DOI: 10.15514/ISPRAS-2025-37(6)-41

Bounding Thread Switches in Dynamic Analysis of Multithreaded Programs

[1] *V.P. Rudenchik, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>*
[1] *P.S. Andrianov, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>*
[1,2] *V.S. Mutilin, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

Abstract. For detecting race conditions in multithreaded programs, dynamic analysis methods can be used. Dynamic methods are based on observing program behavior on real program executions. Since analyzing all possible execution paths is generally infeasible (due to the combinatorial explosion of possible thread interleavings), dynamic methods can overlook certain bugs that manifest only within the specific conditions or thread interleavings. This limitation applies, for instance, to the approach implemented in the previous version of RaceHunter tool, which demonstrates the ability to effectively detect race conditions, but may still miss certain cases. To address the combinatorial explosion problem, context bounding analysis can be used. Context bounding is a dynamic analysis technique that limits the number of thread switches in each explored execution path, enabling more scalable exploration. This method is able to detect bugs missed by other techniques with the bound of only two preemptive thread switches.

In this work, we present an implementation of context bounding within the RaceHunter tool, which provides a unified framework for describing various dynamic analysis techniques. The evaluation shows that the proposed approach is able to detect race conditions that other methods missed, though at the cost of significantly increased analysis time. As expected, this increase in analysis time is caused by repeated executions. Still, the implementation is an important foundation for future integration with other race detection techniques, specifically with the approach already implemented in the RaceHunter tool.

Keywords: verification; dynamic analysis; context bounding.

For citation: Rudenchik V.P., Andrianov P.S., Mutilin V.S. Bounding Thread Switches in Dynamic Analysis of Multithreaded Programs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 6, part 3, 2025. pp. 133-148 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-41.

Ограничение количества переключений потоков при динамическом анализе многопоточных программ

[1] *В.П. Руденчик, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>*
[1] *П.С. Андрианов, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>*
[1,2] *В.С. Мутилин, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*

Аннотация. Для обнаружения состояний гонки в многопоточных программах могут использоваться методы динамического анализа. Динамические методы основаны на наблюдении за поведением программы при её реальном выполнении. Так как анализ всех возможных путей выполнения в общем случае неосуществим (из-за комбинаторного взрыва числа возможных чередований потоков), динамические методы могут упускать определённые ошибки, проявляющиеся только при специфических условиях или порядке выполнения потоков. Это ограничение относится, например, к подходу, реализованному в предыдущей версии инструмента RaceHunter, который демонстрирует способность эффективно выявлять состояния гонки, но всё же может пропускать отдельные случаи. Для решения проблемы комбинаторного взрыва может использоваться анализ с ограничением числа переключений потоков (англ. context bounding). Этот метод подразумевает исследование только тех путей выполнения, в которых ограничено число переключений потоков, что позволяет сделать анализ более масштабируемым. Анализ с ограничением числа переключений потоков способен выявлять ошибки, пропускаемые другими методами, при ограничении всего в два принудительных переключения потоков.

В данной работе мы представляем реализацию анализа с ограничением количества переключений потоков в инструменте RaceHunter, который обеспечивает единую платформу для описания различных методов динамического анализа. Оценка показала, что предложенный подход способен выявлять состояния гонки, которые были пропущены другими методами, хотя и ценой значительного увеличения времени анализа. Как и ожидалось, это увеличение времени связано с повторными запусками программы. Тем не менее, реализация представляет собой важную основу для будущей интеграции с другими техниками обнаружения состояний гонки, в частности с подходом, уже реализованным в инструменте RaceHunter.

Ключевые слова: верификация; динамический анализ; ограничение контекста.

Для цитирования: Руденчик В.П., Андрианов П.С., Мутилин В.С. Ограничение количества переключений потоков при динамическом анализе многопоточных программ. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 133–148 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–41.

1. Introduction

The verification of multithreaded (parallel) programs is more complex than that of single-threaded programs. This is primarily due to race conditions – a type of error unique to multithreaded programs and notoriously difficult to detect. Race conditions occur when multiple threads access shared memory simultaneously and can lead to nondeterministic program behavior. The problem of race detection is NP-hard [1]. As a result, numerous techniques for automatic race detection have been proposed and continue to be actively developed.

Approaches for detecting race conditions are generally classified as static or dynamic. Static methods analyze source code without executing the target program. They can detect potential race conditions across all possible executions but often produce false positives due to over-approximation. Dynamic methods, on the other hand, detect race conditions by monitoring program executions on specific inputs. This paper focuses on dynamic methods.

The main benefit of dynamic analysis is its high precision: it generally reports issues that actually occur during execution. And its main limitation is incompleteness: it can only detect bugs that

manifest under the specific inputs and execution paths explored during analysis. As a result, it may miss rare or nondeterministic bugs unless executions are carefully guided or repeated under varying conditions.

Analyzing all possible execution paths of a multithreaded program is generally infeasible due to the combinatorial explosion of possible thread interleavings. Context bounding is a dynamic analysis technique that addresses this problem by limiting the number of thread switches in each explored execution path. It systematically explores all execution paths that satisfy a specified bound. The method is scalable and has been shown to detect many real-world bugs even with relatively small bounds [2]. However, its high analysis time, caused by repeated executions, and its implementation as a stand-alone tool [3], limit its applicability.

The RaceHunter tool [4] provides a unified framework for describing and implementing various dynamic analysis methods. Each method fits a common interface and is implemented as an independent unit. This allows it to be used independently, and at the same time greatly simplifies its combination with other methods.

