Building a lock-free STM for OCaml

Vesa Karvonen
Tarides
vesa.a.j.k@gmail.com

Bartosz Modelski
Tarides
modelski.bartosz@gmail.com

Carine Morel
Tarides
carine@tarides.com

Thomas Leonard
Tarides
talex5@gmail.com

KC Sivaramakrishnan
Tarides
kc@kcsrk.info

YSS Narasimha Naidu
IIT Madras
shashankyeluri@gmail.com

Sudha Parimala
Tarides
sudharg247@gmail.com

Abstract

The `kcas`[1] library was originally developed to provide a primitive atomic lock-free multi-word compare-and-set operation. This talk introduces `kcas` and discusses the recent development of `kcas` turning it into a proper lock-free software transactional memory implementation for OCaml that provides composable transactions, scheduler friendly modular blocking, and comes with a companion library of composable lock-free data structures.

Motivation

Having just recently acquired the ability to have multiple domains running in parallel OCaml is in a unique position. Instead of having a long history of concurrent multicore programming we can start afresh. What sort of concurrent programming model should OCaml provide?

Libraries like Eio[2] and Domainslib[3] 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.

Traditional mechanisms for communication and synchronization, such as message queues, and mutexes and condition variables, do not compose. On the other hand, in the author's anecdotal experience, composable message passing models tend to be unfamiliar and challenging to programmers. Transactional memory is a more recent abstraction that offers both a relatively familiar programming model and composability.

Composing transactions

Consider the implementation of a least-recently-used (LRU) cache or a bounded associative map. A simple approach to implement a LRU cache is to use a hash table and a doubly-linked list and keep track of the amount of space in the cache:

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

On a cache lookup the doubly-linked list node corresponding to the accessed key is moved to the left end of the list:

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

On a cache update, in case of overflow, the association corresponding to the node on the right end of the list is dropped:

```
let set {table; order; space; _} key value =
  let node =
    match Hashtbl.find_opt table key with
    | None ->
      if 0 = Loc.update space (fun n -> max 0 (n-1))
      then Dllist.take_opt_r order
           |> Option.iter (Hashtbl.remove table);
      Dllist.add_l key order
    | Some (node, _) -> Dllist.move_l node order; node
  in
  Hashtbl.replace table key (node, value)
```

While this is a fine sequential implementation, in a concurrent setting this doesn't work even if the individual operations on lists and hash tables were atomic.

Fortunately, rather than having to wrap the cache implementation behind a mutex and make another individually atomic yet uncomposable data structure, or having to learn a completely different programming model and rewrite the cache implementation, we can use the transactional programming model provided by the `kcas` library and the transactional data structures provided by the `kcas_data`[4] library to trivially convert the previous implementation to a lock-free composable transactional data structure.

To make it so, we simply use transactional versions, `*.Xt.*`, of

[1](“`kcas`: Software Transactional Memory Based on Lock-Free Multi-Word Compare-and-Set” 2023)
[2](“`eio`: Effect-Based Direct-Style IO API for OCaml” 2023)
[3](“`domainslib`: Nested-Parallel Programming Library” 2023)
[4](“`kcas_data`: Compositional Lock-Free Data Structures and Primitives for Communication and Synchronization” 2023)

operations on the data structures and explicitly pass a transaction log, `~xt`, to the operations.

For the `get_opt` operation we end up with

```
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
```

and the `set` operation is just as easy to convert to a transactional version.

Transactional operations, like `get_opt`, can then be performed atomically by committing them

```
Xt.commit { tx = get_opt cache key }
```

or such operations can be composed further — just like we composed the `get_opt` operation itself by explicitly passing a transaction log, `~xt`, through the operations.

In addition to the ability to compose atomic operations, similarly to the Haskell STM,[5] `kcas` also provides the ability to await on arbitrary conditions over the contents of shared memory locations. As an example, we can convert the non-blocking `get_opt` operation to a blocking `get` operation as follows:

```
let get ~xt cache key =
  match get_opt ~xt cache key with
  | None -> Retry.later ()
  | Some value -> value
```

The `Retry.later ()` call tells the transaction mechanism of `kcas` that the transaction should only be retried after some shared memory locations accessed by the transaction have been modified by another thread of execution. While the operation is blocked, schedulers like Eio and Domainslib are free to run other threads on the domain.

