Interleaving Generation for Data Race and Deadlock Reproduction

Luis M. Carril Walter F. Tichy

Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
{luis.carril,walter.tichy}@kit.edu

Abstract

Concurrency errors, like data races and deadlocks, are difficult to find due to the large number of possible interleavings in a parallel program. Dynamic tools analyze a single observed execution of a program, and even with multiple executions they can not reveal possible errors in other reorderings. This work takes a single program observation and produces a set of alternative orderings of the synchronization primitives that lead to a concurrency error. The new reorderings are enforced under a happens-before detector to discard reorderings that are infeasible or do not produce any error report. We evaluate our approach against multiple repetitions of a state of the art happens-before detector. The results show that through interleaving inference more errors are found and the counterexamples enable easier reproducibility by the developer.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Reliability, Verification

Keywords Data race, deadlock, concurrent programs, debug, replay, CSP

1. Introduction

Concurrent programming is complex, and finding and debugging data races and deadlocks is time consuming. A data race is a simultaneous unordered access by two threads to the same memory location, where at least one performs a write operation. A deadlock is a circular dependency of resources where no thread can make any further progress. Both errors are very complicated to detect, especially due to the non-deterministic nature of parallel programs.

For example, the program in Listing 1 is a simplification of a common problem in race detection. The main thread (t_1) executes the $main$ function, it will spawn a new thread (t_2), which executes the $task$ function in parallel. The program uses two shared variables x and y, which both threads are updating concurrently. The expected final value after execution should be $x = 2$ and $y = 3$, but depending on the executed interleaving a race on y can happen producing the unexpected result of $y = 2$.

```
y = 0; x= 0              1
                         2
main() {                 3
  y++                    4
  t1 = fork(task)        5
  y++                    6
  lock(m)                7
  x++                    8
  unlock(m)              9
  join(t1)               10
}                        11
                         12
task() {                 13
  lock(m)                14
  x++                    15
  unlock(m)              16
  y++                    17
}                        18
```

Listing 1. Racy program

Depending on the scheduling, the race will either be detected by a dynamic happens-before based tool (based on the Lamports relation [12]) or not. If the program executes line 7 before line 14, then the main thread locks the mutex, and the unlocking at 9 followed by the corresponding lock at 14 creates a happens-before order between the two instructions hiding the possible data race between 6 and 17. Only if line

SEPS '15, October 25-30, 2015, Pittsburgh, PA, USA.
Copyright © 2015 ACM ISBN 978-1-4503-3910-0/15/10...$15.00.
http://dx.doi.org/10.1145/2837476.2837480

14 is executed before line 7 will a happens-before tool detect the possible race, as both accesses to y will not be ordered.

This timing dependency means that the developer must re-execute the application multiple times with the hope of catching different interleavings and see if there is any kind of error. And in the case that in one repetition a report with a race warning is found, nothing ensures that the developer will be able to reproduce the erroneous interleaving. Even more complicated is the case when the developer wants to attach a debugger and observe the trace, as the debugger, breakpoints and user interaction will influence the scheduling of the program under test. One solution is to record each re-execution of the program under the race detector, and discard all recordings that do not produce any error report; but this does not ensure to observe any interleaving that leads to a possible error, the same could be observed the whole time.

This work presents an approach that, after a first trace step, generates a set of suspicious interleavings, which could lead to a concurrency error. The trace is modeled with specific semantic models for synchronization instructions and searched for data race and deadlock patterns. The candidate schedulings -counterexamples- are enforced in the program, and discarded if they do not produce any error.

This work is an extension of our preliminary CSP modeling of traces [3]. Here we support additional synchronization constructs like signal-wait and barriers, introduce explicit confirmation through reproduction and a more complete evaluation with 11 real-world programs.

2. Interleavings generation

The core of this approach consists of collecting a trace with the minimal amount of memory and synchronization events. The trace is the recording of events of a single program execution under a concrete input. Then we build a predictive model based on the trace and search it for data races and deadlocks. The model generates a set of alternative synchronization reorderings that could lead to an error. We replay each error sequence to prune false positives and provide localization information to the user.

