Dynamische Programmanalysen für nebenläufige Programme - Data Race Prediction mit TSan V2

Seminararbeit

Student:	Frank Ling
Matrikelnummer:	79496
Universität:	Hochschule Karlsruhe – Technik und Wirtschaft
Studiengang:	Informatik, Master
Semester:	Sommersemester 2023
Dozent:	Prof. Martin Sulzmann
Bearbeitet am:	June 6, 2023

Contents

1 Introduction

Nowadays concurrent programs are very common in order to make use of 'hyper-threading and multi-core architectures'**SWB-1830643851**. 'Due to the highly non-deterministic behavior of concurrent programs' **sulzmann** data races may arise but can also be hard to find, as they also 'may only arise under a specific schedule' **sulzmann**. This seminar work shows the motivation and background for the data prediction tool ThreadSanitizer (TSan) V2, which differentiates itself from the first version by utilizing happens-before methods instead of the lockset method. Afterwards the concepts of the FastTrack **cormac** algorithm will be shown as TSan uses a slightly modified version of the FastTrack algorithm. Examples showing the limits of FastTrack and TSan follow, as they are both incomplete and thus do not find every data race. In the following the same notation as in **sulzmann** will be used for traces and events.

2 Motivation and Examples

As stated in the introduction concurrent programs are commonly used and are inherently prone to data races. The problem with data races is that they can cause undefined behavior and crashes while also being hard to reproduce. This makes data races hard to find and they might not always be apparent. This chapter will show a few examples from **sulzmann2** but rewritten in the programming language C++. The following example shows a program, which exhibits a data race (see chapter 3):

```
#include "pthread.h"

int var1;
pthread_mutex_t var2;

void *Thread1(void *x) {
    // w(x)1
    var1++;
    // acq(y)1
    pthread_mutex_lock(&var2);
    // rel(y)1
    pthread_mutex_unlock(&var2);
    return NULL;
}

void *Thread2(void *x) {
    // acq(y)2
    pthread_mutex_lock(&var2);
    // w(x)2
    var1--;
    // rel(y)2
    pthread_mutex_unlock(&var2);
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    pthread_create(&t[1], NULL, Thread2, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}
```

Listing 1: Program containing conflicting events

Acquiring and releasing a lock is accomplished by the `pthread_mutex_lock()`- and the `pthread_mutex_unlock()`-method. The following trace can be generated for the program

1:

	1#	2#
1.	w(x)	
2.	acq(y)	
3.	rel(y)	
4.		acq(y)
5.		w(x)
6.		rel(y)

Table 1: Possible trace of listing 1

In the trace 1 the events $1\#w(x)_1$ and $2\#w(x)_5$ do not appear right next to each other which might indicate that these two events are not part of a data race. By rerunning the program 1 eventually the following trace can be obtained:

	1#	2#
4.		acq(y)
5.		w(x)
1.	w(x)	
6.		rel(y)
2.	acq(y)	
3.	rel(y)	

Table 2: Trace 1 reordered

The trace 2 shows the two events $1\#w(x)_1$ and $2\#w(x)_5$ are now right next to each other and hence they are part of a data race. This only became apparent after reordering the original trace 1 which shows that data races may be hard to detect in some cases so tools are needed to detect them.

3 Background

3.1 Dynamic Data Race Prediction

The method of reordering traces to try to predict data races is called dynamic data race prediction, see **sulzmann2** for more details, and will be discussed in this work. The approach of finding all reorderings is used in exhaustive predictive methods. Since modern programs may utilize multiple threads each executing large blocks of code, finding all possible reorderings might not always be feasible. Therefore effective predictive methods are employed which compromise completeness and soundness for better coverage. FastTrack and TSan are examples for dynamic data race prediction using effective predictive methods. Sulzmann and Stadtmüller **sulzmann** define completeness and soundness as follows:

> Sound means that races reported by the algorithm can be observed via some appropriate reordering of the trace. If unsound, we refer to wrongly a classified race as a *false positive*. Complete means that all valid reorderings that exhibit some race can be predicted by the algorithm. If incomplete, we refer to any not reported race as a *false negative*.

3.2 Data Race

Sulzmann and Stadtmüller **sulzmann** define a data race as follows:

> Let e, f be two read/write events on the same variable where at least one of them is a write event and both events result from different threads. Then, we say that e and f are two *conflicting events*. [...] [I]f two conflicting events can appear right next to each other [,then s]uch a situation represents a *data race*.

An example of a data race can be seen in the trace 2.

3.3 Happens-Before Relation

As stated in **sulzmann2**, the happens-before relation is used to partially order the events in a system. The happens-before relation is defined as a strict partial order so the ordering is transitive but irreflexive. Events can either be ordered and therefore a happens-before relation can be established between them, or they can not be ordered in which case both events 'are said to be *concurrent*' **lamport**. In FastTrack and TSan Lamport's happens-before relation **lamport**, referred to as HB relation, is used. It additionally imposes the program order condition and critical section order condition. The program order condition defines that two events e and f belonging to the same thread, with e appearing before f in the trace, then $e <^{HB} f$ applies. $<^{HB}$ denotes the HB ordering relation. The critical section order condition imposes that for two acquire and release events $acq(y)$ and $rel(y)$, with $rel(y)$ appearing before $acq(y)$ in the trace, then $rel(y) <^{HB} acq(y)$ applies.

3.4 Vector-Clock

3.5 Epoch

4 FastTrack + TSan

```c
#include <pthread.h>

int var;

void *Thread1(void *x) {
    // w(x)1
    var++;
    return NULL;
}

void *Thread2(void *x) {
    // w(x)2
    var--;
    // w(x)3
    // this write will not be reported as part of a data race
    var++;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    pthread_create(&t[1], NULL, Thread2, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}
```

Listing 2: Program demonstrating the incompleteness of TSan

As seen in **marthur** Lamport's happens-before relation is also unsound. Because FastTrack and TSan both use the HB relation they might produce false positives as a result. The following program 3 shows an example in which TSan produces a false negative:

```c
#include <pthread.h>

int var1;
int var2;

void *Thread1(void *x) {
    // r(x)1
    // w(y)2
    var2 = var1 + 5;
    return NULL;
}

```

```c
void *Thread2(void *x) {
    // r(y)3
    if (var2 == 5)
        // w(x)4
        var1 = 10;
    else
        while (true);
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    pthread_create(&t[1], NULL, Thread2, NULL);
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}
```

Listing 3: Program demonstrating the unsoundness of TSan

The program 3 may produce the following trace 3:

	1#	2#
1.	r(x)	
2.	w(y)	
3.		r(y)
4.		w(x)

Table 3: Possible trace of listing 3

5 Conclusion

List of Figures

List of Tables

List of Listings