ThreadSanitizer for OCaml

Olivier Nicole – Fabrice Buoro
(Tarides)
Goal of this talk

- What is ThreadSanitizer (TSan) and how is it useful?
- What is required to integrate the TSan runtime to OCaml programs?
- Hear your questions and suggestions about it
Finally, we can have data races too

A **data race** is a race condition defined by:

- Two accesses are made to the same memory location,
- At least one of them is a write, and
- No order is enforced between them.

Event ordering is formalized in terms of a partial order called *happens-before*. It is defined by the OCaml 5 memory model.

Data races are:

- Hard to detect (possibly silent)
- Hard to track down
ThreadSanitizer (TSan)

- **Runtime** data race detector (dynamic analysis, not static!)
- Initially developed for C++ by Google, now supported in
  - C, C++ with GCC and clang
  - Go
  - Swift
- **Battle-tested, already found:**
  - 1200+ races in Google’s codebase
  - ~100 in the Go stdlib
  - 100+ in Chromium
  - LLVM, GCC, OpenSSL, WebRTC, Firefox

- Requires to compile your program specially
Demo
module Exercise (Q : Queueable) = struct
  let exercise queue =
    for i = 0 to 4 do
      Format.printf "Adding %d\n!" i;
      Q.push i queue
    done

  let work () =
    let go = Atomic.make false in
    let q = Q.create () in
    let d = Domain.spawn (fun () -> Atomic.set go true; exercise q) in
    while not (Atomic.get go) do Domain.cpu_relax () done;
    exercise q;
    Domain.join d
  end

module Seq = Exercise (Queue)
module Par = Exercise (struct
  include Lockfree.Michael_scott_queue
  let push i q = Fun.flip push i q
end)

let () =
  print_endline "With a non domain-safe queue";
  Seq.work ();
  print_endline "With a domain-safe queue";
  Par.work ()
How does it work?
Two components

Program instrumentation

- Memory accesses
- Thread spawning and joining
- Mutex locks and unlocks, …

Runtime library
Race detector state machine

- **spawn**
- **lock**
- **read**

race detector

state

race detected

Race detector state machine
TSan’s internal state

- Each thread holds a **vector clock** (array of $N$ clocks, $N =$ number of threads)
- Each thread increments its clock upon every **event** (memory access, mutex operation…)
- Some operations (e.g. mutex locks, atomic reads) synchronize clocks between threads

Comparing vector clocks allows to establish **happens-before** relations.
Shadow state

Stores information about memory accesses.

8-byte shadow word for an access:

<table>
<thead>
<tr>
<th>TID</th>
<th>clock</th>
<th>pos</th>
<th>w</th>
</tr>
</thead>
</table>

- **TID**: accessor thread ID
- **clock**: scalar clock of accessor, optimized vector clock
- **pos**: offset, size
- **w**: is write

The shadow state stores $M$ shadow words per application word ($M \in [2, 7]$, default $M = 4$)

If shadow words are filled, evict one at random

Virtual memory

$\text{shadow} = M \times \text{addr} \& \text{mask}$

application

0x7fffffffffff

shadow

0x180000000000

0x7f0000000000
Race detection

Upon memory access, compare:

accessor’s clock with each existing shadow word

- do the accesses overlap?
- is one of them a write?
- are the thread IDs different?
- are they unordered by happens-before?
Race detection

Upon memory access, compare:

accessor’s clock with each existing shadow word

- do the accesses overlap?
- is one of them a write?
- are the thread IDs different?
- are they unordered by happens-before?

RACE
Race detection

Upon memory access, compare:

- accessor’s clock with each existing shadow word
- do the accesses overlap?
- is one of them a write?
- are the thread IDs different?
- are they unordered by happens-before?

RACE

Limitations:

- Runtime analysis: data races are only detected on visited code paths
- Finite number of memory accesses remembered ($M$ per memory word)
So what do we need to support TSan?
Instrumentation of memory accesses

fun () ->
  r := 10;
  let x = !r in
  g x
Instrumentation of memory accesses

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
  (store val r/503 21)

