs perform accesses to shared variables, which require application of a synchronization strategy. Absence of synchronization among threads may lead to error situations, such as data races. Data races, which involve concurrent accesses to a single shared variable, are categorized as low-level. Synchronization strategies to alleviate low-level data races do not guarantee freedom from data races on a higher-level of abstraction. In this paper, we discuss some scenarios where accesses to a group of variables may result in data races. Further, we discuss a method to statically detect such situations and describe its integration into Bauhaus tool suite.

## 1 Introduction

In general, low-level data races can be avoided with the use of mutual exclusion. However, this does not ensure the absence of data races on a higher-level of abstraction. This means, a program can manifest racy behavior when the states of a group of shared variables change non-atomically. Meaning, if threads manipulate states of a subset of that group of shared variables in parts, it can result into stale state. Large scale concurrent software systems may contain such scenarios hidden in them. In this paper, we briefly discuss some scenarios of high-level data races, and describe, how traditional race detection tools cannot analyze programs for them. These techniques are implemented in Bauhaus [3] static race detection tool RC_Analyser [2]. This paper is organized as follows, section 2 gives insight on high-level data races and discusses scenarios in which they can manifest. It further describes the techniques to statically detect these erroneous situations. In section 3 we discuss implementation and test results.

## 2 Anomalies and Detection Strategies

### 2.1 Non-Automic Protection

First scenario of high-level data races involves concurrent accesses to a group of variables, as shown in listing 1. This group of variables represents either object fields or components of structures. We name this scenario as *non-atomic protection* and formally define it as follows,

*A non-atomic protection occurs when two threads concurrently access a group $G_v$ of shared variables, which should be accessed atomically, but at least one of the threads accesses subsets of $G_v$ such that those partial accesses diverge.*

Let's assume two threads $T_a$ and $T_b$ manipulate an object $Obj_p$ of class Point. $T_a$ tries to obtain the current values of $x$ and $y$ components by issuing calls to $getX$ and $getY$ functions respectively. However, there is a possibility of a context switch between the two calls. In this case $T_b$ may get the opportunity and update the values of $x$ and $y$ by calling function $setXY$. After getting the CPU back, $T_a$ continues, and retrieves the updated value of $y$ and has old value of $x$, which is inconsistent according to the new state of $Obj_p$.

```
1  class Point{
2  private:
3   int x= 0; int y=0;
4  public:
5   synchronized int  getX ()
6         { return x;}
7   synchronized int  getY ()
8         { return y;}
9   synchronized
10  void setXY(int x, int y)
11        {this.x = x; this.y = y;}
12 }
```

Listing 1: Non-Atomic Protection

### 2.2 Lost-Updates

Second scenario deals with value dependency between two critical code blocks in a thread and a concurrent write in another thread. Following gives the formal definition of a *lost-update* situation,

*A lost-update occurs when a value dependency between two critical code blocks in one thread on a shared variable may not be fulfilled, because of a concurrent write access to that shared variable from another thread.*

An example of lost-update can be seen in listing 2, which shows code for operation *withdraw* from a bank account management system. Suppose a thread $T_a$ attempts to withdraw some amount (*amnt*) from the account, and at the same time another thread $T_b$ attempts to deposit. As depicted in the listing, there are

two critical code blocks ($cb_1$, $cb_2$) in withdraw with accesses to the variable *blnc* (balance). Now consider $T_a$ exclusively reads the balance from the account and exits $cb_1$ to check whether the account contains enough balance. If the account contains at least the required amount it enters in $cb_2$ and decreases the account balance with amount (*amnt*) to be withdrawn. Now, if thread $T_b$ is scheduled between $cb_1$ and $cb_2$ and performs deposit i.e., updates the account balance, then the updated value of balance (*blnc*) will be lost.

```
1  void withdraw(int amnt) {
2   lock(l);     [cb1 = {blnc (r)}]
3    int t = blnc;
4   unlock(l);
5   if (t >= amnt) {
6    lock(l);    [cb2 = {blnc (w)}]
7     blnc = t - amnt;
8    unlock(l);
9   }
10 }
```

Listing 2: Lost-update scenario

### 2.3 Detection Strategies

Detection of above described scenarios requires computation of lock protected code blocks (*cb*), shared variables accesses in them, and value dependencies among code blocks on per thread basis. The analysis requires program control-flow graph and SSA information of the input program.

**Lock protected code blocks:** The detection of lock protected code blocks is an extension to lockset-analysis [1]. The significant difference is, a lock *acquisition* operation triggers an entry against the lock marking the start of a *protected code block*. The lock *release* statement marks the end of a code block. All access to shared variables between start and end of a protected code block are recorded as $G_v$ along with their access kinds (*read*/*write*). For each thread all such protected code blocks are computed, if the analysis cannot find a lock release statement for some acquired lock then the protected code block will be undefined. Further, detection of lock acquisition statement in already started code block triggers the computation of nested code blocks.

**Computing value dependencies:** In essence a value dependency is a $read-write$ chain between protected code blocks of the same thread. To find such chains, we use data-flow techniques i.e., whenever the analysis encounter write access on a shared variable, it performs a backward analysis to locate its read location. If the read access happens to be in a previously detected code block, the analysis stores a value dependency between code blocks on that variable.

**Error Detection:** The error detection takes all threads of the program, computed code blocks, value dependency information and performs two different actions specific to error types. For the detection of non-atomic protection scenarios, a maximal protected code block w.r.t to shared variables is determined and compared against all protected code blocks in other threads. Threads containing protected code blocks, which are subsets of maximal code block, potentially contain errors. This process is repeated for all threads in the program.

To report lost-update scenarios, it is sufficient to find a thread which performs a write access on the shared variable for which a value-dependency has been recorded in another thread. Therefore, the analysis takes the value dependent shared variables, and checks for concurrent writes on them. The detected situations are reported as potential errors.

## 3 Implementation and Tests

The implementation is integrated into RC_Analyser, which is a part of Bauhaus tool-suite. RC_Analyser already contains lockset-analysis, which is further extended to compute protected code blocks. Further, a value-dependency computation module has been implemented. The recoded errors can be further diagnosed with interactive text based shell.

We have tested some opensource programs written in C/C++ using pthread library. The test results are listed in table 1, LU column shows found lost-updates and NaP column shows found non-atomic protections.

| Program | LOC | LU | NaP |
|---|---|---|---|
| pfscan | 1K | 2 | 3 |
| aget | 1K | 0 | 0 |

Table 1: Program statistics

## 4 Conclusion

In this work, we described how race conditions can manifest on a higher-level of abstraction with the help of example scenarios. Further, techniques for detecting such scenarios have been briefly discussed. Results show that these error scenarios are specific to certain types of programs. In future, we plan to integrate more scenarios in which high-level data races can manifest.

## References


[1] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03*, pages 237–252, New York, NY, USA, 2003. ACM.

[2] A. Raza and G. Vogel. RCanalyser: A flexible framework for the detection of data races in parallel programs. In *Ada-Europe 2008*, volume 5026 of *LNCS*, pages 226–239. Springer, 2008.

[3] A. Raza, G. Vogel, and E. Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Ada-Europe 2006*, volume 4006 of *LNCS*, pages 71–82. Springer, 2006.