Runtime Detection of Data Races in OCaml with ThreadSanitizer

Olivier Nicole Fabrice Buoro

2023-09-11

Tarides

Goal of this talk

- What is ThreadSanitizer (TSan) and how is it useful?
- What is required to integrate TSan to OCaml programs?

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

Data race example

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

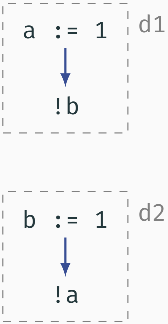
Data race example

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```



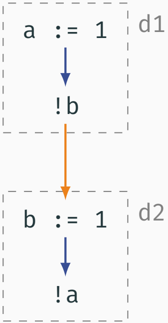
Data race example

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```



a := 1

!b

d1

b := 1

!a

d2

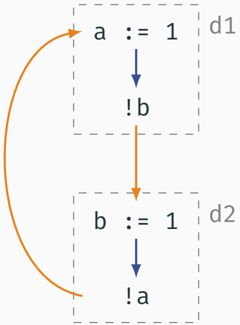
Data race example

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```



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[1]
  - 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

---

[1] Numbers August 2015

```ocaml
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

```ocaml
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

```
WARNING: ThreadSanitizer: data race (pid=3808831)
  Write of size 8 at 0x8febe0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:7 (simple_race.exe+0x420a72)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous read of size 8 at 0x8febe0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:4 (simple_race.exe+0x420a22)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

WARNING: ThreadSanitizer: data race (pid=3808831)
  Read of size 8 at 0x8febf0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:8 (simple_race.exe+0x420a92)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous write of size 8 at 0x8febf0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:3 (simple_race.exe+0x420a01)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

==================
ThreadSanitizer: reported 2 warnings
```

```ocaml
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

```
WARNING: ThreadSanitizer: data race (pid=3808831)
  Write of size 8 at 0x8febe0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:7 (simple_race.exe+0x420a72)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous read of size 8 at 0x8febe0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:4 (simple_race.exe+0x420a22)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

WARNING: ThreadSanitizer: data race (pid=3808831)
  Read of size 8 at 0x8febf0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:8 (simple_race.exe+0x420a92)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous write of size 8 at 0x8febf0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:3 (simple_race.exe+0x420a01)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

==================
ThreadSanitizer: reported 2 warnings
```

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a

let () =
  let h = Domain.spawn d2 in
  let r1 = d1 () in
  let r2 = Domain.join h in
  assert (not (r1 = 0 && r2 = 0))
```

```
WARNING: ThreadSanitizer: data race (pid=3808831)
  Write of size 8 at 0x8febe0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:7 (simple_race.exe+0x420a72)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous read of size 8 at 0x8febe0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:4 (simple_race.exe+0x420a22)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

WARNING: ThreadSanitizer: data race (pid=3808831)
  Read of size 8 at 0x8febf0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:8 (simple_race.exe+0x420a92)
    #1 camlDomain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x440f2f)
    #2 caml_start_program <null> (simple_race.exe+0x47cf37)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x445f7b)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a113)

  Previous write of size 8 at 0x8febf0 by main thread (mutexes: write M86):
    #0 camlSimple_race.d1_271 simple_race.ml:3 (simple_race.exe+0x420a01)
    #1 camlSimple_race.entry simple_race.ml:13 (simple_race.exe+0x420d16)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47cf37)
[...]

==================
ThreadSanitizer: reported 2 warnings
```

```
let d1 () =
  Mutex.lock m;
  a := 1;
  let res = !b in
  Mutex.unlock m;
  res

let d2 () =
  Mutex.lock m;
  b := 1;
  let res = !a in
  Mutex.unlock m;
  res
```