  (let x/514 (load_mut val r/503)
   (app{simple_race.ml:6,46-58} g/42 x/514 val))

  fun () ->
    r := 10;
    let x = !r in
    g x
```


Instrumentation of memory accesses

(fun () ->
  r := 10;
  let x = !r in
  g x

(app{simple_race.ml:6,46-58} g/42 x/514 val))
Instrumentation of memory accesses

\( \text{Instrumentation of memory accesses} \)

\[
\begin{align*}
&\text{(function{simple_race.ml:6,24-59} camlSimple_race.fun_521} \\
&(\text{param/513: val)} \\
&(\text{store val r/503 21}) \\
&(\text{let x/514 (load_mut val r/503)}) \\
&(\text{(app{simple_race.ml:6,46-58} g/42 x/514 val))}
\end{align*}
\]

\[
\begin{align*}
&\text{(function{simple_race.ml:6,24-59} camlSimple_race.fun_521} \\
&(\text{param/513: val)} \\
&(\text{let (newval/531 21 loc/530 r/503)} \\
&(\text{(extcall "__tsan_write8" loc/530 ->unit) 1}) \\
&(\text{(store val loc/530 newval/531)}) \\
&(\text{let x/514}) \\
&(\text{(let loc/533 r/503)} \\
&(\text{(extcall "__tsan_read8" loc/533 ->unit) 1}) \\
&(\text{(load_mut val loc/533)}) \\
&(\text{(app{simple_race.ml:7,47-59} g/42 x/514 val))}
\end{align*}
\]
Instrumentation of memory accesses

- In OCaml, writes are done through `caml_modify` (except for immediates), so it needs to be instrumented too.
- In general, runtime C functions that do significant things (memory accesses, thread operations…) need to be instrumented.
  - We use the built-in TSan support in gcc/clang to instrument them.
Function entries and exits

- Recall: TSan gives the backtrace of **both** conflicting accesses

```
WARNING: ThreadSanitizer: data race (pid=4170290)
Read of size 8 at 0x7f072b6f490 by thread T4 (mutexes: write M0):
#0 camlSimpleRace__fun_524 /tmp/simpleRace.ml:7 (simpleRace.exe+0x43cd9d)
#1 camlStdlib__Domain_body_696 /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/main/.
#2 caml_start_program ???:? (simpleRace.exe+0x4f51c3)
#3 caml_callback_exn /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint
#4 caml_callback /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint
#5 domain_thread_func /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint

Previous write of size 8 at 0x7f072b6f490 by thread T1 (mutexes: write M1):
#0 camlSimpleRace__fun_520 /tmp/simpleRace.ml:6 (simpleRace.exe+0x43dc45)
#1 camlStdlib__Domain_body_696 /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/main/.
#2 caml_start_program ???:? (simpleRace.exe+0x4f51c3)
#3 caml_callback_exn /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint
#4 caml_callback /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint
#5 domain_thread_func /home/olivier/.opam/5.0.8+tsan/.opam-switch/build/ocaml-variants.5.0.8+tsan/runtimes/x86_64-linux-gnu/lib/camlkids.so: Note: Unable to satisfy constraint

Mutex M0 (0x0000000567ad8) created at:
#0 pthread_mutex_init /home/olivier/other_projects/llvm-project/compiler-rt/lib/tsan/rtl/tsan_inter
[...]
SUMMARY: ThreadSanitizer: data race /tmp/simpleRace.ml:7 in camlSimpleRace__fun_524
```

ThreadSanitizer: reported 1 warnings
Function entries and exits

To be able to show backtraces of past program points, TSan requires us to instrument function entries and exits

Tail calls must be handled with care
Technical point #1.1 Exceptions

- In C, it is easy to instrument function entry and exits
- C++ has to take care of exceptions
- In OCaml also:
  - Any function can be exited due to an exception
  - Unlike in C++, exceptions do not unwind the stack

- TSan’s linear view of the call stack does not hold.
Technical point #1.1 Exceptions

```ocaml
let i () = raise MyExn

let h () = i()

let g () = print_and_call_ocaml_h()

let f () = try g () with |
| MyExn -> race()

let () = let d = Domain.spawn (fun () -> race()) in f(); Domain.join d
```

```ocaml
value print_and_call_ocaml_h(value unit)
{
  printf("Hello from C\n");
  caml_callback(*caml_named_value("h"), Val_unit);
  return Val_unit;
}
```
let i () = raise MyExn
let h () = i ()

let g () = print_and_call_ocaml_h ()

let f () = try g () with
  | MyExn -> race ()

let () = let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d

- **Cmm** instrumentation emits call to `tsan_func_entry` when entering a function
- TSan backtrace:
  - f
  - g
let i () = raise MyExn
let h () = i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
let i () = raise MyExn
let h () = i ()
let g () = print_and_call_ocaml_h()
let f () = try g () with | MyExn -> race ()

let () = let d = Domain.spawn (fun () -> race ()) in f (); Domain.join d
- For TSan, we are still in `f / g / print_and_call_ocaml_h / h / i`
  - Calling the race function of the exception handler without any other prior actions would result in an incorrect backtrace.

