Hochschule Karlsruhe
University of
Applied Sciences

HKA

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
Betreuer:	Prof. Martin Sulzmann
Bearbeitet am:	June 12, 2023

Contents

1 Introduction

Nowadays concurrent programs are very common in order to make use of 'hyper-threading and multi-core architectures'**SWB-0643851**. 'Due to the highly non-deterministic behavior of concurrent programs' [1, p. 1] data races may arise but can also be hard to find, as they also 'may only arise under a specific schedule' [1, p. 1]. This will be elaborated in the following section. We specifically look at the dynamic data race prediction algorithms of FastTrack and ThreadSanitizer (TSan) V2. TSan V2 uses a slightly modified version of the FastTrack algorithm and differentiates itself from the first version by utilizing happens-before methods instead of the lockset method.
Firstly the motivation and background for dynamic data race prediction tools will be shown. Then the FastTrack and TSan algorithms will be demonstrated along with examples showing the limitations of FastTrack and TSan. In the following the same

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 [2] but rewritten in the programming language C++. The following example shows a program, which exhibits a data race (see chapter 3):

```
1  #include "pthread.h"
2
3  int var1;
4  pthread_mutex_t var2;
5
6  void *Thread1(void *x) {
7      // w(x)1
8      var1++;
9      // acq(y)2
10     pthread_mutex_lock(&var2);
11     // rel(y)3
12     pthread_mutex_unlock(&var2);
13     return NULL;
14 }
15
16 void *Thread2(void *x) {
17     // acq(y)4
18     pthread_mutex_lock(&var2);
19     // w(x)5
20     var1--;
21     // rel(y)6
22     pthread_mutex_unlock(&var2);
23     return NULL;
24 }
25
26 int main() {
27     pthread_t t[2];
28     pthread_create(&t[0], NULL, Thread1, NULL);
29     pthread_create(&t[1], NULL, Thread2, NULL);
30     pthread_join(t[0], NULL);
31     pthread_join(t[1], NULL);
32 }
```

Listing 1: Program containing conflicting events

Generally the corresponding event in the resulting trace is seen above each operation as a comment in the code. For example acquiring and releasing a lock is accomplished by the

`pthread_mutex_lock()`- and the `pthread_mutex_unlock()`-method, which is used in line 10, where a lock on the variable `var2` is acquired. The corresponding event in the trace would be $1\#acq(y)_2$ which is indicated by the method name and the comment above in line 9. The method name shows which thread the event is from and the comment specifies the event. As a write event a variable can simply be incremented or assigned a value. A simple read event could be printing a variable in the console as it accesses the value of the variable. 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: Obtained 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 next to each other in the trace, thus they are part of a data race. This also shows that there are data races that can only be detected under a specific schedule. Data race prediction tools use various methods to try to predict such cases.

3 Background

3.1 Dynamic Data Race Prediction

Race prediction can be done statically or dynamically. Static race prediction uses static analysis, meaning it tries to find data races before running the code. For more details on static race prediction see [3].
Dynamic data race prediction tools run a program and then, based on the obtained trace, try to predict if there may be a reordered trace that exhibits data races, see [2, p. 2].
The approach of finding all possible reorderings is used in exhaustive predictive methods. Since modern programs may utilize multiple threads each executing large, computationally intensive 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 [1, p. 2] 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 Happens-Before Relation and Data Race

As stated in [2, p. 4], 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*' [4, p. 2]. Further two events '[...]*conflict* if they both access (read or write) the same variable, and at least one of the operations is a write. Using this terminology, a trace has a race condition if it has two concurrent conflicting accesses.' [5, p. 2]. An example of a data race can be seen in the trace 2.
In FastTrack and TSan Lamport's happens-before relation [4], referred to as HB relation, is used. The HB relation additionally to the above 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 $rel(y)$ on the same lock y where both events result from different threads with $rel(y)$ appearing before $acq(y)$ in the trace, then $rel(y) <^{HB} acq(y)$ applies [2, p. 4].

3.3 Vector-Clock

As stated by Flanagan and Freund [5, p. 1]:

> Typically, the happens-before relation is represented using vector clocks (VCs) [...]. Vector clocks are expensive, however, because they record information about each thread in a system. Thus, if the target program has n threads, then each VC requires O(n) storage space and each VC operation requires O(n) time. Motivated in part by the performance limitations of vector clocks, a variety of alternative imprecise race detectors have been developed, which may provide better coverage but can report false alarms on race-free programs.