The main contribution of the paper is an implementation of context bounding analysis in the existing dynamic analysis tool, RaceHunter.

The paper is organized as follows. Section 2 provides a brief overview of existing dynamic program analysis methods and the tools that implement them. Section 3 presents the RaceHunter framework. Section 4 describes the implemented approach in terms of the RaceHunter framework. Finally, the evaluation of the implemented approach is presented in Section 5.

2. Preliminaries

We focus on multithreaded programs - that is, programs that contain more than one execution thread. The central problem is detecting race conditions in such programs. To simplify the description, we will consider data races, which are a narrower class of errors. However, the presented approach is able to detect general race conditions. A *data race* is formally defined as follows: two accesses from different threads to the same memory location are called conflicting if at least one of the accesses is a write. Two conflicting accesses are said to form a data race if the order of their execution is undefined.

Dynamic analysis techniques are used to detect race conditions. Dynamic analysis is a type of program analysis that is based on observing program executions on concrete inputs. A *program execution* refers to a specific run of the program, during which a particular sequence of operations is performed, determined by the program's logic, input data, and runtime environment. The sequence of operations in a program execution is referred to as an *execution path*. It represents one possible interleaving of operations across all threads of a multithreaded program. Note that we assume sequential consistency and do not consider weak memory models.

Below is a brief overview of several dynamic analysis methods, along with examples of tools that implement them.

2.1 Lockset and happens-before algorithms

Most dynamic data race detection tools are based on one of the following algorithms: happens-before [5], lockset [6], or a combination of both [7].

The happens-before algorithm is based on introducing a partial order over the set of memory accesses. In particular, two conflicting accesses are considered to form a race condition if they are not comparable under this order.

The lockset algorithm is based on tracking the set of acquired locks. In the lockset algorithm, two conflicting accesses are considered to form a race condition if their sets of acquired locks do not intersect.

To detect data races, a combination of these algorithms can be used [7]. The combination of the lockset and happens-before algorithms was implemented in ThreadSanitizer [8], a tool for dynamic analysis. However, the combination showed limited practical benefit due to false positives. That is why later versions of ThreadSanitizer do not rely on the lockset algorithm to find data races.

In ThreadSanitizer, memory accesses are grouped into segments - sequences of events within a single thread that contain only memory access events (i.e., no synchronization events). Each memory access event belongs to exactly one segment. A partial order is established on these segments using happens-before relations. The state of ThreadSanitizer is a set of unordered segments for each memory location. On every memory access (which are tracked using instrumentation), the state is updated and the algorithm performs a check for conflicting accesses (accesses from different threads with at least one of them being a write). Finally, if no happens-before relationship exists between two accesses and they are conflicting, a data race is reported.

2.2 Breakpoint-watchpoint approach

Another approach used for race detection is the breakpoint-watchpoint approach. It is based on placing breakpoints and watchpoints on a certain set of memory accesses. When a breakpoint is triggered for the memory that is being accessed, a watchpoint is set for the same memory, and a short waiting period follows for a thread that triggered the breakpoint. If the watchpoint is triggered in this period, which means that some other thread is accessing this memory, a race condition is detected.

This approach is implemented in the tools KCSAN [9], DataCollider [10], and RaceHunter [4]. In KCSAN and DataCollider, the set of memory accesses where breakpoints are placed is chosen randomly. In RaceHunter, each pair of conflicting memory accesses is analyzed.

Let's take a closer look at how the breakpoint-watchpoint approach in the RaceHunter tool works. During the first execution of the program (monitoring phase), various events are tracked for each thread, particularly accesses to shared memory. The resulting trace of events is then analyzed to identify pairs of conflicting memory accesses. For each such pair, the program is executed again (race provocation phase), with breakpoints set at the two conflicting accesses. When one of the breakpoints is triggered, the corresponding thread pauses for a limited time. If the second breakpoint is subsequently triggered, the race condition between the two conflicting accesses is considered detected.

The enhanced capability of RaceHunter to identify data races is achieved at the cost of increased verification time (primarily due to repeated executions) and memory consumption during the monitoring phase.

2.3 Context Bounding

All of the approaches described above can miss race conditions due to exploring only a fraction of possible execution paths of a multithreaded program.

A complete analysis of all possible thread interleavings is practically impossible in most cases because of the number of such thread interleavings. Consider a program with n threads where each thread executes k atomic operations. In this case, the number of possible interleavings of these operations exceeds $(n!)^k$. This means that the total number of interleavings grows at least exponentially with respect to both n and k. Analyzing such a vast number of execution paths is feasible only for the simplest examples and does not scale to real-world programs.

The number of analyzed execution paths can be significantly reduced by using iterative context bounding [2], in which only paths with a limited number of thread switches are analyzed.

Consider a context bound c, which is the maximum number of preemptive thread switches. In a program with n threads, where each thread executes k operations, the number of possible

interleavings with the bound c grows as $(n^2k)^c n!$ [2]. It is a polynomial growth with respect to k, making context bounding a scalable technique for larger programs.

Context-bounding is implemented in the CHESS tool [3]. CHESS systematically explores all possible thread interleavings using iterative context bounding. That means that it starts by analyzing execution paths without preemptions ($c = 0$), and after analyzing all paths with c preemptions, it proceeds to analyze those with $c + 1$ preemptions. For each execution path, CHESS uses Goldilocks [11] algorithm to check for data races. In theory, such an approach could take as long as an exhaustive search, but in practice, CHESS is capable of detecting race conditions missed by other algorithms even with a bound $c < 3$.

The benefits of this approach include its scalability to large programs and the ability to detect all race conditions that can be achieved with a given context bound c.