2.1 Trace model

We define a trace as the sequence of performed events observed in a single execution of the multithreaded program since the beginning of its execution. The events are the following:

- start(t): first event of thread t
- end(t): last event of thread t
- fork(t,t'): thread t spawns a child thread t'
- join(t,t'): thread t blocks until t' ends its execution
- read(t,o): thread t reads object o
- write(t,o): thread t writes object o
- lock(t,m) : thread t acquires mutex m
- unlock(t,m) : thread t releases mutex m
- signal(t,c$_i$): thread t wakes a thread waiting on condition instance c$_i$
- broadcast(t,c$_i$): thread t wakes all threads waiting on condition instance c$_i$
- wait(t,c$_i$) : thread blocks until another thread executes a signal or broadcast on condition instance c$_i$
- barrier_enter(t,b$_i$): thread t blocks at the barrier instance b$_i$
- barrier_exit(t,b$_i$): thread t leaves the barrier instance b$_i$

An object is an identifier for a contiguous memory region. An object is shared if two or more different threads realize a $read$ or $write$ operation on the element, and at least one is a write; we ignore the $read$-only shared objects as races cannot happen there. The $read$ and $write$ events are memory operations, the rest are synchronization operations. A thread segment is the set of consecutive events of a particular thread between two synchronization events, with no other synchronization events in the middle. A condition instance c$_i$ and barrier instance b$_i$, represent each unique use of a condition variable or barrier; i.e. a condition variable (for a signal-wait synchronization) can be reused multiple times, each time in the model is treated as a single instance and differentiated from the others.

Finally, a race-pair candidate is any pair of events from different threads where at least one is a write on the same shared object.

2.2 Modeling and detection

The captured trace is modeled using the Communicating Sequential Processes [8] (CSP) process algebra. We extended our preliminary CSP modeling of traces [3] to support additional synchronization constructs like signal-wait and barriers. The resulting encoded formula represents the captured interleavings as well as alternative ones. In CSP, independent processes (in upper case) execute and communicate synchronously through events (in lower case). Processes can be composed into more complex processes through a set of formal described operators, i.e: alternatives, interruptions, synchronized parallel execution or interleaving (non-synchronized parallel execution).

Figure 1 represents the corresponding model for a trace of the program at Listing 1. Each thread from the original trace is modeled independently as a single CSP process: $THREAD_1$ and $THREAD_2$. The events are mapped one-to-one from the trace to CSP events. Each CSP event carries, separated by dots, a thread identifier and the identifier of the action target. Each process defines a total order of the events it composes. All thread processes are combined using the $|||$ interleaving operator, this mixes all their events freely. The combination has no total order, but the partial orders of the composing processes are respected. To further restrict the

$$
\begin{aligned}
THREAD_1 =& \mathit{fork}.t_1.t_2 \rightarrow \mathit{write}.t_1.y \rightarrow \mathit{lock}.t_1.m \rightarrow \mathit{write}.t_1.x \rightarrow \mathit{unlock}.t_1.m \rightarrow \mathit{join}.t_1.t_2 \rightarrow \mathit{SKIP}\\
THREAD_2 =& \mathit{start}.t_2 \rightarrow \mathit{lock}.t_2.m \rightarrow \mathit{write}.t_2.x \rightarrow \mathit{unlock}.t_2.m \rightarrow \mathit{write}.t_2.y \rightarrow \mathit{end}.t_2 \rightarrow \mathit{SKIP}\\
MUTEX_m =& \mathit{lock}.t_1.m \rightarrow \mathit{unlock}.t_1.m \rightarrow MUTEX_m\\
& \Box\ \mathit{lock}.t_2.m \rightarrow \mathit{unlock}.t_2.m \rightarrow MUTEX_m\\
FORK_2 =& \mathit{fork}.t_1.t_2 \rightarrow \mathit{start}.t_2 \rightarrow STOP\\
JOIN_2 =& \mathit{end}.t_1.t_2 \rightarrow \mathit{join}.t_1.t_2 \rightarrow STOP\\
PROGRAM =& (THREAD_1 \mid\mid\mid THREAD_2) \underset{\{|lock, unlock, fork, start, join, end|\}}{\parallel} (MUTEX_m \mid\mid\mid FORK_2 \mid\mid\mid JOIN_2)
\end{aligned}
$$

Figure 1: CSP model

resulting process, we add synchronization processes for each synchronization construct found in the trace, so it does not contain infeasible event orderings due to the synchronization restrictions.

