Detecting Data Races in Real-Time Operating Systems with RaceHunter Tool

Evgeny Gerlits
ISP RAS
Moscow, Russia
e-mail: gerlits@ispras.ru

***Abstract*—In this paper, we share our experience in applying a dynamic data race detection tool called RaceHunter to an industrial real-time operating system. We evaluate the performance and memory usage of the tool and present some key learnings.**

***Keywords*—RaceHunter, data race, race condition, dynamic program analysis, real-time operating system, RTOS, ARINC-653.**

I. Introduction

Real-time operating systems (RTOS) usually run application software that controls different equipment. RTOSs are widely used in different fields, e.g., avionics, automotive, medical devices, etc. In this paper, we take a particular RTOS [1] from the civil avionics field. As operating systems and application software in this field must not contain any critical errors leading to crashes, their design and development are regulated by an international standard DO-178C [2].

In order to prevent errors in one application from causing errors in another, the RTOS must follow the time and space partitioning specifications defined in the ARINC-653 standard [3]. This standard provides an API (Application Programming Interface) called ARINC-653 APEX for the application processes to communicate with the OS kernel. APEX provides several thread management services (functions), including those for creating, stopping, and synchronizing threads. By calling APEX services, application threads form concurrent execution contexts inside the OS kernel. These thread execution contexts combined with the kernel service threads and asynchronous interrupts, make an ARINC-653 RTOS kernel a highly multithreaded piece of software.

According to [4], one of the most common issues with multithreaded programs is data races, which occur when there is incorrect synchronization between threads. Data races can result in wrong values in memory, leading to wrong program results and memory access violation exceptions. According to the C [5] language standard, a data race is classified as an undefined behavior, making the whole program execution unpredictable and error-prone. Thus, checking multithreaded software, and especially safety-critical software, for data races is necessary. In this paper, we will use a dynamic data race detection tool called RaceHunter [6].

The paper is structured as follows. In Section II, we outline RaceHunter approach. As RaceHunter is a dynamic data race detection tool, it requires tests to run on. In Section III, we generate tests as simple multithreaded applications calling APEX services in parallel. In Section IV, we present some testing statistics that shows real performance of RaceHunter tool and its memory usage. We learn some lessons from this case study research in Section V. At the end of our paper, we summarize our findings and suggest directions for future research.

II. RaceHunter approach

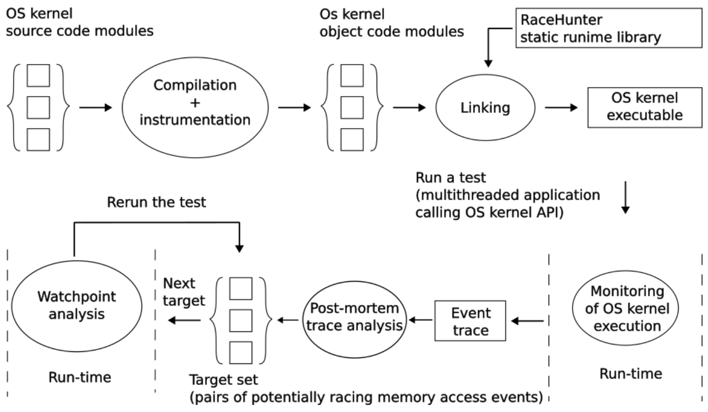

Fig. 1. General scheme of running one test with RaceHunter

A. Instrumentation

Instrumentation makes it possible to monitor and handle various events in the OS kernel executions. More than 99.9% of the OS kernel code is instrumented automatically. RaceHunter instrumentation is implemented as a Clang compiler pass over LLVM IR (Low Level Virtual Machine Intermediate Representation) source code representation. The compiler inserts function calls to the RaceHunter run-time library functions before every memory access instruction, as well as at the beginning of every OS kernel function, before every function return instruction (and implicit function return point). Some code places are instrumented manually, e.g., thread creation function, initial interrupt function, mutex lock and unlock functions.

B. Monitoring

At this stage, a test is run and events that occur in the OS kernel are monitored and recorded in a *trace*. A typical test is a multithreaded application program calling some APEX services from different threads. We do not manage thread scheduling and do not control asynchronous interrupts, i.e., the OS kernel code is executed as usual, but with some slowdown due to instrumentation. Most of the events are executions of OS kernel operations: memory access instruction executions, function calls, and returns. In addition, we monitor and record thread spawning and joining events, interrupt start and finish events, mutex lock and unlock events. The monitoring result is the event *trace* where events within every single execution context (thread or interrupt) are ordered sequentially. Order between events from different execution contexts is not tracked explicitly.