However, it also has drawbacks: the verification time is increased due to the large number of repeated executions, and the standalone implementation of the method makes integration with other approaches challenging.

3. Framework Overview

As previously stated, the goal of this work was to implement context bounding analysis within the existing dynamic analysis tool, RaceHunter. To describe the proposed context-bounding-based approach, we must first introduce the RaceHunter framework into which it is integrated. RaceHunter relies on instrumentation to control program execution, as detailed in Subsection 3.1. The required structure of test programs is described in Subsection 3.2. The RaceHunter algorithm is presented in Subsection 3.3, and the *analysis*, which is an abstraction for representing a dynamic analysis method, is defined in Subsection 3.4.

3.1 Instrumentation

In order to intercept program operations and control program execution, instrumentation is used. At the compilation stage, calls to the **OnEvent** function (see Subsection 3.3) from the RaceHunter library are automatically inserted immediately before each memory access event using ThreadSanitizer's default instrumentation. That allows RaceHunter to acquire control over program execution just before a memory access is performed, enabling the analysis to react based on information about the upcoming event and without executing it yet.

The **OnEvent** function is called concurrently for events from different threads, without the use of synchronization primitives. This concurrency allows RaceHunter to control the execution of the target program: based on the intercepted event, the analysis decides whether the event should be executed immediately or later. In the former case, **OnEvent** returns and the thread proceeds; in the latter, the thread is temporarily suspended within RaceHunter while other threads continue. This mechanism enables the analysis to enforce a specific execution path.

3.2 Test structure

The target test program must have a specific structure. To enable repeated execution, a test function should be wrapped in a loop. An example of such a program is shown in Listing 1. The `main` function contains a loop with calls to the `Reset`, **OnStart**, and **OnRestart** interface functions. The function-under-test `test_func` performs some concurrency operations. Note that the loop body is not limited to a single test function and may include an arbitrary set of operations.

The `test_func` function is called from the loop's body, allowing it to be executed repeatedly. Each iteration of the loop corresponds to a single execution of the test function. The condition for executing the loop's body is the function **OnRestart** (line 27 in Listing 1) of RaceHunter algorithm. The `Reset` function (line 24) resets the values of the shared variables to the default ones. The **OnStart** function (line 25) allows the algorithm to track and handle the start of each iteration.

```
1  int n = 0;
2
3  void Reset() { n = 0; }
4
5  void func1() {
6    tmp1 = n + 1;
7    n = tmp1;
8  }
9
10 void func2() {
11   tmp2 = n + 2;
12   n = tmp2;
13 }
14
15 void test_func() {
16   thread th1(func1);
17   thread th2(func2);
18   th1.join();
19   th2.join();
20 }
21
22 int main() {
23   do {
24     Reset();
25     OnStart();
26     test_func();
27   } while (OnRestart());
28 }
```

Listing 1: A test program.

3.3 RaceHunter algorithm

Algorithm in Listing 2 is the key component of the RaceHunter tool. Its input is an abstract *analysis*, which is used for program analysis, e.g. breakpoint-watchpoint approach or context bounding approach. Its attributes are a current *state*, a current *target* and the sets of targets - *waitlist* and *reached*. Three functions - **OnEvent**, **OnStart**, and **OnRestart** - are called from the instrumented target program.

For each iteration of the algorithm - equivalently, for each execution of the target program - a *target* is specified. Informally, a *target* represents the objective of the current iteration of the analysis. The set *waitlist* contains targets that are yet to be explored, while the set *reached* contains all targets for the analyzed program, both explored and unexplored. By definition, *waitlist* $\subseteq$ *reached*.

At the start of the analysis, *reached* contains only the initial target provided by the *analysis*, *waitlist* is empty, and *target* is the initial target. Both *reached* and *waitlist* are populated with targets by the `GetNewTargets` operator of the analysis, which is called from the **OnEvent** function.

The **OnRestart** function, which is called at the end of each iteration, updates the *target* of the algorithm for the next iteration. It selects a new target from the *waitlist* and assigns it to *target*. If the *waitlist* is empty, the analysis stops.

The *state* in Listing 2 represents the current state of the analyzed program. The **OnStart** function, which is called at the start of each iteration, resets the *state* to its initial value.

The *state* is updated via the `Transfer` operator of the analysis and is used to create new targets through the `GetNewTargets` operator.

The **OnEvent** function of the algorithm is called for each memory access event, as described in Subsection 3.1. It takes as input an event *e* and a thread identifier *tid*.

The **OnEvent** function does the following. First, it calls the `Transfer` operator of the *analysis*. This operator updates the *state* and returns a Boolean value *res*, which is then used to determine whether the current thread should continue execution or be paused. Because of the concurrent nature of **OnEvent**, when a thread is paused within **OnEvent**, other threads proceed and potentially modify the global *state*. As a result, when the paused thread resumes after a timeout and calls `Transfer` again, the state is likely to have changed, which may affect the outcome of the subsequent `Transfer` call. Target program progression and no looping are ensured by each specific implementation of the `Transfer` operator.