Each synchronization mechanism is represented with a process. In the example, the $MUTEX$ process instantiated as $MUTEX_m$, represents the only mutex m in the trace. The $MUTEX_m$ process allows any thread to execute the $lock$ event if the process is at the beginning (using alternative operator $\Box$), or only the $unlock$ event to the thread that has executed the corresponding $lock$ event. Afterwards the $MUTEX$ process returns to the initial state. Similarly the $FORK$ and $JOIN$ processes represents the thread creation and joining respectively. All synchronization mechanisms are combined with the $\mid\mid\mid$ operator as they are disjoint. These processes are combined with the interleaved threads with the synchronized parallel operator $\underset{X}{\parallel}$, forming the process $PROGRAM$. This operator specifies that each time an event on set X is performed in one of the two sides of the operator, it must be ratified (executed simultaneously too) by the other side. The behavior of the process $PROGRAM$ contains all possible interleavings of both threads' events with the exception of simultaneous access to the lock protected events.

Other synchronization primitives appear in Figure 2. For condition variables and barriers a process is created for each instantiation of the condition variable or barrier, this is a difference from a mutex, where a single process is enough.

Errors in the model are discovered by defining them as patterns and applying a refinement relationship to the CSP model of the generalized trace. A refinement relationship ($S \sqsubseteq I$) holds if the $behavior$ of the process I is a subset of the $behavior$ of process S. CSP supports multiple definitions -semantic models- of the $behavior$ of a process. For each type of error we define a specific pattern process and a refinement relationship under a semantic model. When the relationship does not hold, the resulting counterexample is the sequence of events in the model that lead to the erroneous state.

2.2.1 Race detection

The processes $RACE$ and $RACE\_ERR$ in Figure 3 model all possible race event combinations. The process $RACE$ performs the $race$ event when any combination of $read$ and $write$ events (except $read - read$) happens without any other event in between. The process $RACE$ can derive again into process $RACE$ through the interrupt operator $\triangle$, this disallow that the process blocks itself when confronted with any other event, effectively reseting the pattern.

In the CSP trace model $\mathcal{T}$ describes the $behavior$ of a process by only the different set of sequences of events the process produces. A race in process $PROGRAM$ on object p is found if the relationship in Equation 3 in Figure 3 does not hold.

The $RACE$ process follows all memory and synchronization operations of the process $PROGRAM$. And all events that this parallel composition produces are hidden, with the exception of the $race$ event, by the hiding operator: $\setminus\ \Sigma - \{race\}$. By definition, the $STOP$ process has an empty $behavior$ (does not produce any trace); if the parallel composition shows the $race$ event then the relationship does not hold, and a data race on p exists.

Then the sequence of events performed by the process $PROGRAM$ alone is extracted. We only keep the synchronization operations in this sequence, which define the scheduling needed to reach a state where two thread segments are in a race situation. For the given example if we instantiate the refinement of Equation 3 with variable y as the parameter for process $RACE$ the refinement does not hold and we obtain the scheduling:

$$\mathit{fork}.t_1.t_2, \mathit{start}.t_2, \mathit{lock}.t_2.m, \mathit{unlock}.t_2.m$$

It is noteworthy that we do not check each race-pair, but instead we check if there exists a data race on the target shared object. If there is more than one race-pair for object p, a single refinement check for p is done; reducing the number of model checks from one per race-pair to one per shared object. When the refinement does not hold, there is at least one counterexample identifying the first race. If there are

$$\begin{aligned} BARRIER_{b_0} &= barrier\_enter.t_1.b_0 \to barrier\_enter.t_2.b_0 \to barrier\_exit.t_1.b_0 \to barrier\_exit.t_1.b_0 \to STOP \\ SIGNAL_{c_0} &= signal.t_1.c_0 \to wait.t_1.c_0 \to STOP \end{aligned}$$

Figure 2: Other synchronization constructs

$$\begin{aligned} RACE(v) &= RACE\_ERR(v) \triangle \\ &\quad (\Box\, x : \{|read, write, lock, unlock, fork, start, join, end|\} @ x \to RACE(v)) \end{aligned} \tag{1}$$

$$\begin{aligned} RACE\_ERR(v) &= read?t_a!v \to write?t_b : Threads \setminus \{t_a\}!v \to race \to SKIP \\ &\quad \Box\, write?t_a!v \to read?t_b : Threads \setminus \{t_a\}!v \to race \to SKIP \\ &\quad \Box\, write?t_a!v \to write?t_b : Threads \setminus \{t_a\}!v \to race \to SKIP \end{aligned} \tag{2}$$

$$STOP \sqsubseteq_{\mathrm{T}} (PROGRAM \underset{\{|read, write|\} \cup \{|lock, unlock, fork, start, join, end|\}}{\|} RACE(p)) \setminus \Sigma - \{race\} \tag{3}$$