C. Trace analysis

The *trace* is scanned for pairs of conflicting memory access events (memory access instruction executions). Two memory access events conflict if the conjunction of the following conditions holds:

- Two events access a common memory segment.
- At least one event is a write.
- At least one event is not performed atomically.
- The events happen in different execution contexts.
- The events are not synchronized by mutex lock/unlock events, which is checked by the *lock-set analysis*.
- The events are not synchronized by the thread create/join events, which is checked by the *thread concurrency analysis*.

The result of this stage is a set of targets, where the *target* is a pair of *descriptors* of conflicting memory access events. A memory access *descriptor* contains various information about one memory access event, including the memory access instruction location, whether the event has happened in an interrupt or a thread and the function call stack at the time of the event.

D. Watchpoint analysis

For each *target*, the test is re-run once to provoke the potential data race described by the *target*. We provoke data races in the following way. If a memory access event occurs that matches a memory access descriptor from the target, RaceHunter will pause the current execution context (thread or interrupt) for a given amount of time in order to wait for another memory access that matches the other memory access descriptor from the same target. If it occurs, then the potential data race described by the target is confirmed.

E. Comparison with existing dynamic data race detectors

RaceHunter approach strengthens industrial KCSAN [7] approach by systematically checking every conflicting memory access event instead of random memory access sampling performed by KCSAN. Drawbacks of RaceHunter are worse scalability and performance.

RaceHunter applies a variant of *lock-set analysis* to filter out non-conflicting memory access events. It differs from the existing lock-set analyses [8], [9] in that it is a post-mortem *trace analysis*. For the same goals, RaceHunter applies a variant of post-mortem happens-before analysis based on thread spawning and joining events called *thread concurrency analysis*. It differs from the existing happens-before analyses [10], [11] in that it does not utilize vector clocks [12].

RaceHunter approach is most similar to the active testing approach [13], since it looks for potential data races on the first stage and confirms them on the second stage. RaceHunter differs from the active testing in several ways:

- RaceHunter minimizes execution slowdown at the first stage (*monitoring*) by simply logging events for a subsequent post-mortem *trace analysis*, while the active testing approach reveals potential data races at run-time. However, RaceHunter needs to perform extra post-mortem event trace analysis. Run-time analysis can scale better if it consumes a bounded amount of memory, but the active testing approach, presented in the paper [13], does not satisfy this condition.
- RaceHunter does not control the thread scheduling at the second stage (*watchpoint analysis*) except for one delay to catch a data race, while the active testing approach actively controls the thread scheduling, allowing one random thread to execute at a time. Controlling the thread scheduling slows down program execution significantly and does not scale well, but it can help reproduce data races, which is useful for debugging.
- RaceHunter adapts to OS kernels by supporting asynchronous interrupts, while active testing does not support interrupts because it does not provide mechanisms to actively control the arrival of asynchronous interrupts at the second stage.
- RaceHunter tries to accurately detect the target memory access events at the *watchpoint analysis* stage via the *memory access descriptors* containing various information like the function call stack at the time of the event. In contrast, the active testing approach identifies memory access events by the instruction location only, but tries to check nearly every instruction execution with the given location for a data race. Which approach is more effective for finding data races remains a topic for future research. However, the active testing approach leads to excessive delays, slowing down the program execution.

A variant of the active testing approach for distributed memory concurrent programs [14] moves away from pure run-time analysis on the first stage by activating analysis stages in stationary states, which are organized by barrier operations. It also moves away from full control over the execution schedule of processes on the second stage by allowing processes to run freely except for multiple delays applied for instructions with the given location. Memory access instructions are still identified by the instruction location only, which leads to excessive delays. To improve performance and scalability, this variant of the active testing approach gradually lowers the probability of applying a delay for subsequent instruction executions with the target location. However, reducing the probability of applying a delay can lead to missed data races.

III. Tests

RaceHunter is a dynamic data race detector that reveals data races in real program executions. Thus, we had to develop some tests to produce RTOS kernel executions. Our testing approach is as follows. We manually develop a parameterized test t, which is an ARINC-653 application consisting of a number of ARINC-653 processes (threads) running in parallel. We can think of test t as a program with a set of formal parameters $P = \{p_1, ..., p_n\}$. Every thread calls some APEX services that access shared objects in the RTOS kernel. Some test parameters p_i become actual parameters (arguments) of these APEX services. Then we manually prepare a set of values V_i for every test parameter p_i. Our goal is to address the main functional requirements of the APEX services as defined by the ARINC-653 standard.