Sulzmann and Stadtmüller define vector clocks as follows [1, p. 7]:

> A vector clock V is a list of time stamps of the following form.
>
> $$v ::= [i_1, ..., i_n] \tag{1}$$
>
> We assume vector clocks are of a fixed size n. Time stamps are natural numbers and each time stamp position j corresponds to the thread with identifier j. We define
>
> $$[i_1, ..., i_n] \sqcup [j_i, ..., j_n] = [\max(i_1, j_1), ..., \max(i_n, j_n)] \tag{2}$$
>
> to synchronize two vector clocks by building the point-wise maximum. We write $V[j]$ to access the time stamp at position j. We write $\text{inc}(V, j)$ as a short-hand for incrementing the vector clock V at position j by one. We define vector clock V_1 to be smaller than vector clock V_2, written $V_1 < V_2$, if (1) for each thread i, i's time stamp in V_1 is smaller or equal compared to i's time stamp in V_2, and (2) there exists a thread i where i's time stamp in V_1 is strictly smaller compared to i's time stamp in V_2. [...] For each thread i we maintain a vector clock $Th(i)$. [...]Initially, for each vector clock $Th(i)$ all time stamps are set to 0 but position i where the time stamp is set to 1.

Sulzmann and Stadtmüller further state that for two events e and f with vector clocks V_1 assigned to e and V_2 assigned to f, if $V_1 < V_2$ then it can be said that $e <^{HB} f$. Further for synchronization the following applies. $V_1 = V_2 \sqcup V_3$ then $V_1 \leq V_2$ and $V_1 \leq V_3$.
For the cases where neither $V_1 < V_2$ nor $V_2 < V_1$ is true it can be said that V_1 and V_2 are incomparable [2, p. 8]. The events assigned with V_1 and V_2 are therefore concurrent. If at least one of the events is a write event, then a data race has been found.
With Lamport's HB relation for events in the same thread j with the respective vector clock V_n, the operation $\text{inc}(V_n, j)$ is applied after processing the event. This is to enforce the program order condition. That results in events having a pre vector clock pre_V and a post vector clock post_V. As an example in trace 1, $1\#w(x)_1$ will have the pre vector clock $[1, 0]$, as we are in the first thread we must initialize set the time stamp at position one to

zero. Then inc(pre_$V_1, 1$) is applied so the post vector clock would be post_$V_1 = [2, 0]$. For following events in the same thread the pre vector clock is the same as the post vector clock of the previous event. So for $2\#acq(y)_4$ the pre vector clock would be $[2, 0]$ and the post vector clock $[3, 0]$. To enforce the critical section order condition the pre vector clock of an acquire event following a release event on the same lock has to be synchronized. The post vector clock of the acquire event will be the incremented synchronized vector clock. In trace 1 the pre vector clock of $1\#rel(y)_3$ is $[3, 0]$ and $[0, 1]$ for $2\#acq(y)_4$. So the synchronized vector clock is $V_2 = [3, 1]$ and after applying inc($V_2, 2$), post_V_2 for $2\#acq(y)_4$ would be $[3, 2]$. For further details on vector clocks, see [2, p. 8].

3.4 Epoch

Flanagan and Freund [5, p. 2] state that vector clock-based race detectors record clock of the most recent write to each variable x by each thread t. As all writes to x are totally ordered (assuming no races detected so far), meaning that for all $w(x)_n$ events the HB relation $V_1 < V_2 < ... < V_n$ can be established. Therefore recording information only about the very last write to x, specifically the clock and the thread identifier of that write, results in no loss of precision while needing less space. This pair of a clock and thread identifier is referred as epoch. Epochs are denoted as $j\#k$ with the thread identifier j and the time stamp k.

As vector clocks record the time stamps for all n threads they need $O(n)$ space while epochs need $O(1)$ space. However '[m]aintaining a growing and shrinking set of epochs can be costly. This applies for cases where many elements are added but also removed.'[2, p. 10]. Instead adaptive approaches can be utilized which makes switching between epochs and vector clocks depending on the situation is possible.

4 FastTrack and TSan

From [6, p. 183]: 'ThreadSanitizer (Tsan)2 is a runtime data race detector developed by Google. ThreadSanitizer is now part of the LLVM and GCC compilers to enable data race detection for C++ and Go code.'
TSan uses a modified version of the FastTrack algorithm which is epoch-based and used for dynamic race prediction. Compared to FastTrack TSan uses shadow memory [6, p. 183][7, p. 4]
FastTrack and TSan both implement a semi-adaptive approach for storing information [2, p. 10]. That means initially only epochs are used. Once a read event on a variable x is recorded, the epoch is switched to a vector clock and the vector clock is maintained for subsequent reads on the variable x. The assumption is that concurrent reads are very frequent. Write events are always represented as epochs under the assumption that all writes are totally ordered.
TSan also uses shadow memory 'to keep track of accesses to all [memory] location[s].'[8, p. 4]. The memory location x is the location of the space where the variable x is stored. Lidbury further states:

> This [shadow memory] will store up to four shadow words per location. For a given location this allows tsan to detect data races involving one of up to four previous accesses to the location. On each access to the location, all the shadow words are checked for race conditions, after which details of the current access are tracked using a shadow word, with a previous access being evicted pseudo-randomly if four accesses are already being tracked. Older accesses have a higher probability of being evicted. As only four of the accesses are stored, there is a chance for false negatives, as shadow words that could still be used can be evicted.

The resulting limitation will be demonstrated in chapter 4.3.

4.1 Limitations in completeness

While FastTrack and TSan are sound, at least for the first found race (as explained in the following chapter), they are both incomplete, as the example in chapter 1 shows. Analyzing the exact program 1 using TSan, it reports that no race is found, while the program does contain a data race as the reordered trace 2 shows. However switching lines 28 and 29 in program 1 to more closely represent trace 2 is sufficient for TSan to correctly detect the data race after running it again on the modified program.
The assumption that all writes are totally ordered is not always the case as the following example 2, taken from [2, p. 10], shows.

```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

The following trace 3 can be obtained from program 2:

	1#	2#
1.	w(x)	
2.		w(x)
3.		w(x)

Table 3: Trace 1 reordered

This produces the following vector clocks:

- $V_1(1\#w(x)_1) = [1, 0]$
- $V_2(2\#w(x)_2) = [0, 1]$
- $V_3(2\#w(x)_3) = [0, 2]$

From this we can see that event $1\#w(x)_1$ is in a data race with both the events $2\#w(x)_2$ and $2\#w(x)_3$. However TSan only reports the first race with the event $2\#w(x)_2$.

4.2 Limitations in soundness

Furthermore, as seen in [9, p. 2], even though Lamport's happens-before relation is sound for the first detected data race, it may produce false positives. Because FastTrack and TSan both use the HB relation they may also produce false positives as a result. The following program 3 shows an example in which TSan produces a false positive.

```c
#include <pthread.h>

int var1;
int var2;

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

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 produces the following trace 5:

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

Table 4: Obtained trace of listing 3

From the trace 5 it is apparent that $1\#w(y)_2$ and $2\#r(y)_3$ are in a data race. A look at the resulting vector clocks for $1\#r(x)_1$ and $2\#w(x)_4$ shows that both events are unordered and thus also in a data race according to the HB relation. However a valid reordering of trace 5 does not exist where $1\#r(x)_1$ and $2\#w(x)_4$ appear next to each other. Therefore the reported data race is a false positive.

4.3 Shadow word length limit

As TSan has a limit on how many shadow words per memory location is stored, the following program 4 will highlight how data races may be missed due to this limit.

```
1  #include "pthread.h"
2  #include "iostream"
3  using namespace std;
4
5  int var1;
6  int threadcount = 6;
7  pthread_mutex_t m;
8
9  void *Thread1(void *x) {
10     var1=1;
11     pthread_mutex_lock(&m);
12     pthread_mutex_unlock(&m);
13     return NULL;
14 }
15
16 void *ThreadN(void *x) {
17     pthread_mutex_lock(&m);
18     pthread_mutex_unlock(&m);
19     cout<<var1;
20     return NULL;
21 }
22
23 void *Thread6(void *x) {
24     pthread_mutex_lock(&m);
25     pthread_mutex_unlock(&m);
```

```
26      var1=1;
27      return NULL;
28  }
29
30  int main() {
31      pthread_t t[threadcount];
32      for (int i = 0; i < threadcount; i++) {
33          if (i == 0) {
34              pthread_create(&t[i], NULL, Thread1, NULL);
35          } else if (i == threadcount) {
36              pthread_create(&t[i], NULL, Thread6, NULL);
37          } else {
38              pthread_create(&t[i], NULL, ThreadN, NULL);
39          }
40      }
41
42      for (pthread_t& thread : t) {
43          pthread_join(thread, NULL);
44      }
45  }
```

Listing 4: Program showing a missed data race due to the limit on shadow word length

The acquire and release operations in lines 11, 12, 17 and 18 are required to ensure that the first write event and subsequent read events are not in a data race. The read events however are still concurrent as they cannot be ordered under the HB relation. The lines 24 and 25 exist to prevent TSan from detecting a data race between the first write event and the last one.

