# Software Transactional Memory for Multicore OCaml

https://vraajkum.github.io/ocaml-stm-webpage/

**Edward Brownhill** ebrownhi@andrew.cmu.edu
**Vishant Raajkumar** vraajkum@andrew.cmu.edu

## 1 Summary

We will implement and compare two software transactional memory (STM) systems for Multicore OCaml: a lazy versioning + optimistic conflict detection system, partially modeled after the TL2 algorithm, and an eager versioning + pessimistic conflict detection system modeled after LogTM. We will build transactional data structures on top of each, benchmark both against Domainslib mutex-based locks under varying contention, and perform a detailed analysis of throughput, abort rates, and synchronization overhead to understand which design wins and under what conditions.

## 2 Background

OCaml 5.0 removed the global interpreter lock, making true shared-memory parallelism possible for the first time. Each `Domain` maps to an OS thread and runs concurrently on a shared heap. The `Atomic` module provides sequentially consistent CAS, fetch-and-add, and exchange on heap values, which are the primitives any STM runtime needs for version clocks and commit logic. Crucially, ordinary OCaml reads and writes outside of `Atomic` are subject to relaxed reordering under OCaml 5's weak memory model, so getting the ordering right around transaction boundaries is a real concern, not a formality. Domainslib sits on top of domains and provides the mutex and condition-variable primitives we use as our lock-based baseline.

The only production STM for OCaml is `kcas`, maintained by the ocaml-multicore team. It implements the GKMZ multi-word CAS algorithm, which occupies the lazy versioning + optimistic conflict detection combination of the design space: writes are buffered and installed atomically at commit, conflicts are found only at commit time through version-clock validation. `kcas_data` builds a suite of composable data structures on top of it. This is a well-engineered library, but it only covers one point in the design space and leaves the eager versioning + pessimistic detection corner completely unimplemented in OCaml.

That gap is what this project addresses. Eager+pessimistic (LogTM-style) has meaningfully different performance characteristics: it writes directly to memory and checks for conflicts on every operation rather than deferring to commit time. Under high contention this pays off: conflicting transactions fail immediately rather than running to completion only to abort, but it trades that away for higher per-operation overhead and a more complex abort path that requires replaying an undo log. Whether that trade-off is worthwhile depends on contention level and transaction length in ways that have never been measured for OCaml specifically, because the runtime's GC and weak memory model introduce pressures that don't exist in the C or Java systems where LogTM was originally evaluated.

Because conflict detection granularity is a free parameter in any software STM, unlike hardware TM, where it is fixed at the cache line, we also study how tvar granularity affects throughput on array-heavy workloads, comparing object-level tracking against element-level tracking on a concurrent hash map benchmark.

## 3 The Challenge

Building a correct and performant STM in OCaml 5 is non-trivial for several reasons:

1. **OCaml 5's relaxed memory model**
   OCaml 5 is not sequentially consistent. The `Atomic` module provides SC operations, but ordinary field reads and writes may be reordered by the compiler and hardware. Flag increments, read-set validation, and write-set flushing must all be carefully sequenced with explicit atomic operations. A misplaced fence in the commit path may cause incorrect serialization that is extremely difficult to reproduce or diagnose under testing.

2. **Metadata overhead and read-set scalability**
   Every transaction must maintain a read set (for validation) and either a write buffer (LO-STM) or an undo log (EP-STM). These are dynamic data structures updated on every transactional access. At high thread counts and with long transactions, the cost of maintaining and scanning these sets can dominate. Designing efficient representations, maybe hash tables or sorted arrays or per-object version stamps, directly determines whether the STM is competitive with coarse-grained locking.

3. **Contention management and livelock**
   When two transactions conflict under EP-STM, both may repeatedly abort each other with no forward progress. We have flexibility in our implementation of a tie breaker (by priority, transaction age, or randomized backoff) to guarantee liveness. Getting this policy right without introducing starvation or excessive abort rates under high contention is a non-trivial systems problem.

4. **GC interaction**
   OCaml's garbage collector runs concurrently with domain execution in OCaml 5. Minor GC pauses can occur mid-transaction, causing delays that obscure the true performance characteristics of each design, or may actually interfere with logic. We will need to account for GC behavior when interpreting benchmark results, and separate GC time from transaction time in profiling.