The design and evolution of `kcas`

The `kcas` library was originally developed to provide a primitive atomic lock-free multi-word compare-and-set (k-CAS) based on a practical algorithm.[6] k-CAS is a powerful tool for designing concurrent algorithms as it allows one to update an arbitrary number of shared memory locations atomically. k-CAS is also attractive as, with relatively recently developed algorithms,[7] it can be implemented both efficiently, requiring only `k+1` single-word compare-and-set operations, and scalably, such that disjoint operations can progress in parallel without interference. These properties make k-CAS potentially competitive with locking based STM algorithms.[8]

However, there is still room to improve. Programming in terms of compare-and-set operations is not particularly convenient, as it requires one to laboriously build lists of operations, account for read skew, and implement mechanisms to retry in case of interference.

Furthermore, while k-CAS is able to express arbitrary updates of shared memory locations, it is quite common even for mutating operations on data structures to only read some locations.[9] Due to the way shared memory works, implementing read-only operations in terms of read-write operations is inefficient and unscalable.

With the goal to make `kcas` attractive both

- for experts implementing correct and performant lock-free data structures, and
- for everyone gluing together programs using such data structures

the `kcas` library has gone through a series improvements to both the interface and the internal implementation. After introducing the `kcas` and `kcas_data` libraries, we will briefly discuss these improvements.

In particular, this talk will briefly discuss:

- The new obstruction-free k-CAS-n-CMP algorithm[10] implemented by `kcas` that also provides efficient lock-free k-CAS as a subset.
- The direct style transactional programming interface of `kcas` that is easy to use with all the control flow structures of OCaml.
- The internal implementation of the transaction mechanism of `kcas` based on a splay tree and how the properties of splay trees can be exploited to allow allows many transactions to be performed in linear time.
- The scheduler friendly blocking mechanism[11] used by `kcas` and provided by schedulers like Eio and Domainslib allowing blocking abstractions to work across schedulers.

This talk will also briefly discuss the main trade-offs that `kcas` makes in an attempt to provide both scalable performance and a convenient programming model. In particular, this talk will briefly discuss:

- The support for nested transactions provided by `kcas` requiring explicit snapshot and rollback operations.
- The read skew anomaly or lack of opacity[12] and the ways in which `kcas` attempts to mitigate it.

References

Dice, Dave, Ori Shalev, and Nir Shavit. 2006. "Transactional Locking II." In *Proceedings of the 20th International Conference on Distributed Computing*, 194–208. DISC'06. Berlin, Heidelberg: Springer-Verlag. https://doi.org/10.1007/11864219_14.

[5](T. Harris et al. 2005)
[6](T. L. Harris, Fraser, and Pratt 2002)
[7](Guerraoui et al. 2020)
[8](Dice, Shalev, and Shavit 2006)
[9](Luchangco, Moir, and Shavit 2003)
[10](Karvonen 2023)
[11]("`domain-local-await`: A Scheduler Independent Blocking Mechanism" 2023)
[12](Guerraoui and Kapalka 2008)

“`domain-local-await`: A Scheduler Independent Blocking Mechanism.” 2023. https://opam.ocaml.org/packages/domain-local-await/.

“`domainslib`: Nested-Parallel Programming Library.” 2023. https://opam.ocaml.org/packages/domainslib/.

“`eio`: Effect-Based Direct-Style IO API for OCaml.” 2023. https://opam.ocaml.org/packages/eio/.

Guerraoui, Rachid, and Michal Kapalka. 2008. “On the Correctness of Transactional Memory.” In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 175–84. PPoPP ’08. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1345206.1345233.

Guerraoui, Rachid, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. “Efficient Multi-Word Compare and Swap.” https://arxiv.org/abs/2008.02527.

Harris, Tim, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. “Composable Memory Transactions.” In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 48–60. PPoPP ’05. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1065944.1065952.

Harris, Timothy L., Keir Fraser, and Ian A. Pratt. 2002. “A Practical Multi-Word Compare-and-Swap Operation.” In *Distributed Computing*, edited by Dahlia Malkhi, 265–79. Berlin, Heidelberg: Springer Berlin Heidelberg.

Karvonen, Vesa. 2023. “Extending k-CAS with Efficient Read-Only CMP Operations.” https://gist.github.com/polytypic/0efa0e2981d2a5fc4b534a0e25120cc9.

“`kcas`: Software Transactional Memory Based on Lock-Free Multi-Word Compare-and-Set.” 2023. https://opam.ocaml.org/packages/kcas/.

“`kcas_data`: Compositional Lock-Free Data Structures and Primitives for Communication and Synchronization.” 2023. https://opam.ocaml.org/packages/kcas_data/.

Luchangco, Victor, Mark Moir, and Nir Shavit. 2003. “Nonblocking k-Compare-Single-Swap.” In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 314–23. SPAA ’03. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/777412.777468.