How TSan works

Two components

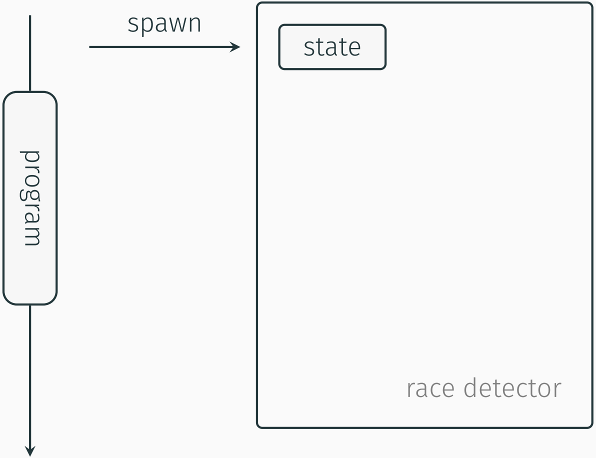
Program instrumentation

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

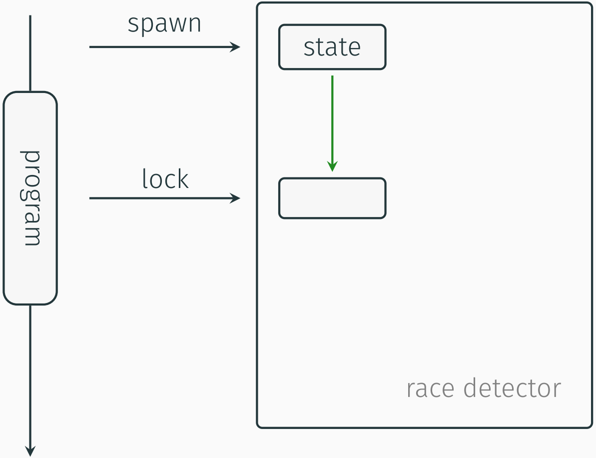
call →

Runtime library

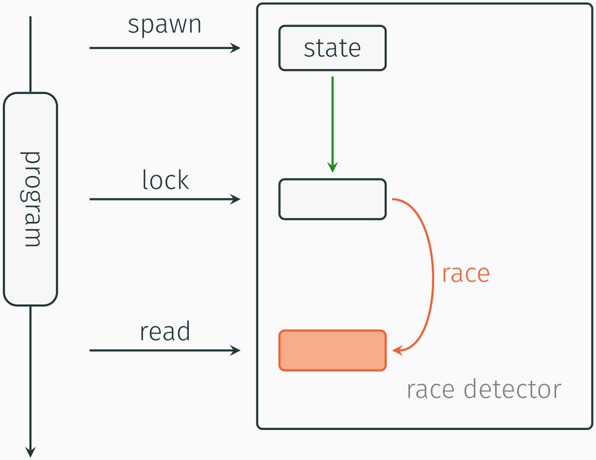
Race detector state machine



Race detector state machine



Race detector state machine



```
let d1 () =
  Mutex.lock m;
  a := 1;
  let res = !b in
  Mutex.unlock m;
  res

let d2 () =
  Mutex.lock m;
  b := 1;
  let res = !a in
  Mutex.unlock m;
  res
```

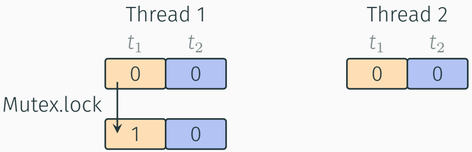
TSan’s internal state

- Each thread holds a **vector clock** (array of N clocks, N = number of threads)

	t_1	t_2
Thread 1	0	0
Thread 2	0	0

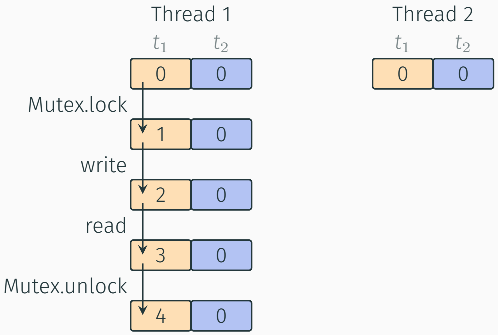
TSan's internal state

- 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…)



TSan's internal state

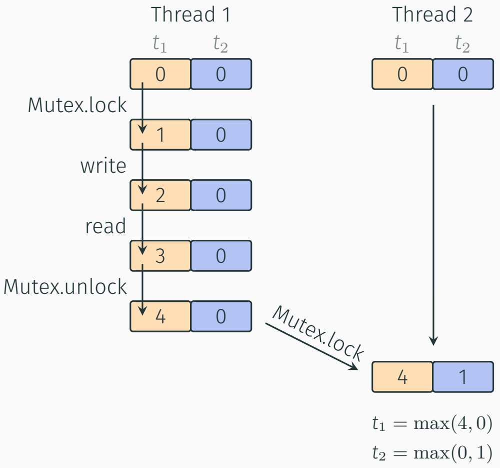
- Each thread holds a **vector clock** (array of N clocks, N = number of threads)
- Each thread increments its clock upon every **event** (memory access, mutex operation...)



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.

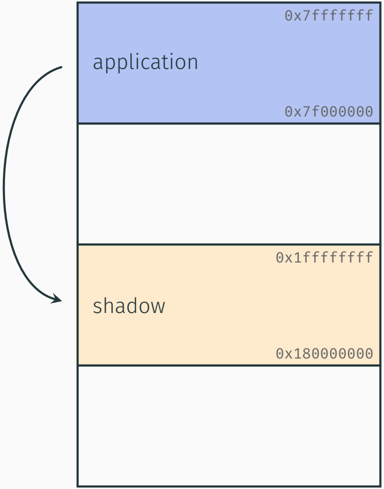


Shadow state

- Stores information about memory accesses.
- 8-byte shadow word for an access:

TID	clock	pos	w

  - **TID**: accessor thread ID
  - **clock**: scalar clock of accessor, optimized vector clock
  - **pos**: offset, size
  - **w**: is write
- If shadow words are filled, evict one at random



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 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 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

So what do we need to support TSan?