```ocaml
let i () = raise MyExn
let h () = i ()
let g () = print_and_call_ocaml_h()
let f () = try g () with
  | MyExn -> race()
let () = let d = Domain.spawn (fun () -> race()) in
  f();
  Domain.join d
```
let i () = raise MyExn
let h () = i ()
let g () = print_and_call_ocaml_h ()
let f () = try g () with | MyExn -> race ()

let () = let d = Domain.spawn (fun () -> race ()) in f (); Domain.join d

- For TSan, we are still in f / g / print_and_call_ocaml_h / h / i
  - Calling the race function of the exception handler without any other prior actions would result in an incorrect backtrace
- While raising the exception, in caml_raise_exn
  - Use frame_descr to scan the stack up to the next exception handler
  - Emit tsan_func_exit for every stack frame
- For TSan, we are still in `f / g / print_and_call_ocaml_h`
- The exception propagates through the C stack, `frame_descr` can't help here
- In `caml_raise`
  - Use `libunwind` to scan the stack up to the next handler
  - Emit `tsan_func_exit` for every C stack frame

```ocaml
let i () = raise MyExn
let h () = i ()
let g () = print_and_call_ocaml_h ()
let f () = try g () with
  | MyExn -> race ()

let () = let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```
- Again in the OCaml stack
- The process repeats: back to using `frame_descr` in `caml_raise_exn` to emit `tsan_func_exit` until the exception handler (in function `f`)

```ocaml
define i () = raise MyExn
define h () = i ()
define g () = print_and_call_ocaml_h ()
define f () = try g () with
  | MyExn -> race ()
define () = let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```
Technical point #1.2 Effect handlers

- Effect handlers are like exceptions, except you can come back

```ocaml
type _ Effect.t += E : string Effect.t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
        print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```

https://kcsrk.info/slides/retro_effects_simcorp.pdf
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
        | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 ";
          continue k "2 ";
          print_string "4 ")
        | _ -> None);
    exnc = (fun e -> raise e); }
- **main** calls `Effect.match_with`
  - Allocates a new fiber
  - Switches to the stack into fiber #1
  - Executes the computation (through `caml_runstack`)

```ocaml
let comp () =
    print_string "0 ";
    print_string (perform E);
    print_string "3 ";

let main () =
    match_with comp () { retc = Fun.id; effc = (fun (type a) (eff : a Effect.t) ->
        match eff with
        | E -> Some (fun (k : (a, unit) continuation) ->
            print_string "1 "; continue k "2 "; print_string "4 ";
        | _ -> None);
        exnc = (fun e -> raise e); }
```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
        print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }

● Perform the E effect
● caml_perform
  ○ In order to resume execution into the effect handler of fiber #0
  ○ Use frame_descr to emit calls to tsan_func_exit
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
         print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
        print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }

- Calls continue to resume execution in the computation
- `caml_resume`
  - In order to resume execution in the fiber #1 stack
  - Use `frame_descr` to emit calls to `tsan_func_entry`
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 ";

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
        print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }

● The computation completes
● caml_runstack
  ○ Free the fiber
  ○ Resume execution in the initial fiber
  ○ Call the value handler
let comp () =
  print_string "0 \n";
  print_string (perform E);
  print_string "3 \n"

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
            print_string "1 \n"; continue k "2 \n"; print_string "4 \n")
      | _ -> None);
    exnc = (fun e -> raise e); }

• Completes the effect handler and so the **match_with**
Technical point #2: Memory model

- TSan understands the **C11 memory model**
- The OCaml 5 memory model is quite different

We map OCaml memory accesses to C11 accesses. The 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
- Race-free programs (in the OCaml sense) must be mapped to race-free programs (in the C11 sense) as we don’t want false positives