Figure 3: Race pattern process and refinement

multiple counterexamples, then there exists more than one state in the model (with its associate trace) that leads to a race situation. The number of counterexamples extracted can be bounded during the model checking process to reduce the time cost, at the expense of having false negatives on the same variable but in different locations and interleaving.

When a trace contains multiple events from a repeating code section, some race-pairs are redundant. For example if the events of function $task$, executed by thread t_2 in Listing 1 where inside a loop with N iterations, for variable y instead of a single race-pair candidate we will have N race-pair candidates: one for each combination between thread t_1 and each access from t_2 in each loop iteration. The same situation happens if instead of a loop there were N instances of the thread t_2 not a single one. In these cases it is enough to detect a single race-pair. If the counterexample generation is bounded to one, at least one race for each variable will be found; and all other redundant race-pairs are ignored. After fixing the program, new checks can reveal new races but they will affect different code localizations.

2.2.2 Deadlock detection

CSP contains support for deadlock detection through the failure model $\mathcal{F}$. This model is an extension of the trace model $\mathcal{T}$ where to each specific observable trace a set of $refusals$ is added. Each $refusals$ is a set of events in the system that the process cannot execute.

The following refinement holds for a non-deadlocking process $PROGRAM$:

$$\begin{aligned} &LIVE \sqsubseteq_{\mathrm{F}} (PROGRAM \setminus \Sigma\,;LIVE) \\ &LIVE = live \to LIVE \end{aligned}$$

The specification process is the recursive process $LIVE$, which always has the event $live$ available. If $PROGRAM$ ends successfully then it will behave like $LIVE$, but if not it will never perform the $live$ event, and the refinement will not hold. Additionally all events from $PROGRAM$ are hidden because none of them appear in $LIVE$. If the refinement does not hold, there is a sequence of events that lead to a state where the process will deadlock.

Similar to the race case, if a deadlock is found, we extract the events performed by the process $PROGRAM$ alone, and this sequence will lead to a situation where the original program deadlocks.

2.3 Replay, confirmation and localization

Each counterexample describes the order of synchronization operations that lead to a data race or deadlock. The program is re-executed with the same input but the synchronization operations are blocked until the previous operations defined in the counterexample have been executed. The counterexample only defines the scheduling until it reaches the erroneous thread segments, so the replay is only deterministic until that point; afterwards the program runs non-deterministically. The use of coarse-grained enforcing allows a certain degree of parallelism between threads and does not impose a speed penalty like a complete serialization of the execution or a fine-grained replay based on memory accesses.

Replaying plays two additional roles outside of enabling consistent debugging by the user: confirmation and error localization. The detection can generate false positives because the modeling does not consider that the control flow has diverged due to a reorder of thread segments. Under the new order, the program can take a different branch and never reach the erroneous situation. Or the program is unable to execute the whole counterexample, because the new order does not exist in the program. Also, the tracing step does not collect information about how the specific program instructions map to the source code; this reduces the size of the trace and tracing speed penalty. But the concrete source code lines involved in the specific error are unknown.

This two issues are solved by replaying the program with the counterexamples and executing in parallel a dynamic online happens-before detector. If the happens-before detector does not detect any race nor is the program deadlocked then the counterexample is discarded, otherwise it leads to a concurrency error. If the counterexample cannot be enforced, then it is dropped. The use of a happens-before detector ensures that no false positives are produced during replaying. The localization of the error is provided by the happens-before detector in the case of a data race, and can be observed with a standard debugger in the case of a deadlock.

3. Implementation

We target multithreaded C programs using the pthread library [1]. The tracing and replayer are modifications of ThreadSanitizer [18], the input program is compiled using the LLVM framework, statically instrumented and linked against an auxiliary runtime library.