5. **Correctness verification under concurrency**
   Sequential unit tests are not enough: a data structure can pass every single-threaded test and still produce wrong results under concurrent access due to rare interleavings that only show up under load. OCaml's `multicoretests` suite can stress-test for these by running operations in parallel and checking that some legal sequential ordering of those operations explains the observed result. Getting that to work with a custom STM means exposing enough internal state for the framework to reconstruct what happened.

We hope to develop practical intuition for how the two classic axes of STM design, both versioning strategy and conflict detection timing, interact with a real language runtime, and to produce a quantitative answer to: under what contention regimes and transaction lengths does each design outperform the other, and when does either outperform mutex locking?

# 4 Resources

We will develop and benchmark on GHC lab machines, which provide enough cores (8) to observe contention effects clearly. All code will be written in OCaml 5.2+ using the standard `Atomic` module and `Domain` API; OCaml 5.2 or later is required due to known bugs in the `Atomic` module present in earlier 5.x releases.

We start from scratch for both STM implementations. We will study the `kcas` source code as a reference for correctly interfacing with OCaml 5's memory model, but will not reuse its code.

**References:**
V. Viklund et al. `kcas`: Software Transactional Memory for OCaml. `https://github.com/ocaml-multicore/kcas`

# 5 Goals and Deliverables

We plan to achieve:

- A correct EP-STM implementation (eager versioning, pessimistic conflict detection) with a basic contention manager (exponential backoff on abort), verified against sequential reference executions and linearizability tests using `multicoretests`.
- Transactional versions of three benchmark data structures built on EP-STM: a shared counter, a singly-linked list, and a hash map.
- A benchmark suite measuring throughput (transactions/second), abort rate, and wall-clock time for each data structure under low, medium, and high contention, comparing EP-STM against `kcas` (the production lazy+optimistic reference) and Domainslib `Mutex`-based locks across 1–8 domains.
- A report answering: under what contention levels and transaction lengths does EP-STM outperform mutex locking, and where does the lazy+optimistic design of `kcas` have the edge?

Additionally, we hope to achieve:

- A correct LO-STM implementation (lazy versioning, optimistic conflict detection) for a direct within-codebase comparison against EP-STM, isolating the effect of versioning strategy from other implementation differences.
- A comparison of object-level vs. element-level conflict detection granularity within EP-STM, measuring false-sharing effects on the hash map benchmark.
- A contention manager study within EP-STM (aggressive abort vs. randomized backoff vs. priority-by-age), with abort-rate data across contention levels.

In case of slower progress:

- If EP-STM correctness proves too difficult within the timeline, we will focus on a thorough profiling study of EP-STM internals (per-operation conflict check cost, undo log overhead, GC interference) that still produces a strong 15-418 analysis.
- If the implementation is not complete enough to benchmark, we will contribute a detailed performance comparison of `kcas` against Domainslib `Mutex` across our benchmark suite, with analysis of where lock-free STM wins and loses.

## 6 Platform Choice

We will develop and benchmark on the GHC lab machines (multi-core x86). STM performance is driven by contention between threads sharing a cache hierarchy, and the GHC machines provide enough cores (8+) to observe the contention effects we care about, the crossover between low-contention and high-contention regimes, without needing a larger machine. More cores would not change the analysis; we expect that the interesting behavior emerges well before we saturate 8 threads. A GPU is entirely unsuitable: STM is a latency-sensitive, thread-collaborative workload that depends on fast inter-core communication and shared caches, which only become more needed if we switch to eager pessimistic implementations.

## 7 Schedule

| Date | Goals |
|---|---|
| Apr. 7 – Apr. 14 | EP-STM core: in-place writes, undo log, per-operation conflict check, and exponential backoff contention manager. Shared counter correct sequentially. |
| **Apr. 14** | **Milestone report due (11:59pm).** EP-STM skeleton sorta operational. Sequential correctness on shared counter established, or minor bugs away froms. |
| Apr. 14 – Apr. 21 | Complete EP-STM. Verify linearizability with `multicoretests` on counter and linked list. Implement transactional hash map. Baseline benchmark vs. Domainslib `Mutex` and `kcas`. |
| Apr. 21 – Apr. 28 | Run full benchmark matrix (3 data structures $\times$ 3 contention levels $\times$ 1–8 domains). Analyze crossover points. Stretch: granularity comparison and contention manager study. |
| Apr. 28 – Apr. 30 | Write final report. Prepare poster. |
| **Apr. 30** | **Final report due (11:59pm).** |
| **May 1** | **Poster session.** 8:30–11:30am. |

Table 1: Project schedule.