⇒ What we “show” to TSan is not necessarily the real memory operations.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Location in the codebase</th>
<th>Implementation</th>
<th>TSan view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic load</td>
<td><code>caml_atomic_load</code></td>
<td><code>fence(acquire)</code>; <code>atomic_load(seq_cst)</code></td>
<td><code>atomic_load(seq_cst)</code></td>
</tr>
<tr>
<td>Atomic store</td>
<td><code>caml_atomic_exchange</code></td>
<td><code>fence(acquire)</code>; <code>atomic_exchange(seq_cst)</code>; <code>fence(release)</code></td>
<td><code>atomic_exchange(seq_cst)</code></td>
</tr>
<tr>
<td>Non-atomic load</td>
<td>assembly</td>
<td><code>atomic_load(relaxed)</code></td>
<td>plain load</td>
</tr>
<tr>
<td>Non-atomic store (initializing)</td>
<td>assembly or <code>caml_initialize</code></td>
<td>plain store</td>
<td>-</td>
</tr>
<tr>
<td>Non-atomic store (assignment, integer)</td>
<td>assembly or <code>caml_modify</code></td>
<td><code>fence(acquire)</code>; <code>atomic_store(release)</code></td>
<td>plain store</td>
</tr>
<tr>
<td>Non-atomic store (assignment, pointer)</td>
<td>assembly or <code>caml_modify</code></td>
<td><code>fence(acquire)</code>; <code>atomic_store(release)</code></td>
<td>plain store</td>
</tr>
<tr>
<td>Non-atomic store (non-word-sized field)</td>
<td>assembly</td>
<td>plain store</td>
<td>plain store</td>
</tr>
<tr>
<td>Operation</td>
<td>Location in the codebase</td>
<td>Implementation</td>
<td>TSan view</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------</td>
<td>------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Atomic load</td>
<td>caml_atomic_load</td>
<td>fence(acquire)</td>
<td>atomic_load(seq_cst)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atomic_load(seq_cst)</td>
<td></td>
</tr>
<tr>
<td>Atomic store</td>
<td>caml_atomic_exchange</td>
<td>fence(acquire)</td>
<td>atomic_exchange(seq_cst)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atomic_exchange(seq_cst)</td>
<td></td>
</tr>
<tr>
<td>Non-atomic load</td>
<td>assembly</td>
<td>atomic_load(relaxed)</td>
<td>plain load</td>
</tr>
<tr>
<td>Non-atomic store (initializing)</td>
<td>assembly or caml_initialize</td>
<td>plain store</td>
<td>-</td>
</tr>
<tr>
<td>Non-atomic store (assignment, integer)</td>
<td>assembly or caml_modify</td>
<td>fence(acquire)</td>
<td>plain store</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atomic_store(release)</td>
<td></td>
</tr>
<tr>
<td>Non-atomic store (assignment, pointer)</td>
<td>assembly or caml_modify</td>
<td>fence(acquire)</td>
<td>plain store</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atomic_store(release)</td>
<td></td>
</tr>
<tr>
<td>Non-atomic store (non-word-sized field)</td>
<td>assembly</td>
<td></td>
<td>plain store</td>
</tr>
</tbody>
</table>
Current status

- The instrumentation has a performance cost: about 7-13x slowdown
  - compared to 5-15x for C/C++
- Memory consumption is increased by 2-7x (compared to 5-10x for C/C++)
- No cost if TSan is not enabled on your opam switch
- An earlier version based on OCaml 5.0 is already available on opam:
  \texttt{opam switch create 5.0.0+tsan}
- We have already used the mode to find races in
  - Lockfree: \url{ocaml-multicore/lockfree#40}, \url{ocaml-multicore/lockfree#39}
  - Domainslib: \url{ocaml-multicore/domainslib#72}, \url{ocaml-multicore/domainslib#103}
  - The OCaml runtime: \url{ocaml/ocaml#11040}
- A feature complete PR is ready: \url{ocaml/ocaml#12114}
  - \(\approx\)1,700 lines of diff + 1,000 lines of test suite
  - No full review yet
Thank You
Backup slide #1: scalar clocks vs vector clocks

Optimization 2

<table>
<thead>
<tr>
<th>TID</th>
<th>clock</th>
<th>pos</th>
<th>wr</th>
</tr>
</thead>
</table>

scalar clock, not full vector clock.

g_x access:

3 3 2

Credits: go test -race Under the Hood
Backup slide #1: scalar clocks vs vector clocks

- **g1**: count == 0
- **race read(...)**
- **g1**: count++
- **race write(...)**
- **g2**: count == 0
- **race read(...)**

by compiler instrumentation

and check for race