The number of read and write events performed by a program can be huge, so we apply several strategies to reduce the number of logged events. During instrumentation, an escape analysis removes local operations (scope local variables). Inside the same thread segment, it is enough to trace a single access to a specific address (and it is a write), because if this operation races, similar operations in the same thread segment are going to produce a data race too. During tracing, each thread uses a thread local cache for the current thread segment, that stores the last access to each variable and the events; the cache is emitted to the trace at the end of the thread segment (a new synchronization operation). Remaining redundant events are removed later offline.

Then each thread trace is collected independently, as no synchronization is done between threads; which contains a single execution of the whole program. This file is processed offline afterwards: to remove remaining redundant memory operations in the same thread segment, match signal-wait events and compute the shared objects. The generated trace follows the format presented in Section 2.1. As described, the format assumes that each object is a unique memory region. To apply a real program trace to this model, we would need to consider each byte as a different shared object and then apply a CSP check for each object (i.e. each byte). Instead, for each read and write, the starting address and operation size is logged; and in this step each shared region (composed of consecutive shared bytes) is computed and replaced by an unique identifier. Each identifier represents a single shared object in the final trace.

Also, in Section 2.1, the operations on condition variables and barriers are defined for each unique use of the synchronization construct. For barriers, it is enough to identify them and log the enter and exit on the primitive. But in the signal-wait synchronization case, the wait primitive is not always observed: if the waiting precondition is met before reaching the corresponding wait-loop (as recommended by the pthread library), the $pthread\_cond\_wait$ primitive is never executed. This makes creating the signal-wait relationship for individual uses of the condition variable difficult, as there can be a logical signal-wait order between two threads, but without the appearance of the wait event. To solve this issue, we implemented an additional instrumentation step, that finds the wait-loops and instruments them after and before. This additional instrumentation allows the postprocessing to match the wait-loop with the signal independently if the wait primitive is observed or not.

For data races, we make a first and fast filter of race-pairs. We use a hybrid algorithm to prune non racy memory accesses without generating false negatives. This will reduce the number and the complexity of the the more expensive CSP checks afterwards. It is a two step algorithm, which first executes a weakened happens-before algorithm without mutex induced edges; it removes all race-pairs correctly ordered by the following synchronizations: fork-start, end-join, signal-wait and barriers. Then we use a standard lockset algorithm to remove race-pairs that share at least one lock. The remaining shared variables have at least one race-pair candidate.

We build a CSP model for each remaining shared variable, without the accesses to the other variables, to minimize the number of events and the checking time. Then each shared variable is checked for races using a refinement check as described. In the case of checking for deadlocks a CSP is built directly from the captured trace without any intermediate step as all memory accesses are unnecessary (shared object computation and the hybrid happens before are irrelevant). The model and check are coded in CSP$_M$(machine readable CSP) and passed to the Failures-Divergences Refinement 3 model checker [7].

The replayer is also a modification of ThreadSanitizer, it executes the given program under a concrete synchronization order and checks for races simultaneously. The order is defined by the synchronization events in the counterexample and enforced through a semaphore [4]; all non-synchronization operations are executed freely. ThreadSanitizer executes its normal happens-before race detection simultaneously, but it ignores all happens-before edges due to the replayer internal machinery.

The replayer aborts the execution if the counterexample cannot be enforced due a control flow divergence, so it can be differentiated from a deadlock of the program under test.

Scalability As other trace based techniques, the length of the trace limits the scalability of the approach. A longer trace produces a more complex model, and the model checker needs more time.

The complexity of our model increases only with the number of synchronization operations in the trace, as for each shared variable a different model is built and non-related memory operations are removed. The number of synchronization events is expected to be a fraction of the whole program trace. But this is not enough, a common strategy is to split the trace in windows and process each window individually. We use this approach, splitting the trace in fixed sized windows, and analyzing each one independently for errors. Previous windows are only used to compute the initial state of mutexes and to generate the prefix of the generated schedules. The prefix for a given window is the same interleaving as the observed until this point. The splitting introduces false negatives, e.g.: data races between memory events in different windows are overlooked. This also limits the effect of control flow divergence during replaying as it can only happen inside the analyzed window.

Further reduction in the number of model checks and trace size is possible trough a reduction in the number of instrumented memory accesses. Instead instrumenting the whole program, a set of functions or specific instructions can included or excluded for instrumentation based on user input or output of a static race detector. In any case, all synchronization operations are still tracked.