Instrumentation of memory accesses

```
let d1 () =
  a := 1;
  !b
```

```
(function d1 (param)

 (store a 1)


 (load_mut b))
```

Instrumentation of memory accesses

```
let d1 () =
  a := 1;
  !b
```

```
(function d1 (param)

 (store a 1)


 (load_mut b))
```

```
(function d1 (param)
 (extcall "__tsan_write8" a)
 (store a 1)

 (extcall "__tsan_read8" b)
 (load_mut b))
```

Function entries and exits

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

```
==================
WARNING: ThreadSanitizer: data race (pid=3080294)
  Write of size 8 at 0x7f70feffebe0 by thread T1 (mutexes: write M90):
    #0 camlSimple_race.d2_274 simple_race.ml:7 (simple_race.exe+0x420a72)
    #1 camlStdlib__Domain.body_706 stdlib/domain.ml:211 (simple_race.exe+0x44119f)
    #2 caml_start_program <null> (simple_race.exe+0x47d1a7)
    #3 caml_callback_exn runtime/callback.c:197 (simple_race.exe+0x4461eb)
    #4 domain_thread_func runtime/domain.c:1167 (simple_race.exe+0x44a383)

  Previous read of size 8 at 0x7f70feffebe0 by main thread (mutexes: write M86):
    #0 camlSimple_race.main_277 simple_race.ml:13 (simple_race.exe+0x420b36)
    #1 camlSimple_race.entry simple_race.ml:34 (simple_race.exe+0x420fcf)
    #2 caml_program <null> (simple_race.exe+0x41ffb9)
    #3 caml_start_program <null> (simple_race.exe+0x47d1a7)
[...]
```

Function entries and exits

```
let d1 () =
  a := 1;
  !b
```

```
(function d1 (param)

 (extcall "__tsan_write8" a)
 (store a 1)

 (extcall "__tsan_read8" b)
 (load_mut b))
```

- 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

Function entries and exits

```
let d1 () =
  a := 1;
  !b
```

```
(function d1 (param)

 (extcall "__tsan_write8" a)
 (store a 1)

 (extcall "__tsan_read8" b)
 (load_mut b))
```

```
(function d1 (param)
 (extcall "__tsan_func_entry" return_addr)
 (extcall "__tsan_write8" a)
 (store a 1)

 (extcall "__tsan_read8" b)
 (let res (load_mut b)
   (extcall "__tsan_func_exit")
   res))
```

- 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

A first challenge: exceptions

- In C, it is easy to instrument function entries and exits
- C++ has to take care of exceptions
- OCaml has exceptions too:
    - Any function can be exited due to an exception
    - Unlike in C++, exceptions do not unwind the stack[1]
- TSan's linear view of the call stack does not hold

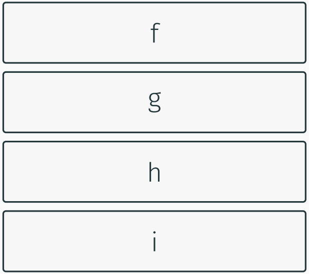
---

[1] Fabrice Buoro, "OCaml behind the scenes: exceptions"

A first challenge: exceptions

```
let race () = (* ... *)

let i () = raise Exit  ←
let h () = i ()
let g () = h ()
let f () =
  try g () with | Exit → race ()
```

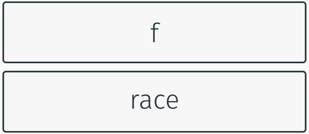


- TSan backtrace:
  - i
  - h
  - g
  - f

A first challenge: exceptions

```
let race () = (* ... *)

let i () = raise Exit
let h () = i ()
let g () = h ()
let f () =
  try g () with | Exit → race () ⟵
```


f

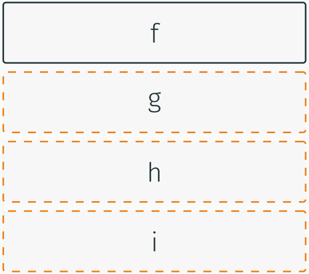
race

- TSan backtrace:
  - race
  - i
  - h
  - g
  - f

A first challenge: exceptions

```
let race () = (* ... *)

let i () = raise Exit ←
let h () = i ()
let g () = h ()
let f () =
  try g () with | Exit → race ()
```