After that, the **OnEvent** function calls the `GetNewTargets` operator of the analysis. This operator returns new targets, which are then added to the sets *reached* and *waitlist*. Finally, the **OnEvent** function returns, allowing the intercepted event to be executed.

```
 1 waitlist := ∅
 2 reached := {analysis.getInitialTarget()}
 3 target := analysis.getInitialTarget()
 4 state := analysis.getInitialState()
 5 function OnStart
 6   state := analysis.getInitialState()      # Reset state
 7 end function
 8 function OnRestart
 9   if !waitlist.Empty() then
10     target := waitlist.Pop()
11     return true              # Restart with a new target
12   end if
13  return false                # No more targets ⇒ stop analysis
14 end function
15 function OnEvent
16   res := analysis.Transfer(state, e, tid, target)
17   while !res do              # While Transfer not successful
18     WaitWithTimeout()        # Pause thread
19     res := analysis.Transfer(state, e, tid, target)
20   end while
21   newTargets := analysis.GetNewTargets(state, e, tid, target)
22   for t : newTargets do
23     if !reached.contains(t) then
24       waitlist.add(t)
25       reached.add(t)
26     end if
27   end for
28 end function
```

Listing 2: RaceHunter algorithm.

3.4 Dynamic Analysis

As previously stated, in the RaceHunter framework, each approach is implemented as an instance of an abstract *analysis*. Each *analysis* must define a *state* and a *target* and implement each of the following interface functions: `Transfer`, `GetNewTargets`, `GetInitialTarget`, and `GetInitialState`. These functions are used by the algorithm, and although their specific implementations depend on the concrete analysis, they can be generally described as follows:

- `Transfer`: updates the *state* of the analysis for a given event *e* in thread *tid* and the *target*, and determines whether the thread *tid* should be suspended.
- `GetNewTargets`: generates new targets based on the current *state*, the given event, and the *target*.

- `GetInitialTarget`: returns the initial target for the first iteration.
- `GetInitialState`: is used to reset the *state* at the beginning of each iteration.

The `Transfer` operator not only updates the *state* for a given target program event, but also determines whether the thread should pause or continue execution, as described in Section 3.3. It returns a Boolean value indicating whether the current thread should be suspended or allowed to proceed. Note that the *state* is passed to `Transfer` by reference, allowing the operator to modify it directly.

In the breakpoint-watchpoint approach implemented within the RaceHunter framework, the *target* is defined as a pair of conflicting accesses to shared memory. The *state* tracks which memory accesses have been executed so far. When one of the accesses from the *target* is encountered by the `Transfer` operator, the operator temporarily suspends the corresponding thread. During this suspension, `Transfer`, when called for other threads, checks whether the other access in the pair is about to be executed. If it is, a race condition is reported. The `GetNewTargets` operator for a given shared memory access identifies new pairs of conflicting accesses, which then become new targets for the next iterations.

4. Implementation

The proposed approach performs a systematic analysis of all possible program executions such that the number of thread switches in the execution path does not exceed a specified bound. The description of the implementation of this approach in the framework described in the previous section is presented in Subsection 4.1. Subsections 4.2-4.4 introduce the auxiliary class Scheduler used in the implementation. The limitations of the approach are discussed in Subsection 4.5, and some implementation features are provided in Subsection 4.6.

4.1 Context Bounding analysis

We now describe context bounding analysis. To do so, we need to define the four functions introduced in Section 3.4 along with the *state* and the *target* of the analysis. We begin by defining the *target* and the *state*.

The goal is to analyze all possible execution paths in which the number of thread switches does not exceed a given context bound. Accordingly, in our analysis, the *target* of each iteration represents a unique execution path. Similarly, the *state* of the analysis represents the current partial execution path.

Each execution path is described by a sequence of events. Since the events themselves are not important in our implementation, the sequence of events is abstracted to a sequence of corresponding thread identifiers. We refer to such sequences of thread identifiers as prefixes. Both the *target* and the *state* in our analysis are represented by such prefixes.

The `GetInitialState` operator returns a *state* with an empty prefix.

The `GetInitialTarget` operator returns a *target* with an empty prefix, which, in the case of targets, corresponds to an arbitrary execution path. Listing 3 provides a description of `Transfer` and `GetNewTargets` operators.

The `Transfer` operator updates the *state* (line 3 in Listing 3) by appending a thread identifier to the prefix. It does so as follow. For a given event, the `Transfer` operator decides if the event should be executed. The decision process, which includes selecting the thread necessary to recreate the execution path specified by the *target*, is described in detail in Sections 4.2, 4.3, and 4.4. When execution is permitted, the `Transfer` operator adds the corresponding thread identifier to the end of the *state*'s prefix and returns `True`; otherwise, the prefix remains unchanged and `False` is returned. As a result, by the end of an iteration, the *state*, which was empty at the start of the iteration, accumulates the full path of this execution.

```
 1 function Transfer(&state, e, tid, target)
 2   if scheduler.ResumeThread(state, tid, target) then
 3     state.Update()
 4     return true
 5   else
 6     return false
 7   end if
 8 end function
 9
10 function GetNewTargets(state, e, tid, target)
11   if TargetRecreated(state, target) then
12     for id : {AllIds \ state.LastThreadId} do
13       newTarget = copy(state)
14       newTarget.swapLastElementTo(id)
15       if newTarget.abide(bound) then
16         targets.add(newTarget)
17       end if
18     end for
19     return targets
20   else
21     return ∅
22   end if
23 end function
```

Listing 3: Context Bounding analysis functions

The `GetNewTargets` operator generates new *targets* only if the *target* from the current iteration is already recreated (see the `TargetRecreated` function at line 11). It does so as follows: it replaces the last thread identifier (line 14) in the updated state's prefix (just added by the `Transfer` operator) with each of the other available thread identifiers. Note that `GetNewTargets` does not modify the *state*. All resulting prefixes that satisfy the context bound (line 15) are returned as new targets (line 19).