4. Evaluation

We evaluated our approach in two parts: a set of microbenchmarks representing concrete multithreading orderings and a set of known open source applications with concurrency errors. The evaluation has been done on a Intel Xeon eight core machine with 3.7GHz processor and 64GB RAM. The splitting window size is set to 10000 events executed by the program. No manual or static detector prefiltering for instrumentation has been used. We evaluate the results of our tool against ThreadSanitizer [18] (TSAN), a state of the art dynamic happens-before race detector for C programs.

4.1 Microbenchmarks

The microbenchmarks evaluation, Table 1, is an aggregation of scenarios from multiple sources[18–20]. Scenarios described in papers that are not available have been coded explicitly. The third column is the total number of real errors: races and deadlocks. The fourth column is the number of errors detected by our approach. The fifth column is the number errors detected by TSAN.

Our approach can find and correctly generate a reproducible counterexample for all the microbenchmark programs, finding 100% of the errors. Most of the missed errors (32,3% of the total number of errors) by TSAN are related to critical section commutativity (a specific order of accesses hides the race). Even for a relaxed version of a happens-before detector it is very difficult, if not impossible, to reason about all these scenarios [19].

4.2 Application benchmark

The application benchmark is composed of several applications commonly used in data race benchmarking. Aget is a parallel download http application. BoundedBuffer is a producer-consumer implementation with a limited buffer. Ctrace is a library for debugging and tracing multithreaded programs. Qsort is a parallel implementation of the quick-sort ordering algorithm. Blackscholes, fmm, fft, lu, lu-non, streamcluster and water-nsquared are applications and kernels from the SPLASH-2 multithreaded benchmark [22]. This benchmark compares a single execution of our tool against multiple executions of TSAN.

The results appear in Table 2. The first column is the application name and the second column the aggregated lines of code. The third column is the total number of events traced by the program before offline postprocessing. The fourth column is the final number of events of the postprocessed simplified trace, and the fifth is how many events are synchronization events. The sixth column is the number of suspicious counterexamples emitted by the CSP refinement checking. The seventh column shows how many counterexamples are replay feasible and report at least one error. The eight column is the number of unique localizations (in source code) with errors found by our counterexamples after replaying. Finally, the ninth column shows how many erroneous source code localizations TSAN finds in total after 100 executions.

All the errors detected by both tools are data races, no deadlock is reported by any tool. For most race free programs -blackscholes, boundedBuffer,lu, lu-non and water-nsquared- we see that our tool generate no counterexample candidate. Only in fft a single candidate is generated but later discarded because the reproduction did not produce any warning.

In streamcluster a single valid counterexample is generated for a single error, matching the results of TSAN. Ctrace produces an extra counterexample that turns out useless, but the other three counterexamples reveal an error more as TSAN (who has found only two).

For aget, TSAN found one more data race as our approach, which generates two counterexamples for two confirmed errors. These additional data race is mutually exclusive with another data race, both are in different program paths and they cannot be found a single execution. This data race is found in 10% of the TSAN executions, the other data race appears in the rest 90% of the executions; as our tool examines a single trace it is highly dependent on the path explored by that trace.

Finally we have the qsort and fmm applications. In both cases, our approach finds more erroneous locations than TSAN. In qsort although initially only two counterexamples are generated; each reveal more than one race, for a total of six. This shows that a single counterexample is enough for races on the same thread segment, as the trace is simplified and shared memory regions are treated as a single block. Fmm generates 190 counterexamples, of them 10 are infeasible (they cannot be enforced), the remaining 180 generate different amounts of race reports, from 2 to 40 reports per counterexample. The 180 feasible counterexamples reveal a

Table 1. Microbenchmarks evaluation

Scenarios	LOC	Real errors	Found errors	TSAN errors
48	2274	31	31	21

Table 2. Application evaluation

		Trace			Counterexamples			
Application	LOC	Captured	Reduced	Sync	Candidates	Feasible	Found errors	TSAN-errors
aget	847	1792	291	80	2	2	2	3
blackscholes	327	1568	84	4	0	0	0	0
boundedBuffer	252	829	326	105	0	0	0	0
ctrace	772	1543	275	64	4	3	3	2
fft	701	40440	207	18	1	0	0	0
fmm	3200	1078923	2730	1470	190	180	50	36
lu	732	59857037	1185	138	0	0	0	0
lu-non	533	971559	490	42	0	0	0	0
qsort	511	3305	195	40	2	2	6	1
streamcluster	1443	388	115	6	1	1	1	1
water-nsquared	1188	27560223	9220	8300	0	0	0	0

total of 50 race locations, while TSAN has only found 36 of them. In this case there is a overlapping between counterexample, and the same location is reported multiple times.

