Runtime Detection of Data Races in OCaml with ThreadSanitizer

Olivier Nicole
Tarides

Fabrice Buoro
Tarides

1 Introduction

The possibility to write truly parallel OCaml code brings forth new possibilities of bugs. Among those, data races (concurrent accesses to the same data) are hard to detect and dangerous, as they are non-deterministic, possibly silent, and can lead to highly unexpected results. ThreadSanitizer (TSan) is an open-source library and program instrumentation pass to reliably detect data races at runtime [1]. TSan has been instrumental in finding thousands of data races across many programming languages. We will describe the core principles of data race detection in TSan, explain why it was challenging to apply it to OCaml, and the adaptations needed to the runtime system. We plan to demo how you can already use it in your own code, and explain the limitations to be aware of.

2 Example Usage

An OCaml programmer wrote a piece of code that populates a table of clients from several sources (database files, say). To gain time, they make it multicore, using two `Domains` for two different data sources:

```
1 let clients = Hashtbl.create 16
2 let free_id = Atomic.make 0
3
4 let clients1 = (* Some data source *)
5
6 let clients2 = (* Some data source *)
7
8 let record_clients =
9   Seq.iter (fun c ->
10    Hashtbl.add
11      clients
12      (Atomic.fetch_and_add free_id 1)
13      c)
14
15 let () =
16   let d = Domain.spawn
17     (fun () -> record_clients clients1)
18   in
19   record_clients clients2;
20   Domain.join d
```

Each incoming client is bound to a unique ID. Our programmer was careful to use the `Atomic` module for ID generation, to make sure the IDs were really unique. Alas, they don't know that the `Hashtbl` module is not designed for concurrent use (instead, they should have used domain-safe structures from, e.g., the Saturn [2] library). Using a `Hashtbl.t` in parallel can cause data races and lead to surprising results. For instance, two domains adding elements in parallel can result in some elements being silently dropped. The resulting bugs will cost a lot of time to debug as they are non-deterministic. Worse still, the programmer's project may depend on libraries that use `Hashtbl`, making them possibly unsafe to use in parallel, without it being explicit in the documentation.

In contrast, if our developer builds their program on a special `opam` switch with a TSan-enabled compiler, all memory accesses will be instrumented with calls to the TSan runtime, which will detect data races:

```
$ opam switch create tsan-tests ocaml-
option-tsan
$ opam install dune
$ dune exec ./clients.exe
```

and while running it will output a data race report as shown in Listing 1.

TSan has detected two memory accesses, a write and a read, to the same memory location, that are not ordered. This constitutes a data race and TSan reports it, along with the backtraces of both accesses. Here, clearly, something is going on with the `Hashtbl` operations (called from line 9, highlighted), which is a serious hint for our developer.

3 How TSan Works

Executables are instrumented with calls to the TSan runtime library, which tracks accesses to shared data, and ordering relations established between these accesses (usually called "happens-before" relations). Internally the TSan runtime associates to each OCaml domain (i.e., each system thread) a vector clock. Comparing clocks allows to establish ordering between events [3]. A data race is reported every time two memory accesses are made to overlapping memory regions, and:

```
==================
WARNING: ThreadSanitizer: data race (pid=790576)
  Write of size 8 at 0x7f42b37f57e0 by main thread (mutexes: write M86):
    #0 caml_modify runtime/memory.c:166 (clients.exe+0x58b87d)
    #1 camlStdlib__Hashtbl.resize_749 stdlib/hashtbl.ml:152 (clients.exe+0x536766)
    #2 camlStdlib__Seq.iter_329 stdlib/seq.ml:76 (clients.exe+0x4c8a87)
    #3 camlDune__exe__Clients.entry /workspace_root/clients.ml:9 (clients.exe+0x4650ef)
    #4 caml_program <null> (clients.exe+0x45fefe)
    #5 caml_start_program <null> (clients.exe+0x5a0ae7)

  Previous read of size 8 at 0x7f42b37f57e0 by thread T1 (mutexes: write M90):
    #0 camlStdlib__Hashtbl.key_index_1308 stdlib/hashtbl.ml:507 (clients.exe+0x53a625)
    #1 camlStdlib__Hashtbl.add_1312 stdlib/hashtbl.ml:511 (clients.exe+0x53a6f8)
    #2 camlStdlib__Seq.iter_329 stdlib/seq.ml:76 (clients.exe+0x4c8a87)
    #3 camlStdlib__Domain.body_703 stdlib/domain.ml:202 (clients.exe+0x50bf60)
    #4 caml_start_program <null> (clients.exe+0x5a0ae7)
    #5 caml_callback_exn runtime/callback.c:197 (clients.exe+0x56917b)
    #6 caml_callback runtime/callback.c:293 (clients.exe+0x569cb0)
    #7 domain_thread_func runtime/domain.c:1100 (clients.exe+0x56d37f)
    [...]

SUMMARY: ThreadSanitizer: data race runtime/memory.c:166 in caml_modify
==================
[...]
ThreadSanitizer: reported 2 warnings
```

Listing 1: Example output of the program (truncated).