We state that in our implementation, each *target* generated by the `GetNewTargets` operator during analysis uniquely corresponds to a specific execution path. The *target* prefix defines the beginning of the path but does not explicitly define the rest of it. However, the way the rest of the path is constructed during execution ensures its uniqueness. Moreover, presuming that all execution paths are finite, we state that each execution path that satisfies the given context bound is represented by some *target* generated by the `GetNewTargets` operator and is explored during the analysis.

To better illustrate this, let us consider the tree of possible execution paths shown in Fig. 1 for the program in Listing 1. For simplicity, only operations from the concurrent functions `func1` (thread 1) and `func2` (thread 2) are shown. Each edge represents a program operation, and each vertex represents a prefix. Vertex *A* corresponds to the beginning of an execution, while vertices *N* through *S* correspond to the end of an execution. An execution path is a sequence of edges from the root *A* to one of the leaves *N*–*S*. There is a one-to-one correspondence between all execution paths and the leaves *N*–*S*, so we refer to each execution path by its corresponding leaf. There are a total of six execution paths in this example: *N*, *O*, *P*, *Q*, *R*, and *S*.

First iteration. The analysis starts with an empty *target* prefix (*target* = {}) and an empty *state* prefix (*state* = {}), both of which correspond to the vertex *A* in the example. Assuming there is no context bound, our goal is to explore each of the six execution paths exactly once.

The `Transfer` operator selects the next thread to execute and appends its thread identifier to the *state*. The `GetNewTargets` operator generates new targets by appending all other possible thread

identifiers to the original *state*. In the example, the `Transfer` operator selects one of the two threads, updating the *state* to correspond to either vertex *B* or *C*. Suppose that `Transfer` selects the first thread, so the *state* now corresponds to vertex *B* (*state* = {1}). The `GetNewTargets` operator then replaces the last thread identifier in the *state* with the other available one, generating a new *target* corresponding to vertex *C* (new *target* = {2}).

After that, the `Transfer` operator selects a thread to execute again. Suppose it selects the first thread once more, updating the *state* to correspond to vertex *D* (*state* = {1, 1}). The `GetNewTargets` operator then generates a new *target* corresponding to vertex *E* (new *target* = {1, 2}). Then, the execution continues in the only remaining possible way and finishes, updating the *state* to correspond to vertex *N* (*state* = {1, 1, 2, 2}).

The result of the first iteration is as follows: the execution path *N* is explored, and two new targets are generated: {1, 2} and {2}, corresponding to vertices *E* and *C*, respectively.

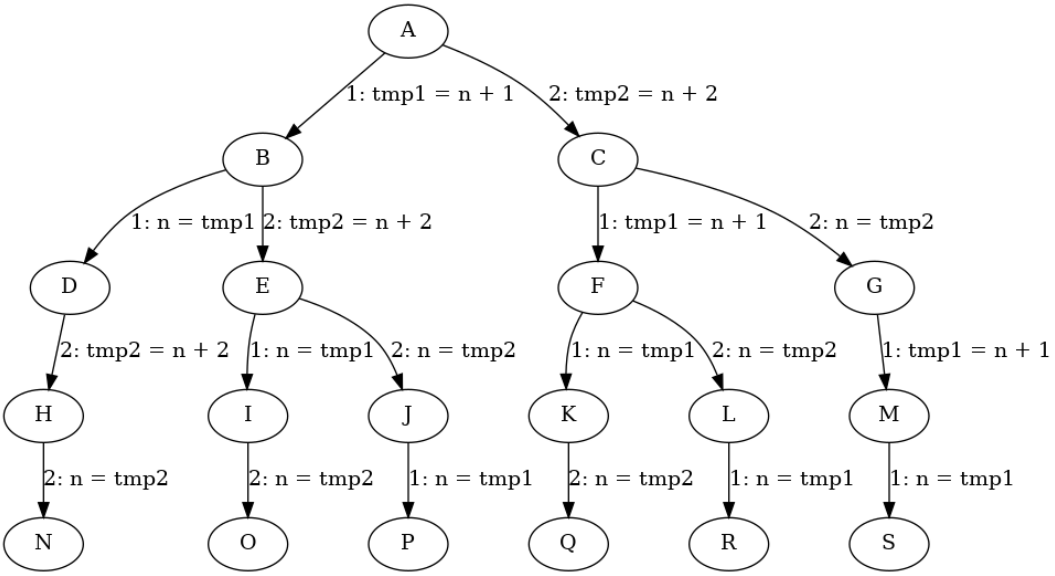

Fig. 1. Execution paths tree.

Second and subsequent iterations. Next, the analysis explores one of the new targets. Suppose that it explores *target* = {1, 2}, which corresponds to vertex *E*. In this iteration, one of the two possible execution paths, *O* or *P*, will be explored, and the other one will be represented as a new target that corresponds to vertex *I* or *J* and will be explored in a future iteration. Similarly, during the iteration with the target corresponding to vertex *C*, one of the execution paths *Q*, *R*, or *S* from this subtree (the subtree with the root *C*) will be explored, and the remaining paths will be represented as new targets and explored later.

Summing up, each iteration starts with the *target* prefix recreation, which corresponds to going down the tree to an unexplored subtree represented by the *target*. In the first iteration, the entire tree is unexplored - this is precisely what an empty target represents. For such an unexplored subtree, the `Transfer` operator incrementally selects a path from the root of the subtree to a leaf, while the `GetNewTargets` operator keeps track of all other possible paths in this subtree by representing each unexplored sub-subtree as a new *target*. This process ensures that each execution path is explored exactly once.

4.2 Scheduler

In this section, we describe the scheduler - an auxiliary component in our implementation. Its internal attributes can be viewed as part of the *state*, and its methods as helper functions for the context bounding *analysis*.