The results show that from a single execution our approach can find more races than TSAN in multiple executions. A non-predictive race detector relies on reaching a specific timing to be able to see some races. But our solution dilutes this factor exploring other interleavings offline. Although the errors we can detect are constrained by the explored path of the trace.

Also race detectors usually provide only the localization of the race, but no information on when and how the program has reached that position. Mixing the race detector with interactive debugging can make the erroneous state difficult to reach, because of the probe effect [6]. Our tool provides a step by step counterexample of the synchronization events that can be used to reproduce the observed data race.

The time costs of our approach in comparison with TSAN is vastly outperformed. TSAN being a dynamic detector with a strong focus on performance introduces a typical slowdown of 5x-15x. As our tracing and replay tools are build on top of TSAN, they increase the performance penalty of TSAN, e.g. by 20x for aget. This is due to the hugely increase of output generated (the trace is stored as a readable character based file) and partial serialization of the reproduction (threads are blocked until they reach the corresponding event). But a reproducible version of TSAN would incur in similar costs. Additional costs come from the interleaving generation part: postprocessing and model checking, both together can take from a couple of seconds (in aget) to half an hour (in lu). Although the resource consuming is much bigger, the approach is automatized and the developer only need to pay attention to the feasible generated counterexamples.

5. Related work

Predictive trace analysis techniques, like this work, encode the trace of a single observation and generate reorderings of the events. The reorderings are analyzed for errors providing a witness. The work of Said et al. [16] encodes a trace as a satisfiability problem and uses SMT to explore the reorders. This technique is conservative as it enforces a complete read-write consistency. It logs all the read and write operations including their values, and enforces the dependencies between these reads and writes as constraints. Additional constraints are formed by the semantics of synchronization primitives. Due to the read-write consistency, the technique does not suffer from control flow divergence and does not emit false positives. Huang et al. [9] improves the technique explicitly taking into account branch instructions, so it reduces the amount of control flow dependencies and optimizations to minimize the number of constrains. These two techniques have the advantages of not producing false positives and producing a fine-grained witness. But they require a much more detailed trace, including non-shared accesses and the actual values of memory operations; also if only a selective tracing is desired (find errors in a set of instructions) they need to trace the complete backwards slice to maintain their capabilities. Our approach works with less information, only needs to know which addresses are accessed in each thread segment, enabling selective tracing without additionally costs. We take an empirical approach testing the generated witnesses to discard false positives, instead of offering a sound predictive step.

Most common detectors are dynamic detectors based on happens-before [5, 18](based on the Lamport's relationship [12]). They build a directed graph of the behavior, any two non-ordered shared memory accesses conform a race. It covers a limited set of interleavings and cannot explore alternative reordering of synchronization operations; so multiple runs can produce different detection results due to different observed timings. They do not produce false-positives but can generate false negatives of races in non observed interleavings. Variations of happens-before relax the model (e.g. eliding mutexes) to cover more schedulings in a single run [19]. The lockset algorithm [17] stores the locks used to access each variable, when accesses to the same variable without sharing any lock, a race is reported. Lockset produces false alarms but is partially agnostic to the specific observed interleaving. There are also hybrid approaches, that combine happens-before and lockset, to maintain the precision with a higher coverage [10, 14]. Lockset has been commonly applied to static detectors [21], but tend to generate too many false alarms, rapidly making the developer lose interest in the tool. Static approaches also have the disadvantage of not producing a reproducible test case for the error.

Active testing approaches re-execute multiple times the program under test with the same input; they force the program to execute different reorders. Some use a random scheduler and introduce thread preemptions at different points in the source code [11]. Others use model checking techniques -partial order reduction- to explore systematically multiple interleavings [13][23].

Most of the previous tools also detect if a lock has been reached or the mutex acquisition order is inconsistent along the execution. But inconsistent lock acquisition does not mean that a deadlock is possible, as different orders can be used in different disjoint parts of a program. Most specific deadlock tools are static tools [15] with the false alarms that static tools carry on, or focus on efficient online detection [2].

