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

Summary

We implemented two software transactional memory (STM) systems for Multicore OCaml from scratch and evaluated them against the production `kcas` library and a standard-library Mutex baseline on the GHC lab machines. We benchmarked the systems across 1 to 8 OCaml domains on four transactional data structures: a shared counter, a hash map, a linked list, and a bank account. Both STM designs trailed `kcas` and `Mutex` by a significant margin on high-contention workloads, and EP-STM exhibited substantially higher abort rates than LO-STM at moderate domain counts, but we observed relative throughput depends heavily on the read/write ratio.

Background

Software transactional memory gives programmers a declarative atomicity primitive. Rather than manually acquiring and releasing locks, a programmer wraps a block of code in an atomic clause, which guarantees the block executes as if it were a single indivisible atomic operation. Two concurrent transactions can conflict if one reads from a location the other writes to or if both write to the same location. In this case, at least one transaction must abort and retry from scratch. Transactions that do not conflict proceed entirely in parallel.

STM Designs:

We decided to implement two different STM designs: *LO-STM* (Lazy-Optimistic STM) and *EP-STM* (Eager-Pessimistic STM), which differ in their versioning strategy and conflict detection timing.

LO-STM uses lazy versioning, meaning uncommitted writes are stored privately and shared memory is only updated when the transaction is committed. This is paired with optimistic conflict detection, meaning a transaction assumes no conflict will occur and defers validation to the commit phase, where it checks if any location it read from has been modified concurrently.

EP-STM uses eager versioning, meaning writes immediately update shared memory. The old values are preserved in an undo log to support rollback. This is paired with pessimistic conflict detection, meaning every read and write checks for conflicts with concurrent transactions as they execute. The transaction then aborts immediately upon detecting one.

The two designs are optimized for different workload contention rates. LO-STM is optimized for low-contention environments where most transactions succeed and early conflict checking would introduce unnecessary overhead. In contrast, EP-STM is optimized for high-contention environments, catching conflicts early and avoiding the wasted computation of running a transaction to completion only to discover at commit time that it must be retried.

Data Structures:

Transactional Variables

Both STM designs are built on *transactional variables* (`tvar`), the unit of shared mutable state. Each LO-STM `tvar` is a record of four fields: a unique integer `id`, a mutable `value`, an atomic integer `version` stamp recording the clock time of the last committed write, and a per-`tvar` `Mutex` held only during the commit phase. Each EP-STM `tvar` contains the same four fields, with an additional boolean atomic `write_locked` flag that is set when a transaction first writes to the `tvar`, allowing concurrent readers to detect the conflict without acquiring the mutex.

```
(* LO-STM tvar *)
type 'a tvar = {
  id            : int;
  mutable value : 'a;
  version       : int Atomic.t;
  lock          : Mutex.t;
}
```

```
(* EP-STM tvar *)
type 'a tvar = {
  id            : int;
  mutable value : 'a;
  version       : int Atomic.t;
  lock          : Mutex.t;
  write_locked  : bool Atomic.t;
}
```

Figure 1: `tvar` layouts for LO-STM (left) and EP-STM (right)

Global and Per-Transaction Data Structures

Both designs share a single global version clock, a sequentially consistent atomic integer that every committing writing transaction increments via `Atomic.fetch_and_add` to obtain a unique write timestamp and that every transaction reads at `begin_txn` to establish a consistent read window.

Each transaction owns a *descriptor*, an ephemeral record discarded on commit or abort. The LO-STM descriptor holds a `read_set`, a hash table mapping each `tvar`'s `id` to the version stamp observed on first read. It also holds a `write_buf`, a separate hash table mapping each `tvar`'s `id` to the new value which is kept private until commit. The EP-STM descriptor replaces the write buffer with a `write_set`, a hash table recording every `tvar` that the transaction owns, and an `undo_log`, which is a list of closures with each capturing the old value of one `tvar` (applied in reverse order on abort).

Benchmarking

We also built three transactional data structures used in benchmarking. The *transactional hash map* (`Thashmap`) stores n separate bucket of `tvar`s, each holding an association list. Operations on different buckets never conflict, providing good parallelism under low-to-medium contention. The *transactional list* (`Tlist`) stores the entire sorted spine in a single `tvar`. Every insert or remove is one read plus one write, creating high contention. The *shared counter* benchmark is simply one `tvar` that every domain increments in a loop, which is the worst-case conflict scenario. We also implemented a `Bank` module on top of LO-STM, to test and demonstrate multi-tvar atomicity: a `transfer` operation debits one account tvar and credits another in a single `atomically` call, with no intermediate state visible to other domains.

