Verified multi-word compare-and-set and software transactional memory for OCaml 5

ANONYMOUS AUTHOR(S)

OCaml has recently acquired the ability to have multiple domains running in parallel. Libraries like Eio [Madhavapeddy and Leonard [n. d.]] and Domainslib [Sivaramakrishnan [n. d.]] utilize OCaml 's support for algebraic effects to provide lightweight threads of control. But, while threads are a prerequisite for concurrent programming, we also need mechanisms for threads to communicate and synchronize, such as message queues, mutexes and condition variables. However, such mechanisms do not compose and can be challenging to use. Transactional memory [Shavit and Touitou 1995] is a more recent abstraction that offers both a relatively familiar programming model and composability.

We present the Kcas library, a software transactional memory implementation for OCaml based on a state-of-the-art multi-word compare-and-set algorithm [Guerraoui, Kogan, Marathe and Zablotchi 2020] enhanced with optimized read-only operations. Kcas features a convenient direct style interface and comes with a library of composable, reasonably well performing, concurrent data structures. It supports scheduler friendly blocking and timeouts.

We formally verify the core multi-word compare-and-set algorithm using the Iris [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018] concurrent separation logic. This effort involves advanced proof techniques, including logical atomicity [Jacobs and Piessens 2011] and prophecy variables [Jung, Lepigre, Parthasarathy, Rapoport, Timany, Dreyer and Jacobs 2020].

1 INTRODUCTION

1.1 Software transactional memory

Since its introduction in 1995 [Shavit and Touitou 1995], *software transactional memory* (STM) has been implemented in several programming languages, including Haskell [Harris, Marlow, Jones and Herlihy 2005], Scala [Goes and the ZIO Contributors [n. d.]] and C++ [Granin [n. d.]].

```
type ('k, 'v) cache =
  { mutable space: int;
    table: ('k, 'k Dllist.node * 'v) Hashtbl.t;
    order: 'k Dllist.t;
  }

let get_opt { table; order; _ } key =
  Hashtbl.find_opt table key
  |> Option.map @@ fun (node, value) ->
     Dllist.move_l node order ; value

type ('k, 'v) cache =
  { space: int Loc.t;
    table: ('k, 'k Dllist.Xt.node * 'v) Hashtbl.Xt.t;
    order: 'k Dllist.Xt.t;
  }

let get_opt ~xt { table; order; _ } key =
  Hashtbl.Xt.find_opt ~xt table key
  |> Option.map @@ fun (node, value) ->
     Dllist.Xt.move_l ~xt node order ; value
```

1.2 Multi-word compare-and-set algorithm

Our implementation of STM, as described in more detail in Section 3, relies on the *multi-word compare-and-set* (MCAS) algorithm, a generalization of the *single-word compare-and-set* (CAS) primitive: given a set of distinct shared-memory locations, each associated to an expected value and a desired value, this algorithm atomically either 1) updates all locations from expected to desired value and succeeds or 2) observes an unexpected value at some location and fails. However, while CAS is supported by most architectures, the multi-word variant has to be implemented at the software level.

It has been shown in the literature that MCAS can be made practical [Harris, Fraser and Pratt 2002]. Recent work [Guerraoui, Kogan, Marathe and Zablotchi 2020] further demonstrated that it can be implemented using only $k + 1$ (where k is the number of shared-memory locations) CAS in the uncontended case.

When building a transaction, loads, stores and more complex memory operations to be committed together atomically are translated to a MCAS operation. For instance, consider the two following transactions, involving locations a, b, x and y:

```
let x_to_b_sub_a ~xt () =
  let a = Xt.get ~xt a
  let b = Xt.get ~xt b in
  Xt.set ~xt x (b - a)
```

```
let y_to_a_add_b ~xt () =
  let a = Xt.get ~xt a
  let b = Xt.get ~xt b in
  Xt.set ~xt y (a + b)
```

Initially, a is set to 10, b to 52, x and y to 0. When they are committed, these transactions essentially correspond to the following MCAS operations:

```
CAS (a, 10, 10)
CAS (b, 52, 52)
CAS (x, 0, 42)
```

```
CAS (a, 10, 10)
CAS (b, 52, 52)
CAS (y, 0, 62)
```

CAS with equal expected and desired values essentially expresses an operation that does not change the logical content of the target location, but only "asserts" that it does not change during the operation.