6. Conclusion

This paper describes a predictive approach to generate alternative interleavings and detect data races and deadlock. We capture a single trace of a multithreaded application and model it in CSP process algebra. The CSP model represents alternative reorderings of the synchronizations events, following the particular semantics of the synchronization constructs. We generate a set of counterexamples applying model checking with generic patterns describing the concurrency errors. These counterexamples are enforced in the program and tested for usefulness along with a happens-before detector.

The results show all of the accepted generated counterexamples enforce some concurrency error, and altogether they find more real errors than a common dynamic detector, up to 44% more erroneous locations. Even more, the counterexamples enable coarse deterministic debugging of the program and allows the developer to examine the behavior of the program live.

Acknowledgments

The authors would like to thank Siemens Corporate Technology for their financial support as well the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

[1] B. L. L. N. L. Barney. POSIX Threads Programming. URL `https://computing.llnl.gov/tutorials/pthreads/`.

[2] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. *Hardware and Software, Verification and ...*, pages 1–17, 2006. URL `http://link.springer.com/chapter/10.1007/11678779_15`.

[3] L. M. Carril and W. F. Tichy. Predicting and witnessing data races using CSP. In *NFM 2015*, page 6, 2015.

[4] R. H. Carver and K. C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 1991. ISSN 07407459. .

[5] C. Flanagan and S. N. S. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. ISBN 9781605583921. . URL `http://doi.acm.org/10.1145/1542476.1542490http://dl.acm.org/citation.cfm?id=1542490`.

[6] J. Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[7] T. Gibson-robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. *Tools and Algorithms for the Construction and Analysis of Systems*, 8413:187–201, 2014.

[8] C. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978. URL `http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf`.

[9] J. Huang, P. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014. ISBN 9781450327848. URL `http://dl.acm.org/citation.cfm?id=2594315`.

[10] A. Jannesari and W. F. Tichy. On-the-fly race detection in multi-threaded programs. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*, PADTAD '08, pages 6:1—-6:10, New York, NY, USA, 2008. ACM. ISBN 9781605580524. . URL `http://doi.acm.org/10.1145/1390841.1390847http://www.cs.umd.edu/~pugh/ISSTA08/padtad2008/papers/a8-jannesari.pdf`.

[11] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV '09)*, pages 675–681, 2009.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. URL `http://dl.acm.org/citation.cfm?id=359563`.

[13] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, Nov. 2007. URL `ftp://ftp.research.microsoft.com/pub/tr/TR-2007-149.pdf`.

[14] E. Pozniansky and A. Schuster. MultiRace: efficient on the fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19 (3):327–340, 2007. . URL `http://onlinelibrary.wiley.com/doi/10.1002/cpe.1064/abstract`.

[15] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.*, 33:3:1–3:55, 2011. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/1889997.1890000$\delimiter"026E30F$nhttp://dl.acm.org/ft_gateway.cfm?id=1890000&type=pdf`.

[16] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NFM'11 Proceedings of the Third international conference on NASA Formal methods*, pages 313–327, 2011. ISBN 0001409107. URL `http://link.springer.com/chapter/10.1007/978-3-642-20398-5_23`.

[17] 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, Nov. 1997. ISSN 0734-2071. . URL `http://doi.acm.org/10.1145/265924.265927http://portal.acm.org/citation.cfm?doid=265924.265927`.

[18] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *WBIA '09 Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009. ISBN 9781605587936. URL `http://dl.acm.org/citation.cfm?id=1791203`.

[19] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '12*, page 387, 2012. . URL `http://dl.acm.org/citation.cfm?doid=2103656.2103702`.

[20] Valgrind. Helgrind: a data-race detector, 2007. URL `http://valgrind.org/docs/manual/hg-manual.html`.

[21] J. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ESEC/FSE '07 Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007. ISBN 9781595938114. URL `http://dl.acm.org/citation.cfm?id=1287654`.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs. In *Proceedings of the 22nd annual international symposium on Computer architecture - ISCA '95*, number June, pages 24–36, New York, New York, USA, 1995. ACM Press. ISBN 0897916980. . URL `http://portal.acm.org/citation.cfm?doid=223982.223990`.

[23] Yang, Yu, Xiaofang Chen and G. Gopalakrishnan. Inspect : A Runtime Model Checker for Multithreaded C Programs. Technical Report i, 2008.