Operations:

The core operations are `txn_read`, `txn_write`, and `commit`, wrapped by `atomically` which handles the retry loop with exponential backoff.

LO-STM

- `txn_read` - If the `tvar` is already in the write buffer, return the buffered value. Otherwise, read `version` before and after reading `value`. If the two versions differ, a concurrent commit was in progress and we retry. If the stable version exceeds `start_ts`, the data comes after our transaction and we abort immediately. Otherwise, record the observed version in the read set.
- `commit` - Read-only transactions validate in a single pass with no locking. Read-write transactions proceed in five phases:
  1. Sort the write-set keys by id to establish a global lock order and acquire per-`tvar` mutexes.
  2. Re-validate every read-set entry. If any version has changed, release all locks and abort.
  3. Call `Clock.increment` to obtain a write timestamp.
  4. Install new values and stamp versions.
  5. Release all locks.

EP-STM

- `txn_write` - If the `tvar` is already owned, overwrite in place (the undo log already holds the original value). Otherwise, `try_lock` the per-`tvar` mutex. Failure means a write-write conflict and we abort immediately. Set `write_locked` to announce ownership, push a restore onto the undo log, and perform the eager in-place write.
- `txn_read` - If the `tvar` is in the write set, return the current in-place value. Otherwise, poll `write_locked`. If true, another transaction owns the `tvar` and we abort. Take a snapshot as in LO-STM, then re-check `write_locked` to close the race between the initial poll and the value read.
- `abort` - Replay the undo log in reverse order to restore old values, then clear `write_locked` and release all per-`tvar` mutexes.

Parallelism, Dependencies, and Locality

Workloads are not data-parallel in the way that we usually deal with in this class. Each domain does not process a regular, independent partition of data in isolation. Instead, parallelism occurs whenever the given workload allows for it to happen. Transactions involving disjoint `tvar`s execute concurrently with no synchronization, whereas transactions involving shared `tvar`s are serialized at commit time. Thus, the degree of parallelism depends on the conflict rate. This is controlled in our benchmarks by the number of keys (more keys decreases collisions) and the write fraction (more writes increases conflicts).

The critical dependency in both designs is the global version clock. Every committing writer performs a `Atomic.fetch_and_add` on a single shared cache line, which can become a sequential bottleneck at high domain counts regardless of conflict rates.

The designs are difficult to parallelize with SIMD execution. STM involves a lot of control flow and pointer-chasing. Hash table lookups, linked-list traversals, and conditional aborts all involve data-dependent branches and irregular memory access patterns that aren't friendly to vectorization. GPU execution would be equally unsuitable. STM depends on fine-grained shared-cache communication between threads, which is not available across CUDA warps.

Approach

Technologies and Platform

Both STM systems are implemented from scratch in OCaml 5.2.1, built with `dune`, and targeted at the GHC lab machines. We relied on three standard library primitives throughout:

1. `Atomic` - sequentially consistent `get`, `fetch_and_add`, `compare_and_swap` for the version clock and per-`tvar` ownership flags
2. `Mutex` - try-lock and blocking lock for write-set locking during commit
3. `Domain` - `spawn`, `join`, `cpu_relax` for parallelism and spin-wait hints

We studied the `kcas` library source as a reference for correctly sequencing atomics under OCaml 5's relaxed memory model, but no code was reused. The TL2 paper and the LogTM paper served as the primary algorithmic references.

Inputs, Outputs, and Workload

The input to each benchmark run is a set of D OCaml domains, each executing N transactional operations against a shared data structure. The output is an execution history and the final state of the data structure is identical to some valid sequential interleaving of all committed transactions. Correctness is verified by assertion after each benchmark. For the counter, we assert `value = D * N` For the hash map and list, correctness is verified separately using the `multicoretests` linearizability harness.

Mapping to Parallel Hardware

Each OCaml `Domain` maps directly to one OS thread, which the OS scheduler assigns to a physical core. There is no warp, thread block, or gang structure. All parallelism is multiple instructions, multiple data (MIMD), with each domain running an independent instruction stream and communicating only through shared memory.

Shared State

The shared state that all domains access concurrently consists of three layers. At the top is the global version clock, a single `int Atomic.t` wrapped in its own record so the compiler places it on its own cache line. Every committing writer calls `Clock.increment` (`fetch_and_add`) on this location.

Below that are the per-`tvar` metadata fields, `version` and `write_locked`, each an `Atomic.t` updated at commit or abort. At the bottom is the `tvar` `value` field, a plain OCaml mutable record field. In LO-STM, it is written only under the `tvar`'s mutex while in EP-STM, it is written eagerly by the first transaction to acquire the `tvar`. Per-transaction state is strictly domain-local. It is never shared with other domains.