The Cartesian product $V = V_1 \times V_2 \times ... \times V_n$ gives us a set of argument vectors for $t(p_1, ..., p_n)$:

$$A^{|V| \times n} = \begin{bmatrix} a_1 \\ ... \\ a_{|V|} \end{bmatrix} = \begin{bmatrix} v_{1,1} & ... & v_{1,n} \\ ... & ... & ... \\ v_{|V_1|,1} & ... & v_{|V_n|,n} \end{bmatrix}, v_{i,j} \in V_j$$

Calling the parameterized test $t(p_1, ..., p_n)$ with an argument vector a_i gives us a single test case $t(a_i)$. A set of test cases $TS = \{t(a_1), ..., t(a_{|V|})\}$ produced for a single parameterized test t forms a single test suite.

Table I describes all the test suites that we have developed. The name of a test suite consists of a number of short alias names of APEX services separated by a hyphen meaning that these APEX services are called in parallel from different threads. For example, the test suite name *queuing_port-send-clear-get_status-get_id* implies that there are four threads in every test case running in parallel and executing one and only one of these APEX services: *send_queuing_message*, *clear_queuing_port*, *get_queuing_port_status*, *get_queuing_port_id*. The number of threads in every test case of a test suite (column *Threads* in Table I) is two more than the number of threads we can understand from the test suite name because there is one extra initialization thread and there is one extra auxiliary thread that starts and stops worker threads that actually call APEX services.

IV. Testing results

We have found two data races in the RTOS kernel and some non data race bugs. This result confirms the ability of the RaceHunter approach to find real data race bugs in industrial operating systems.

Table II includes some statistics collected during testing. Table columns have the following meaning. *N* is the test suite number from Table I. T_1 is the test suite execution time without RaceHunter. T_2 is the test suite execution time with RaceHunter. $S = (T_2 - T_1)/T_1$ is the slowdown induced by RaceHunter. *V* is the average *trace* size in kilobytes after the *monitoring* stage of a test case. *E* is the average number of events in a single test case run. *I* is the average number of interrupts in a single test case run. *G* is the average number of *targets* built by the *trace analysis* stage of a test case.

TABLE I
Test suites

N	Test suite name	Test cases	Threads
1	buffer-send-receive	192	4
2	buffer-send-send	288	4
3	buffer-send-get_status-get_id	432	5
4	blackboard-display-read	240	4
5	blackboard-display-display	200	4
6	blackboard-display-clear-clear -get_status-get_id	320	7
7	queuing_port-send-receive	432	4
8	queuing_port-receive-clear -get_status-get_id	324	6
9	sampling_port-write-read	270	4
10	sampling_port-write-write	480	4
11	sampling_port-write-get_status-get_id	540	5

TABLE II
Testing statistics

N	T_1	T_2	S	V	E	I	G
1	00:00:44	05:01:12	410	813	37302	26	171
2	00:00:50	07:10:35	515	805	36933	26	155
3	00:04:30	17:05:09	226	1120	51039	40	186
4	00:00:40	06:34:55	591	852	39043	31	142
5	00:00:33	06:01:32	656	874	39920	37	128
6	00:02:50	21:56:37	463	1596	72547	59	224
7	00:02:13	10:13:20	276	870	39890	26	144
8	00:03:13	14:42:58	273	1384	63070	41	223
9	00:00:15	02:51:12	683	721	33239	15	106
10	00:00:38	06:57:04	657	762	35046	19	107
11	00:02:06	11:38:55	332	1030	47039	26	152

V. Lessons learned

A. Performance

A weak point of RaceHunter is performance. The slowdown fluctuates between 200 and 700 which is too high compared to the industrial tools [7], [8], [10] that slow down execution by 5-20 times.

The overall performance of RaceHunter is directly proportional to the number of targets built by the *trace analysis* stage. Therefore, the more accurate the post-mortem *trace analysis*, the higher the performance. Implementation of post-mortem *lock-set analysis* and post-mortem *thread concurrency analysis* increased the performance of RaceHunter by more than an order of magnitude.

We run all test suites in parallel to reduce the testing time to the duration of the longest test suite. To achieve this, we emulate the target hardware devices with QEMU [15] and run them on multiple high-performance desktop computers (hosts).