One might then attempt to perform both MCAS operations in parallel. Unfortunately, this is not allowed by the MCAS implementations we are aware of. Indeed, every CAS actually updates the target locations. This means two things: 1) CAS operations targeting the same location can only execute sequentially; 2) CAS operations, even those that do not change the logical content of a location, cause contention as after the operation only the cache of the writer will have a valid copy of the location.

To address this issue, we extend upon the state-of-the-art algorithm [Guerraoui, Kogan, Marathe and Zablotchi 2020] to allow read-only CMP operations to be expressed directly and not write into memory. For instance, the two above transactions would generate the following operations, that can be run in parallel:

```
CMP (a, 10)
CMP (b, 52)
CAS (x, 0, 42)
```

```
CMP (a, 10)
CMP (b, 52)
CAS (y, 0, 62)
```

There is one drawback, however. This new algorithm is *obstruction-free* but not *lock-free* like the original one. In particular, two MCAS involving a common location may basically cancel each other indefinitely. To get the best of both worlds, we first attempt the MCAS operations in obstruction-free mode (CMP and CAS) and switch to lock-free mode (CAS only) after a number of failed attempts. The resulting algorithm therefore guarantees lock-free behavior.

```
module Loc : sig
  type !'a t
  val make : ?padded:bool -> ?mode:Mode.t -> 'a -> 'a t
  val get : 'a t -> 'a
  val set : 'a t -> 'a -> unit
  val compare_and_set : 'a t -> 'a -> 'a -> bool
  val exchange : 'a t -> 'a -> 'a
  val fetch_and_add : int t -> int -> int
  val incr : int t -> unit
  val decr : int t -> unit
end
```

Fig. 1. Interface for shared-memory locations (excerpt)

```
module Xt : sig
  type 'x t
  val get : xt:'x t -> 'a Loc.t -> 'a
  val set : xt:'x t -> 'a Loc.t -> 'a -> unit
  val update : xt:'x t -> 'a Loc.t -> ('a -> 'a) -> 'a
  val modify : xt:'x t -> 'a Loc.t -> ('a -> 'a) -> unit
  val exchange : xt:'x t -> 'a Loc.t -> 'a -> 'a
  val swap : xt:'x t -> 'a Loc.t -> 'a Loc.t -> unit
  val compare_and_set : xt:'x t -> 'a Loc.t -> 'a -> 'a -> bool
  val compare_and_swap : xt:'x t -> 'a Loc.t -> 'a -> 'a -> 'a
  val fetch_and_add : xt:'x t -> int Loc.t -> int -> int
  val incr : xt:'x t -> int Loc.t -> unit
  val decr : xt:'x t -> int Loc.t -> unit

  val post_commit : xt:'x t -> (unit -> unit) -> unit
  val validate : xt:'x t -> 'a Loc.t -> unit

  type 'a tx = { tx : 'x. xt:'x t -> 'a } [@@unboxed]
  val call : xt:'x t -> 'a tx -> 'a
  val commit : ?timeoutf:float -> ?mode:Mode.t -> 'a tx -> 'a
end
```

Fig. 2. Interface for explicit transaction logs (excerpt)