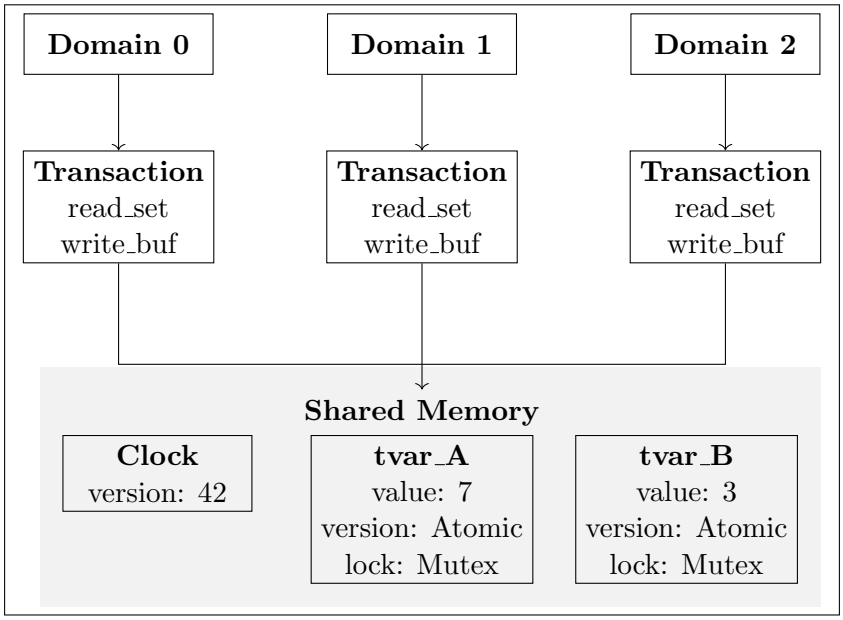

Figure 2: LO-STM architecture overview.

LO-STM: Design Decisions and Iterations

Seqlock-style reads. The first version of `txn_read` read `tv.value` and `tv.version` in two separate `Atomic.get` calls with no bracketing. Under OCaml 5's relaxed memory model, a concurrent committer could install a new value between the two reads, leaving us with a stale version stamp that matched a value we never actually observed. We fixed this by adopting a seqlock snapshot. We read `version` before the value, read the value, read `version` again, and retry if the two version reads differ. This is the same technique used in `seqlock` and is safe because `version` is an `Atomic.t` (SC load) while `value` is bracketed by the two SC reads.

Global lock ordering at commit. An early version of `commit` acquired write-set locks in hash table iteration order, which is non-deterministic. Two transactions with overlapping write sets could each hold one lock the other needed, deadlocking permanently. We fixed this by sorting write-set keys by tvar id before acquiring locks (`sort_keys` in `stm.ml`). Since `tvar` ids are assigned sequentially at allocation via a global `Atomic.fetch_and_add`, sorting by id gives a stable total order, eliminating the deadlock.

Read-only fast path. Profiling early benchmarks showed that a significant fraction of `atomically` calls on the hash map were read-only. The full commit path acquired locks unnecessarily in these cases. We added an explicit read-only fast path in `commit`. If the write buffer is empty, skip all lock acquisition and validate the read set directly, which is safe because no concurrent writer can commit to a `tvar` we are validating without first acquiring that `tvar`'s lock.

Exponential backoff. The initial retry loop restarted immediately on abort, which under high

contention caused an problem where all aborting domains re-entered the transaction simultaneously, re-conflicted, and aborted again. We added exponential backoff using `Domain.cpu_relax` (a hardware pause hint). After each abort, the domain spins for `backoff` iterations, then doubles `backoff` up to a cap of 1024 before retrying.

EP-STM: Design Decisions and Iterations

The `write_locked` flag. The central design challenge in EP-STM is detecting read-write conflicts without requiring readers to acquire a lock. We added a `write_locked : bool Atomic.t` field to the EP-STM `tvar`.

GC-stable hashtable keys. Both descriptors use `tv.id` as the hash table key rather than the `tvar` pointer itself. OCaml's minor GC can move heap-allocated objects between the initial `txn_read` and a later lookup in the same transaction, invalidating a pointer-keyed table entry. Integer ids are immediate (unboxed, untraced) values that the GC never moves, so a `tvar` looked up twice in one transaction always finds the same bucket. We store an `Obj.t` reference to the tvar as the hash table value so we can recover the full tvar pointer after a potential GC move.

Contention manager. The EP-STM contention manager is a single `raise Abort`, combined with the same exponential backoff in the `atomically` retry loop as LO-STM.