This can be observed in the following trace which can be obtained from the program 4:

	1#	2#	3#	4#	5#	6#
1.	w(x)					
2.	acq(y)					
3.	rel(y)					
4.		acq(y)				
5.		rel(y)				
6.		r(x)				
⋮						
16.						acq(y)
17.						rel(y)
18.						w(x)

Table 5: Obtained trace of listing 4

TSan will report a data race on the events 5# and 6# instead of 2# which means that the first data race was missed.

4.4 Thread count

TSan fails to show stack trace for data races that are at least 16 threads apart from each other

```
#include "pthread.h"
#include "iostream"
using namespace std;

int var1;
int threadCount = 18;
pthread_mutex_t m;

void *Thread1(void *x) {
    var1++;
    pthread_mutex_lock(&m);
    pthread_mutex_unlock(&m);
    return NULL;
}

void *ThreadN(void *x) {
    pthread_mutex_lock(&m);
    pthread_mutex_unlock(&m);
    cout<<var1;
    return NULL;
}

void *Thread18(void *x) {
    var1--;
    return NULL;
}

int main() {
    pthread_t t[threadCount];
    for (int i = 0; i < threadCount; i++) {
        if (i == 0) {
            pthread_create(&t[i], NULL, Thread1, NULL);
        } else if (i == threadCount - 1) {
            pthread_create(&t[i], NULL, Thread18, NULL);
        } else {
            pthread_create(&t[i], NULL, ThreadN, NULL);
        }
    }

```

```
40      for (pthread_t& thread : t) {
41          pthread_join(thread, NULL);
42      }
43  }
```

Listing 5: Program showing thread count limitation

This program starts thread 1 with a write event and 16 threads with non-conflicting read events in between another write event. The two write events are in a data race as the are no lock events for the last write event. TSan will detect the data race but will show the following message for the last thread:

```
1  Previous write of size 4 at 0x564180a42160 by thread T1:
2    [failed to restore the stack]
```

If the variable `threadCount` is set to a value greater than 18 TSan will print the same message. Once it is set to a value less than 18, TSan will print the correct stack trace.

5 Conclusion

After analyzing the FastTrack and TSan algorithms it can be concluded that there are neither algorithm is entirely sound nor complete. Both have limitations that make them unable to detect data races in certain situations. However as of this moment there are no scalable data race prediction tools that are completely sound and complete. Keeping that in mind both of these tools produce good results with good performance in regards to memory and time consumption in comparison to other tools as [5][10] show.

List of Literature

[1] M. Sulzmann and K. Stadtmüller, "Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction (extended version)," *CoRR*, vol. abs/2004.06969, 2020. arXiv: 2004.06969. [Online]. Available: https://arxiv.org/abs/2004.06969.

[2] M. Sulzmann, *Dynamic data race prediction*. [Online]. Available: https://sulzmann.github.io/AutonomeSysteme/lec-data-race.html, accessed Jun. 5, 2023.

[3] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," *SIGPLAN Not.*, vol. 41, no. 6, 308–319, 2006, ISSN: 0362-1340. DOI: 10.1145/1133255.1134018. [Online]. Available: https://doi.org/10.1145/1133255.1134018.

[4] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, 558–565, 1978, ISSN: 0001-0782. DOI: 10.1145/359545.359563. [Online]. Available: https://doi.org/10.1145/359545.359563.

[5] C. Flanagan and S. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," vol. 53, Jun. 2009, pp. 121–133. DOI: 10.1145/1542476.1542490.

[6] Lin, Pei-Hung and Liao, Chunhua and Schordan, Markus and Karlin, Ian, "Runtime and Memory Evaluation of Data Race Detection Tools," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 179–196, ISBN: 978-3-030-03421-4.

[7] L. Yu, F. Jin, J. Protze, and V. Sarkar, "Leveraging the Dynamic Program Structure Tree to Detect Data Races in OpenMP Programs," in *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, 2022, pp. 54–62. DOI: 10.1109/Correctness56720.2022.00012.

[8] C. Lidbury and A. F. Donaldson, "Dynamic Race Detection for C++11," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17, Paris, France: Association for Computing Machinery, 2017, 443–457, ISBN: 9781450346603. DOI: 10.1145/3009837.3009857. [Online]. Available: https://doi.org/10.1145/3009837.3009857.

[9] U. Mathur, D. Kini, and M. Viswanathan, "What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018. DOI: 10.1145/3276515. [Online]. Available: https://doi.org/10.1145/3276515.

[10] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, "Array Shadow State Compression for Precise Dynamic Race Detection (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 155–165. DOI: 10.1109/ASE.2015.19.

List of Figures

List of Tables

List of Listings