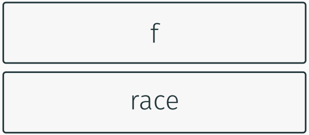


- TSan backtrace:
  - f

A first challenge: exceptions

```
let race () = (* ... *)

let i () = raise Exit
let h () = i ()
let g () = h ()
let f () =
  try g () with | Exit → race ()
```



f

race

- TSan backtrace:
  - race
  - f

A new challenger has arrived: Effect handlers

- Effect handlers are a **generalisation of exceptions**: **perform**-ing an effect jumps to the associated effect handler, and additionally, a delimited continuation makes it possible to **resume** a computation [1]
- As with exceptions, we must signal to TSan the frames that are **exited** when an effect is **performed**, and **re-entered** when a continuation is **resumed**

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

let () =
 match_with comp () {
 retc = Fun.id;
 effc = (fun eff ->
  match eff with
  | E -> Some (fun k ->
   print_string "1"; continue k "2";
   ↪   print_string "4")
  | _ -> None);
 exnc = raise; }
```

---

[1] KC Sivaramakrishnan et al, Retrofitting Effect Handlers onto OCaml, PLDI 2021

Final boss: The OCaml memory model

- TSan can detect data races in programs following the C11 memory model
- OCaml's memory model[1] is different from the C11 one
  - It offers more guarantees, such as **Local Data Race Freedom implies Sequential Consistency** (LDRF-SC)
- To enforce the OCaml memory model, some operations are implemented particularly, and special instructions are inserted in the code
  - **Bounding data race in space and time** (LDRF-SC) requires fences at strategic positions
  - OCaml's runtime, written in C, use strong instructions to prevent **Undefined Behavior** at C level

---

[1] Dolan et al., *Bounding Data Races In Space and Time*, PLDI 2018

Final boss: The OCaml memory model

OCaml

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a
```

✓ Well-defined behavior

C analogous

```
int a = 0, b = 0;

int d1() {
  a = 1;
  return b;
}

int d2() {
  b = 1;
  return a;
}
```

✗ Undefined behavior

Final boss: The OCaml memory model

OCaml

```
let a = ref 0 and b = ref 0

let d1 () =
  a := 1;
  !b

let d2 () =
  b := 1;
  !a
```

✓ Well-defined behavior

C analogous

```
int a = 0, b = 0;

int d1() {
  atomic_store_release(&a, 1);
  return atomic_load_acquire(&b);
}

int d2() {
  atomic_store_release(&b, 1);
  return atomic_load_acquire(&a);
}
```

✓ Well-defined behavior

Final boss: The OCaml memory model

- TSan will not detect data races on C11 atomic operations
- We do not signal the “real” operations to TSan
- Instead, we map OCaml memory operations to C11 memory operations so that TSan detects OCaml data races.

Current status

- Performance cost: about 2-7x slowdown (compared to 5-15x for C/C++)
- Memory consumption is increased by 4-7x (compared to 5-10x for C/C++)
- Only supported on 64 bits, non-Windows (TSan limitations), only x86_64 for now

Conclusion

- Merged in trunk, will be released with OCaml 5.2
- For convenience, there is a backport on OCaml 5.1 (currently rc3):

```
sudo apt install libunwind-dev
opam switch create 5.1.0~rc3+tsan
```

- We have used TSan to find races in
  - Lockfree: ocaml-multicore/lockfree#40, ocaml-multicore/lockfree#39
  - Domainslib: ocaml-multicore/domainslib#72, ocaml-multicore/domainslib#103
  - The OCaml runtime: ocaml/ocaml#11040
- TSan has also been helpful to Irmin and Eio
- User feedback welcome

Acknowledgements

- Thanks to Anmol Sahoo, Jacques-Henri Jourdan, Luc Maranget, Guillaume Munch-Maccagnoni, and to the reviewers of the TSan PR
- The first slides contain some information and diagrams from the great talk *"go test -race" Under the Hood* by Kavya Joshi at Strange Loop 2016

Conclusion

- Merged in trunk, will be released with OCaml 5.2
- For convenience, there is a backport on OCaml 5.1 (currently rc3):

```
sudo apt install libunwind-dev
opam switch create 5.1.0~rc3+tsan
```

- We have used TSan to find races in
  - Lockfree: ocaml-multicore/lockfree#40, ocaml-multicore/lockfree#39
  - Domainslib: ocaml-multicore/domainslib#72, ocaml-multicore/domainslib#103
  - The OCaml runtime: ocaml/ocaml#11040
- TSan has also been helpful to Irmin and Eio
- User feedback welcome

Thanks!

Acknowledgements

- Thanks to Anmol Sahoo, Jacques-Henri Jourdan, Luc Maranget, Guillaume Munch-Maccagnoni, and to the reviewers of the TSan PR
- The first slides contain some information and diagrams from the great talk *“go test -race” Under the Hood* by Kavya Joshi at Strange Loop 2016