Data Structure Implementation

Both `Thashmap` and `Ep_thashmap` use the same design: an array of n bucket tvars, each holding an association list. The two implementations are structurally identical, except the tvar and transaction types, confirming that the STM interface is a clean abstraction boundary. The same is true of `Tlist` and `Ep_tlist`. One multi-tvar operation worth noting is `Thashmap.move`, which atomically transfers a value from one bucket to another by reading and writing both bucket tvars in a single `atomically` call. This operation would require ordered lock acquisition or a global lock under a mutex-based design; with STM it is a two-line extension of the single-key `set` operation.

Results

Experimental Setup

All benchmarks were run on GHC lab machines using OCaml 5.2.1 with the `dune` build system. Each domain executes a fixed number of transactional iterations against a shared data structure. Wall-clock time is measured with `Unix.gettimeofday`. The primary performance metric is *throughput*: committed transactions per second, computed as (domains $\times$ N)/elapsed. We also record total commits and total aborts per implementation via per-domain atomic counters, from which we derive the *abort rate*: aborts/(commits + aborts).

We compare four implementations throughout:

- **LO-STM**: our TL2-based lazy/optimistic implementation
- **EP-STM**: our LogTM-based eager/pessimistic implementation
- **kcas**: the production lock-free STM library for OCaml, based on a multi-word CAS algorithm
- **Mutex**: a handcrafted standard-library `Mutex` baseline, providing the same conflict granularity as the STM implementations

Domain counts tested were 1, 2, 4, and 8. Contention level is controlled by key range. High contention uses a 16-key range (all domains hash to few buckets), medium uses 64 keys, and low uses 256 keys. Each benchmark was run with $N = 100{,}000$ iterations per domain for the counter and $N = 50{,}000$ for the hash map and list benchmarks.

Shared Counter

The shared counter is the highest-contention benchmark. Every domain reads and writes the same single `tvar` on every transaction, so every pair of concurrent transactions conflicts.

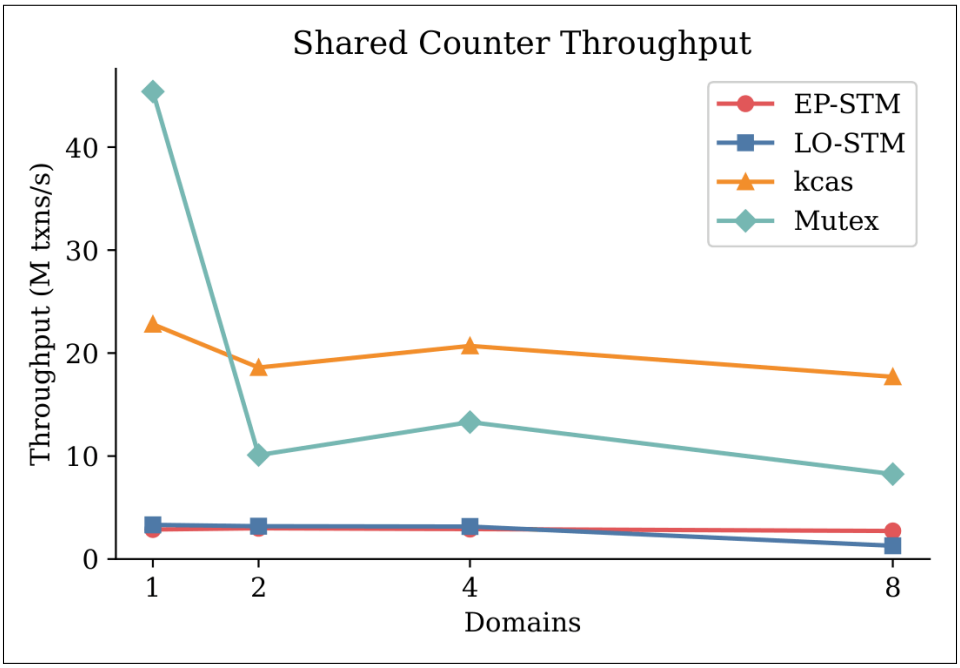

Figure 3: Shared counter throughput (M txns/s) vs. domain count for EP-STM, LO-STM, kcas, and Mutex.

As shown in Figure 3, EP-STM achieved roughly 2.7-3.0 M txns/s across all domain counts while LO-STM held near 3.2 M txns/s at 1-4 domains before dropping to 1.29 M at 8. `kcas` achieved ∼18-23 M txns/s and single-domain `Mutex` posted ∼46 M txns/s. `Mutex` throughput collapsed sharply with parallelism, dropping from 46 M at 1 domain to ∼8 M at 8 domains, because a single global lock serializes all domains entirely. `kcas` degraded less sharply, settling near 18 M txns/s at 8 domains, reflecting its lock-free MCAS design which avoids the hard serialization of a mutex but still contends on a single location.