The scheduler's main purpose is to choose which thread will proceed next at a given point during an iteration. This is done through a call to the scheduler's boolean function `ResumeThread` from the `Transfer` operator, as shown in Listing 3.

The `ResumeThread` function ensures *target* prefix recreation. It does so by identifying a thread required to recreate the *target* from the current *state* and allowing only this thread to proceed.

The *target* prefix is recreated in our implementation as follows. At the start of each iteration, the scheduler checks if the *target* prefix is empty. If it is not (as is the case in all iterations except the first), the prefix has to be recreated first. For this, the scheduler creates an iterator over the *target* prefix. This iterator tracks the part of the prefix that has already been recreated and points to the identifier of the next thread required for recreation. When the `ResumeThread` function is called for a thread *tid*, it first checks whether the iterator still points to an identifier to see if the prefix has been already recreated. If it is not the case, the `ResumeThread` function then compares *tid* to the identifier to which the iterator is pointing to. If they are equal, then the iterator moves to the next identifier and the scheduler allows the thread to proceed. Otherwise, the thread is suspended, and the scheduler waits for another thread.

Once the *target* prefix has been fully recreated, the `ResumeThread` function chooses the next thread according to the scheduler's strategy, which is described in the next section.

4.3 Scheduler strategies

After the *target* prefix has been recreated, the `Transfer` operator guides the program execution to the end, thereby exploring a unique execution path. Theoretically, the specific choice of which path is executed has no effect on the correctness of the analysis - it only determines the order in which the execution paths are explored. Regardless of this order, each execution path will be explored during the analysis exactly once. However, in practical applications, the choice of execution path can be important. To demonstrate this, we implemented three scheduler strategies for selecting the next thread to execute: the *random strategy*, the *minimum switches strategy*, and the *fair scheduling strategy*.

The *random strategy* does not suspend any thread and allows every event to be executed straight away, thus performing essentially a random execution path. This strategy is natural and straightforward, but it has an important flaw. The number of thread switches in a random execution path is typically rather big. Therefore, the executed path will likely not satisfy the context bound and the analysis will waste time exploring it. That applies to essentially all of the iterations, where the *target* prefix represents a partial and not full execution path. Although the *target* prefix satisfies the context bound, the rest of the path will be a random sequence of events, and the total number of thread switches in the full execution path is likely to exceed the bound.

Essentially, with this strategy, the analysis will initially explore irrelevant execution paths that do not satisfy the context bound. However, on each iteration, it will generate *targets* with increasingly longer prefixes that do satisfy the bound. Eventually, a large enough part of the path will be defined by the *target* prefix rather than by the strategy, making it more likely for the full path to satisfy the bound. Ultimately, all possible execution paths that satisfy the context bound will be explored, but the analysis is likely to first spend a significant amount of time exploring irrelevant paths. Therefore, this strategy is not well-suited for our purposes.

The *minimum switches strategy* chooses the same thread that was executed most recently, attempting to continue execution without performing a thread switch. The scheduler tries to do so as long as it is possible. When forced to switch to another thread, it chooses any other available thread.

The idea behind this strategy is that by avoiding thread switches, the execution is likely to follow a path that satisfies the context bound. In contrast to the *random strategy*, this strategy significantly reduces the number of iterations and, consequently, the overall analysis time. A disadvantage of this strategy is that the scheduler can start unrolling a loop in the target program that can only be exited by an action from another thread, causing the analysis to be stuck in an infinite loop. This problem can be avoided by the next strategy.

The *fair scheduling strategy* chooses the same thread that was executed most recently, but only if the number of consecutive executions of events in this thread does not exceed a certain limit. That helps avoid unrolling infinite loops by enforcing a switch to another thread after some time, while still executing a path with relatively few thread switches that is likely to satisfy the context bound.

4.4 Ensuring analysis progression

During target program execution, situations may occur where the thread that should be selected by the scheduler is not available for execution - for example, if a thread is waiting on a lock in the target program. Waiting indefinitely for this thread and not allowing other threads to progress would lead to a deadlock. However, the scheduler accounts for such situations and includes a mechanism to avoid them.

As described in Section 3.3, threads that were not selected by the scheduler are suspended with a certain timeout. After the timeout expires, these threads call the scheduler (through `Transfer`) again to check if the scheduler will allow them to proceed now. In our implementation, the scheduler counts the number of such calls. If this number exceeds a certain threshold and the originally selected thread has not become available during that time, the scheduler assumes that the selected thread is blocked and chooses a different thread to execute. Note that the threshold is proportional to the number of threads in the target program. In this way, the scheduler ensures that the target program progresses.

4.5 Limitations of the approach

As mentioned above, each execution path is represented as a sequence of events, which in our implementation is abstracted as a sequence of corresponding thread identifiers. The main drawback of this approach is that it does not account for potential non-determinism in the behavior of the target program.

Our implementation is based on the assumption that thread scheduling is the only source of non-determinism, and therefore that the number of events in each thread remains the same across iterations. In practice, however, this is not always the case. When such extraneous non-determinism occurs during analysis, the scheduler may execute a path that is different from the one intended.

Adequate handling of non-determinism could be achieved by tracking and interleaving specific events rather than relying on the consistency of the number of events (i.e., performing a thread switch after a predetermined event rather than after a fixed number of events). This is one of the directions for future improvement.

Note that one of the differences from the implementation of context bounding in the CHESS tool is the definition of the bound: in CHESS, the bound limits the number of preemptive thread switches, whereas in the version of our analysis used in this paper, the bound applies to the total number of switches.