B. Precision

RTOS actively uses synchronization primitives that are not defined in the standard library of the programming language. There are also synchronization operations in assembly code. Happens-before-based data race detectors [10], [11] may give false data race warnings when they don't observe even one synchronization event. In contrast, RaceHunter does not output false warnings due to unobserved synchronization events, because it catches data races at the time they really occur. We did not receive any false data race warnings during this case study.

C. Soundness

Data races can occur in certain states of the OS kernel that are difficult to reach by simply calling a specific APEX service and varying its arguments. To improve soundness, we should apply some techniques aimed at the exploration of different thread inter-leavings, like context switch bounding [16].

D. Scalability

Embedded operating systems like RTOS [1] are significantly smaller than general purpose operating systems in code size and the number of concurrent execution contexts. Therefore, the number of events that occur during the execution of even a simple test can be overwhelming. To scale the RaceHunter approach to the general purpose operating systems we should reduce the number of events saved in the *trace* without significant losses in the completeness of the analysis. To do this we probably need to shift *lock-set analysis* and *thread concurrency analysis* to the *monitoring* stage but without control over the thread scheduling and over the arrival of interrupts.

E. Integration complexity

We believe it takes 4-5 man-months to integrate the RaceHunter tool into a new embedded OS or RTOS. Most of the time is spent on the following:

- RaceHunter requires some functions to be implemented in the OS kernel, e.g., a function that returns the identifier of the current execution context.
- Some low-level OS kernel functions must be excluded from instrumentation by manually marking them with a special function attribute.
- Some OS kernel functions should be instrumented manually, e.g., the thread creation function.
- We needed to integrate RaceHunter feature into the OS building process.

VI. Conclusions

A data race in the RTOS kernel can lead to the whole system crashing. ARINC-653 safety features like time and space partitioning do not protect from this kind of errors. Careful testing according to the DO-178C standard using single threaded tests can not guarantee the absence of data races in the highly multithreaded kernel code. Therefore, special data race detection tools like RaceHunter must be applied in the secure software development life-cycle of ARINC-653 compatible real-time operating systems and other safety-critical software.

VII. Directions for further research

Development of tests for RaceHunter is a labor-intensive process. Thus, the transformation of RaceHunter into a dynamic data race fuzzer is a promising future research direction.

We can speed up the *watchpoint analysis* stage by checking each *target* in parallel rather than sequentially. We can also parallelize the *trace analysis* algorithm.

References

[1] V. Cheptsov, A. Khoroshilov, "Robust resource partitioning approach for ARINC 653 RTOS", *Proceedings of 2023 Ivannikov Ispras Open Conference (ISPRAS)*, pp. 33-39, 2023.
[2] RTCA, "Software Considerations in Airborne Systems and Equipment Certification", 2011.
[3] Aeronautical Radio Inc, "Avionics application software standard interface part 1 required services. ARINC Specification 653P1-2", 2005.
[4] S. Lu, S. Park, E. Seo and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics", *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.
[5] ISO and IEC, "ISO International Standard ISO/IEC 9899:2024: Programming languages - C", 2024
[6] E.A. Gerlits, "RaceHunter Dynamic Data Race Detector", *Programming and Computer Software*, vol. 50, no. 6, pp. 467-481, 2024.
[7] M. Elver, et al, "Concurrency bugs should fear the big bad data-race detector", 2020. [Online]. Available: https://lwn.net/Articles/816850.
[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs", *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391-411, 1997.
[9] T. Elmas, S. Qadeer, S. Tasiran, "Goldilocks: Efficiently computing the happens-before relation using locksets", *Proceedings of the International Workshop on Formal Approaches to Software Testing*, pp. 193-208, 2006.
[10] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice", *Proceedings of the Workshop on Binary Instrumentation and Applications*, pp. 6271, 2009.
[11] C. Flanagan, S. Freund, "FastTrack: efficient and precise dynamic race detection", *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121-133, 2009.
[12] C.J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering", 1987.
[13] K. Sen, "Race directed random testing of concurrent programs", *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 11-21, 2008.
[14] C.-S. Park, K. Sen, P. Hargrove and C. Iancu, "Efficient data race detection for distributed memory parallel programs", *Proceedings of 2011 International Conference for High Performance Computing*, Networking, Storage and Analysis, pp. 1-12, 2011.
[15] F. Bellard, "QEMU, a fast and portable dynamic translator", *Proceedings of the USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46, pp. 10-55, 2005.
[16] M. Musuvathi, S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs", *ACM Sigplan Notices*, vol. 42, no. 6, pp. 446-455, 2007.