- at least one of them is a write, and
- there is no established happens-before relation between them.

In addition, each word of application memory is associated with a number of “shadow words”. Each shadow word contains information about a recent memory access to that word, including a scalar clock (projection of a vector clock). This information is maintained as a “shadow state” in a separate memory region, and updated at every (instrumented) memory access.

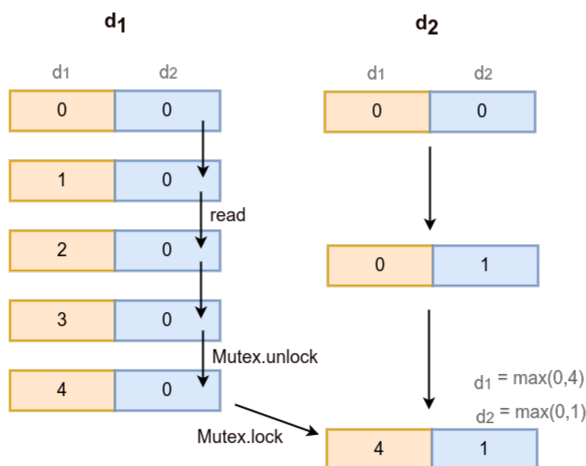

Figure 1: Each domain holds a vector clock, and increments its own clock upon every event (memory access, mutex operation...). Some operations synchronize clocks between domains.

In addition to memory accesses, operations such as `Domain.spawn` and `Domain.join`, or mutex operations, are also relevant for operation ordering, as illustrated in Figure 1, and therefore are also instrumented.

4 Challenges

Using ThreadSanitizer with OCaml thus requires support from the compiler, to instrument the executables. We have developed a version of the OCaml compiler that does just that. It instruments all memory accesses (except for immutable values, which cannot cause data races), and domain and mutex operations.

But, perhaps less expectedly, TSan also requires the compiler to instrument function entries and exits. This is required for TSan to be able to show backtrace of previous memory accesses—recall that TSan reports two backtraces, one for each conflicting access, one of which is in the past. TSan keeps a log of function entry and exit events in order to be able to reconstruct such backtraces when needed.

While instrumenting the entry point and return point of functions is straightforward, functions can also be exited due to an exception. And, since OCaml 5, control flow can also switch back and forth between an effect handler and the fiber that performed the effect [4]. To let TSan know about these function entries and exits, we take

the approach of emitting an instrumentation call for each exited or entered frame, by unwinding the stack upon every exception raising, effect performing, or resuming of a continuation.

Another challenging point is that TSan is designed to detect data races according to the memory model of C and C++, namely C11. OCaml's memory model is quite different. For instance, non-atomic accesses in OCaml have more ordering guarantees than non-atomic accesses in C11. Therefore, the instrumentation of memory accessses, conceptually, must map OCaml programs to C programs. This mapping must be such that racy programs (in the OCaml sense) must be mapped to racy programs (in the C11 sense) so that *OCaml data races are detected*; and race-free programs (in the OCaml sense) must be mapped to race-free C programs because *we don't want false positives*. We found that there exists in fact a mapping between the two models that has these good properties.

5 Status and Limitations

ThreadSanitizer support has been integrated into the OCaml compiler [5]. It has already allowed to find data races in the Saturn and Domainslib libraries and in the OCaml runtime itself. The instrumentation has a substantial performance cost: it incurs a slowdown in the range 2x–7x, and increases memory consumption by a factor of 4x–7x. These are however lower than the reported overheads for C/C++ (5x–15x for time and 5x–10x for space).

In C, C++ and Go, binaries instrumented with TSan can be freely linked with non-instrumented binaries; for OCaml programs, this compatibility property is lost due to the unwinding mechanism that records function entries and exits upon exceptions and effects, which assumes instrumentation of all traversed code. Future work could restore compatibility by recording which functions are instrumented and which are not; it would allow for easier deployment, since one would no longer have to build all libraries and C stubs with instrumentation.

6 Related Work

Static detection Static detection of data races can be baked into the language, akin to the borrow checker in Rust [6]. Such approaches must be adopted from the very start of a project. Another approach is to apply a static analysis to the code [7, 8]. Static analyses inevitably produce approximations, which can cause a number of false alarms or miss some data races.