Another important consideration is the choice of events to be treated as interleaving points. In the current implementation, the sequence of thread identifiers abstracts the sequence of all memory access events in the target program. That means that every memory access is considered a possible interleaving point, which is excessive. For instance, authors of [2] show that it is sufficient to insert scheduling points before synchronization operations to not miss any errors in the program. Reducing the set of interleaving points - particularly through partial-order reduction techniques - is one of the directions for future optimization.

4.6 Implementation Features

One common source of non-determinism in the program behavior is lazy initialization that causes a data structure to be initialized during the first execution and not beforehand. Because of that, the number of events can vary between the first execution and subsequent ones, potentially affecting scheduling in our implementation. To address this problem, we added preparatory iterations before the analysis begins. During these iterations, the target program is executed in full, ensuring that all initializations occur on these iterations, before the actual analysis. A similar issue and solution are present in the CHESS tool [3].

Unlike the context bounding analysis implemented in the CHESS tool (see Section 2.3), the proposed method is not iterative; that is, it does not increment the bound after analyzing all execution paths for a given bound. This limitation is mostly technical, stemming from an implementation decision. The proposed method can be easily modified to perform iterative context bounding if needed.

5. Evaluation

We evaluated the approach on two benchmark sets. The first one contains small manually prepared tests. The second one is based on existing tests for an industrial virtual machine. We used a machine with an Intel® Core™ i5-8250U CPU (8 cores, 1.60 GHz) and 8 GiB of memory running Ubuntu 22.04.5 LTS (64-bit).

5.1 Small tests

The implemented approach explores all possible execution paths but does not itself detect or report race conditions. To detect race conditions, we combine it with ThreadSanitizer's data race detection. This combination can be done easily in the RaceHunter framework.

Our context bounding approach, in combination with ThreadSanitizer (**CB + TSan**), was evaluated against three other methods:

- plain ThreadSanitizer (**TSan**);
- the breakpoint-watchpoint approach implemented in the RaceHunter tool (**WP**);
- a combination of WP and TSan (**WP + TSan**).

The reason we evaluated our approach against the **WP + TSan** combination is to ensure that the **CB + TSan** combination does not find a race that **TSan** alone misses simply due to a few additional executions. The **WP + TSan** combination provides a few additional executions that can improve **TSan**'s ability to find races. We want to distinguish this factor from the benefit provided by the **CB** analysis.

By default, the context bounding analysis was used with a bound of two thread switches. The default scheduling strategy (see Section 4.3) was the *minimum switches strategy*. If this strategy resulted in unrolling an infinite loop, the *fair scheduling strategy* was used instead, with a limit of 1000 consecutive events in one thread.

The benchmark set is composed of two parts. 15 of the tests are from the RaceHunter repository.

The other 32 tests were adapted from the *sctbench* repository [12] to satisfy the required test structure (see Section 3.2). Some of the 32 adapted tests contain assertions that check if the existing race condition alters the expected program behavior. A violation of such an assertion means that a race condition does exist in the test and that it did affect the program's outcome. One of the tests contains a deadlock. This test was excluded from the benchmark set as irrelevant. Therefore, the overall number of tests is 46.

The results for 46 of the tests are presented in Table 1. For each method, the total analysis time was calculated as the average over three runs. The total number of reported races was determined as the maximum across three runs. For each method, the number of reported races in any single run differed

from the maximum by no more than three, meaning that only a few of those reports appear non-deterministically. Note that some of the reported races are false positives.

Table 1. Results for small benchmarks.

Method	Total time	Races reported
TSan	9.889s	28
WP	24.051s	30
WP+TSan	26.24s	36
CB+TSan	22m 10.907s	42

The **CB** + **TSan** combination can detect race conditions in two ways: either TSan can report a race, or **CB** can explore an execution path in which an assertion that indicates a race condition is violated. Table 2 groups all 46 tests by how the **CB** + **TSan** combination compares to the others: the first column shows the result category, and the second column shows how many tests fall into that category.

Table 2. Summary of results.

Result of **CB+TSan**	Number of tests (out of 46)
New races reported by **TSan**	3
New race-indicating assertion violation	7
Same as plain **TSan**	31
Same as **WP+TSan**	3
Non-unique race-indicating assertion violation	1
Analysis error	1

In 3 of the tests, **TSan** in combination with **CB** was able to report true data races that were missed by all of the other approaches. In 7 other tests, the combination of **CB** and **TSan** resulted in an assertion violation that indicates a race condition. This assertion was not triggered by any of the other approaches, and this is the main benefit of the **CB** approach.

For 31 out of 46 tests, the **CB** + **TSan** combination reported the same data races as plain **TSan**, meaning that context bounding did not provide any improvement for those tests.

For 3 of the tests, the **CB** + **TSan** combination reported the same data races as **WP** + **TSan**, which were not detected by plain **TSan**. This suggests that TSan did not necessarily need context bounding to report these races, but rather benefited from the additional execution paths explored when combined with the **WP** approach. Note that for one of these three tests, the reported races were false positives.

In one of the tests, an assertion violation indicating a race condition was triggered by both the **CB** + **TSan** combination and the **WP** approach. The analysis of another test was terminated due to a violation of an internal assertion of the context bounding analysis.

For the excluded test with a deadlock, **TSan** successfully detected and reported the deadlock even without any of the added methods. The context bounding analysis, however, was able to explore an

execution path that led to the deadlock and, therefore, did not complete. This raises an interesting question regarding the applicability of context bounding analysis for deadlock detection.