1.3 Verified multi-word compare-and-set algorithm

1.4 Contributions

2 OVERVIEW OF THE LIBRARY

2.1 Shared memory locations

2.2 Explicit transaction logs

transactional API

A transaction is a specification for generating a list of CAS

```
type 'a loc =
  { atomic state: 'a state;
    id: int;
  }
and 'a state =
  { casn: 'a casn;
    mutable before: 'a;
    mutable after: 'a;
  }
and 'a cas =
  { loc: 'a loc;
    state: 'a state;
  }
```

```
and 'a casn =
  { atomic status: 'a status;
    proph: (ghost_id * bool) proph;
  }
and 'a status =
  | Undetermined of 'a cas list
  | Before
  | After
```

Fig. 3. Type definitions for implementing multi-word compare-and-set

```
let transfer ~xt () =
  let a' = Xt.get ~xt a
  and b' = Xt.get ~xt b in
  Xt.set ~xt s (a' + b')
Xt.commit { tx = transfer }
```

2.3 Standard concurrent data structures

3 IMPLEMENTATION OF THE LIBRARY

4 VERIFIED MULTI-WORD COMPARE-AND-SET ALGORITHM

4.1 Specification

4.2 Resource algebras

4.3 Proof sketch

4.4 Support for read-only operations

4.5 Physical equality taken seriously

5 BENCHMARKS

6 RELATED WORK

7 CONCLUSION

```
let finish gid casn status =
  match casn.status with
  | Before -> false
  | After -> true
  | Undetermined _ as old_status ->
      resolve (
        Atomic.Loc.compare_and_set [%atomic.loc casn.status] old_status status
      ) casn.proph (gid, status == After) |> ignore ;
      casn.status == After

let rec determine_as casn cass =
  let gid = ghost_id in
  match cass with
  | [] ->
      finish gid casn After
  | cas :: cass' ->
      let { loc; state } = cas in
      let state' = loc.state in
      if state == state' then
        determine_as casn cass'
      else
        let v = get_as state' in
        if get_as state' != state.before then
          finish gid casn Before
        else
          match casn.status with
          | Before -> false
          | After -> true
          | Undetermined _ ->
              if Atomic.Loc.compare_and_set [%atomic.loc loc.state] state' state
              then determine_as casn cass'
              else determine_as casn cass
and get_as state =
  if determine state.casn then state.after else state.before
and determine casn =
  match casn.status with
  | Before -> false
  | After -> true
  | Undetermined cass -> determine_as casn cass
```

Fig. 4. Implementation of multi-word compare-and-set (1)

```
let make v id =
  let _gid = ghost_id in
  let casn = { status= After; proph= proph } in
  let state = { casn; before= v; after= v } in
  Atomic.make { state; id }

let get loc =
  get_as loc.state

let cas cass =
  let casn = { status= After; proph= proph } in
  let cass =
    Lst.map cass (fun (loc, before, after) ->
      let state = { casn; before; after } in
      { loc; state }
    )
  in
  casn.status <- Undetermined cass ;
  determine_aux casn cass
```

Fig. 5. Implementation of multi-word compare-and-set (2)

REFERENCES

John A. De Goes and the ZIO Contributors. [n. d.]. *ZIO*. https://github.com/zio/zio

Alexander Granin. [n. d.]. *cpp_stm_free*. https://github.com/graninas/cpp_stm_free

Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient Multi-Word Compare and Swap. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:19. https://doi.org/10.4230/LIPICS.DISC.2020.4

Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw (Eds.). ACM, 48–60. https://doi.org/10.1145/1065944.1065952

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings (Lecture Notes in Computer Science, Vol. 2508)*, Dahlia Malkhi (Ed.). Springer, 265–279. https://doi.org/10.1007/3-540-36108-1_18

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 271–282. https://doi.org/10.1145/1926385.1926417

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Anil Madhavapeddy and Thomas Leonard. [n. d.]. *Eio*. https://github.com/ocaml-multicore/eio

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, James H. Anderson (Ed.). ACM, 204–213. https://doi.org/10.1145/224964.224987

K. C. Sivaramakrishnan. [n. d.]. *Domainslib*. https://github.com/ocaml-multicore/domainslib