Runtime detection Runtime detection tools such as ThreadSanitizer [1] are generally easier to apply in that they report very few false positives; but the performance cost that they incur can be an obstacle to deployment. Race detection based on causally-precedes relation [9], rather than vector clocks, can increase precision at the cost of a further increased overhead.

Interleaving exploration Another approach is to insert assertions in verified programs and test as many interleavings as possible [10, 11, 12]. The methods to explore more interleavings can be stochastic such as inserting thread yields or injecting random delays; or the system can explore all interleavings [11]. Dscheck [13] is a library for OCaml that replaces the standard `Atomic` module and explores all interleavings, using partial order reduction. While potentially discovering more bugs through exhaustive search, interleaving exploration has the drawback of requiring programmer assertions. Exhaustive search is also impractical on large-scale projects. Parafuzz [14] combines Quickcheck-style property-based testing, grey-box fuzzing—using AFL—for coverage-guided input generation, and randomization of the scheduling using the input from AFL. Neither Parafuzz nor Dscheck currently account for the possible out-of-order reads resulting from data races.

Automated test generation QCheck-Lin and QCheck-STM [15] take the approach of Quickcheck-style, random input generation to exercise the tested API in parallel on multiple domains. Unlike in the coverage-guided approach, the API is treated as a black box. QCheck-Lin simply checks that the test runs are linearizable (i.e., can be explained by a sequential interleaving). QCheck-STM goes further and checks that the outputs conform to a model provided by the programmer. The approach has allowed to find numerous concurrency bugs in the OCaml runtime and standard library. Such tests can be instrumented with TSan to help detect silent data races.

Bibliography

[1] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for Thread-

Sanitizer," in *Runtime Verification - Second Int. Conf. (RV '11)* in Lecture Notes in Computer Science, vol. 7186, 2011, pp. 110–114. [Online]. Available: https://doi.org/10.1007/978-3-642-29860-8/_9

[2] "Saturn — Lock-free data structures for multicore OCaml," Multicore OCaml, 2023. Accessed: Jun. 1, 2023. [Online]. Available: https://github.com/ocaml-multicore/saturn

[3] K. Joshi, ""go test -race" Under the Hood," in *Strange Loop, Qcon Sf*, San Francisco, 2016. [Online]. Available: https://www.youtube.com/watch?v=5erqWdlhQLA

[4] K. C. Sivaramakrishnan, S. Dolan, et al., "Retrofitting effect handlers onto OCaml," in *PLDI '21: 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation, Virtual Event, Canada, June 20-25, 2021*, 2021, pp. 206–221.

[5] O. Nicole, and F. Buoro, "Add ThreadSanitizer support," The OCaml compiler repository. https://github.com/ocaml/ocaml/pull/12114

[6] S. Klabnik, and C. Nichols, *The Rust Programming Language*, San Francisco: No Starch Press, 2019.

[7] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proc. 27th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI '06)*, Ottawa, Ontario, Canada, Jun. 11, 2006, pp. 308–319. [Online]. Available: https://dl.acm.org/doi/10.1145/1133981.1134018

[8] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "RacerD: compositional static race detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[9] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proc. 39th ACM SIGPLAN-SIGACT Symp. Princ. Program. Languages (POPL '12)*, 2012, pp. 387–400.

[10] K. Sen, "Race directed random testing of concurrent programs," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI '08)*, Tucson AZ USA, Jun. 7, 2008, pp. 11–21. [Online]. Available: https://dl.acm.org/doi/10.1145/1375581.1375584

[11] M. Musuvathi, "Systematic concurrency testing using CHESS," in *Proc. 6th Workshop Parallel Distrib. Systems: Testing, Analysis, Debugging (ISSTA '08)*, Seattle Washington, Jul. 20, 2008, p. 1. [Online]. Available: https://dl.acm.org/doi/10.1145/1390841.1390851

[12] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. (Oct. 27, 2019). Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. Presented at SOSP '19: ACM SIGOPS 27th Symp. Operating Syst. Princ. [Online]. Available: https://dl.acm.org/doi/10.1145/3341301.3359638

[13] "Dscheck — tool for testing concurrent OCaml programs," 2023. Accessed: Jun. 1, 2023. [Online]. Available: https://github.com/ocaml-multicore/dscheck

[14] S. Padhiyar, A. Kamath, and g.-i. family=Sivaramakrishnan given=KC, Parafuzz: Coverage-guided Property Fuzzing for Multicore OCaml programs. Presented at Ocaml Users Developers Workshop 2021, 2021.

[15] J. Midtgaard, O. Nicole, and N. Osborne, "Multicoretests – Parallel Testing Libraries for OCaml 5.0," in *Ocaml Users Developers Workshop 2022*, 2022.