In summary, for 10 out of 46 tests, the use of context bounding analysis resulted in the detection of race conditions that were missed by other approaches. In 7 of these tests, the context bounding analysis was able to find and explore an execution path in which the race condition directly causes incorrect behavior, resulting in an assertion violation. The availability of such a path to an assertion violation clearly points to a real bug in the program and significantly simplifies the debugging process.

5.2 ARK VM tests

The benchmark set is based on existing TypeScript tests for the ARK VM [13]. The ARK VM was modified to support the RaceHunter tool.

The benchmark set contains 22 tests, but the majority of them require significant resources. Thus, we selected only 4 of them that required less time for 1000 iterations. The main goal is to evaluate time consumption.

The tool configuration uses plain context bounding analysis without ThreadSanitizer. Thus, no data races are reported, and the tool just explores different executions. Note that the breakpoint-watchpoint approach was successfully applied to the VM and found races, but this is beyond the scope of this paper. The context bounding analysis is limited by 2 thread switches and the scheduling strategy is the *fair scheduling strategy* with a limit of 10 000 consecutive events in one thread.

The results are presented in Table 3. The table contains an approximate number of memory accesses (rounded to the nearest 1000) for one iteration and the time required to explore the first 1000 iterations for a total of 4 tests. The time required to explore one iteration exceeds one second even for the smallest of the tests.

Table 3. Results on ARK VM tests.

Test name	Approximate number of memory accesses	Time for 1000 iterations
async_call_3	49000	25m28s
Await	45000	27m46s
concurrent_start_gc	1300000	2h22m41s
launch_instruction_3	24000	20m31s

The estimate of the total number of iterations is tens of thousands. So, the estimate of the total time is multiple hours for simple tests and multiple days for more complicated tests. It is barely acceptable, as the analysis would still take too long to complete. However, the approach fundamentally can be applied to complicated software and the further problem is scalability. For instance, the problems arise with overflowing some variables and using too much memory, since the framework was not initially designed for these many iterations. This shows that the problem of reducing the state space is relevant and should be explored in future work.

6. Conclusion

We presented a context bounding approach which is integrated into the RaceHunter tool. The most promising further direction is a combination of the context bounding analysis with the watchpoint-based approach.

The context bounding approach shows fundamental applicability to industrial software. However, it definitely requires further optimizations to reduce the number of considered executions. One

possible idea is to develop a coverage-based approach such as fuzzing. The target is considered only if new coverage is discovered.

A problem with non-determinism can be mitigated by more control over the target program. One possible idea is to implement a “step-by-step” version of the context bounding analysis. This approach produces only sequentialized executions. However, the main research question in this case is its scalability.

References

[1]. R. H. B. Netzer and B. P. Miller, “What are race conditions? Some issues and formalizations”, LOPLAS, vol. 1, pp. 74–88, 1992.
[2]. M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs”, vol. 42, p. 446–455, jun 2007.
[3]. M. Musuvathi, S. Qadeer, and T. Ball, “Chess: A systematic testing tool for concurrent software”, Tech. Rep. MSR-TR-2007-149, November 2007.
[4]. E. A. Gerlits, “Racehunter dynamic data race detector”, Programming and Computer Software, vol. 50, pp. 467–481, Dec 2024.
[5]. L. Lamport, “Time, clocks, and the ordering of events in a distributed system”, Commun. ACM, vol. 21, p. 558–565, jul 1978.
[6]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs”, SIGOPS Oper. Syst. Rev., vol. 31, pp. 27–37, Oct. 1997.
[7]. R. O’Callahan and J.-D. Choi, “Hybrid dynamic data race detection”, inProceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’03, (New York, NY, USA), p. 167–178, Association for Computing Machinery, 2003.
[8]. K. Serebryany and T. Iskhodzhanov, “Threadsanitizer – data race detection in practice”, in Proceedings of the Workshop on Binary Instrumentation and Applications, (NYC, NY, U.S.A.), pp. 62–71, 2009.
[9]. Kernel Concurrency Sanitizer (KCSAN) google.github.io”, https://google.github.io/kernel-sanitizers/KCSAN.html. Accessed 05-05-2025.
[10]. J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, “Effective data-race detection for the kernel”, pp. 151–162, jan 2010.
[11]. T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: Efficiently computing the happens-before relation using locksets”, pp. 193–208, 01 2006.
[12]. https://github.com/mc-imperial/sctbench.git. Accessed 05-05-2025.
[13]. https://gitee.com/openharmony/arkcompiler_runtime_core.git. Accessed 05-05-2025.

Информация об авторах / Information about authors

Вероника Павловна РУДЕНЧИК – стажер-исследователь ИСП РАН. Научные интересы: статический и динамический анализ программ.

Veronika Pavlovna RUDENCHIK – researcher at ISP RAS. Research interests: static and dynamic program analysis.

Павел Сергеевич АНДРИАНОВ – научный сотрудник ИСП РАН, кандидат физико-математических наук. Научные интересы: статическая верификация, параллельные программы.

Pavel Sergeevich ANDRIANOV – researcher in ISP RAS, Ph.D. Research interests: software model checking, parallel programs.

Вадим Сергеевич МУТИЛИН – кандидат физико-математических наук, ведущий научный сотрудник Института системного программирования им. В.П. Иванникова РАН и доцент Московского физико-технического института. Сфера научных интересов: статический и динамический анализ программ.

Vadim Sergeevich MUTILIN – Cand. Sci. (Phys.-Math.), leading researcher at Ivannikov Institute for System Programming of the RAS and associate professor at Moscow Institute of Physics and Technology. Main research interests: static and dynamic program analysis.