Neither LO-STM nor EP-STM showed meaningful scaling on this workload. This is expected since the global version clock requires every committing transaction to perform a `Atomic.fetch_and_add` on a single shared cache line, which is a sequential bottleneck independent of how the conflict is detected. Even at 1 domain, where there is no actual contention, both STMs are an order of magnitude slower than `Mutex` and `kcas`, revealing that the per-transaction overhead of maintaining read sets and write buffers (LO-STM) or undo logs (EP-STM) is the dominant cost at low parallelism, not contention itself.

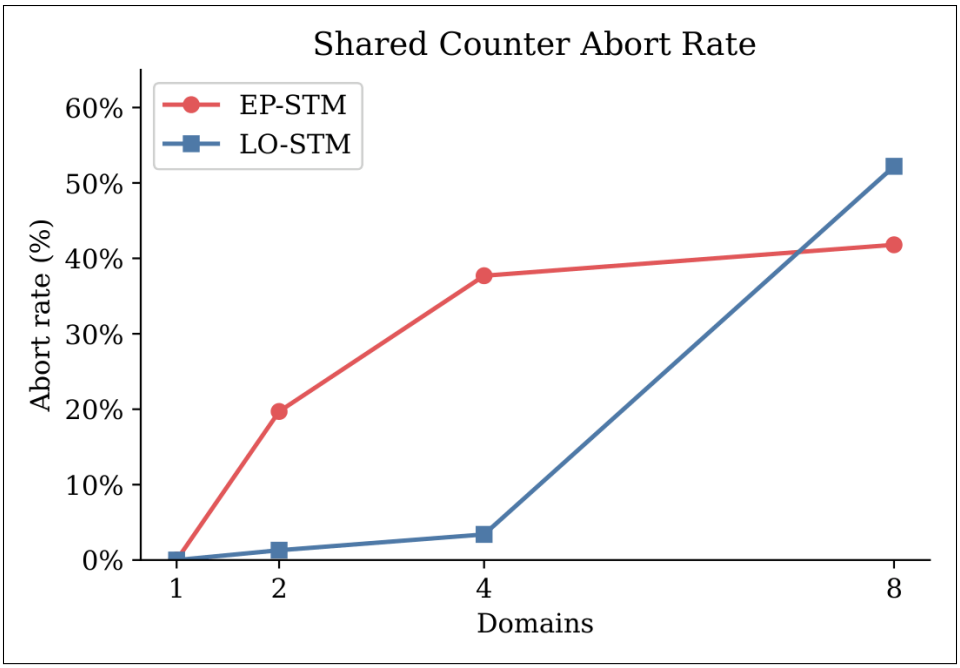

Figure 4: Shared counter abort rate (%) vs. domain count for EP-STM and LO-STM.

Figure 4 shows that EP-STM's abort rate was substantially higher than LO-STM's at nearly all domain counts. The gap was roughly 20% vs. 2% at 2 domains and 38% vs. 3% at 4 domains, The abort rates did converge near 42% and 52% respectively at 8 domains, with the EP-STM abort rate appearing to taper off but the LO-STM abort rate continuing to rise. This reflects the fundamental difference between the two designs: EP-STM detects conflicts eagerly on every read and write and aborts immediately, so at 2-4 domains a significant fraction of transactions are aborted mid-execution before they have done much work. LO-STM defers detection to commit time, so transactions run to completion before discovering a conflict. At low domain counts, most transactions succeed because conflicts are relatively rare and short-lived, keeping the abort rate low.

Despite a lower abort rate, LO-STM delivered only 1.29 M txns/s at 8 domains versus EP-STM's 2.73 M, a 2.1× disadvantage, because EP-STM aborts fail fast at `try_lock` before any in-place writes occur, whereas each LO-STM abort discards a fully-executed transaction body including write buffer maintenance and lock acquisition. The extra aborts consume CPU cycles but do not compound because EP-STM's contention manager immediately aborts on conflict with no backoff delay before the first retry. Aborted work is discarded quickly, and the domain re-enters the transaction with minimal latency.

Hash Map

Figure 5 shows hash map throughput at 8 domains under high contention across three workload types: write-only, mixed (80% reads / 20% writes), and read-only. Several patterns are worth noting.

Under the *write-only* workload, all four implementations performed relatively similarly, with `kcas` slightly ahead. Write-only hash map transactions are short (one bucket read, one bucket write) and always conflict at high contention, so the workload is highly serialized.

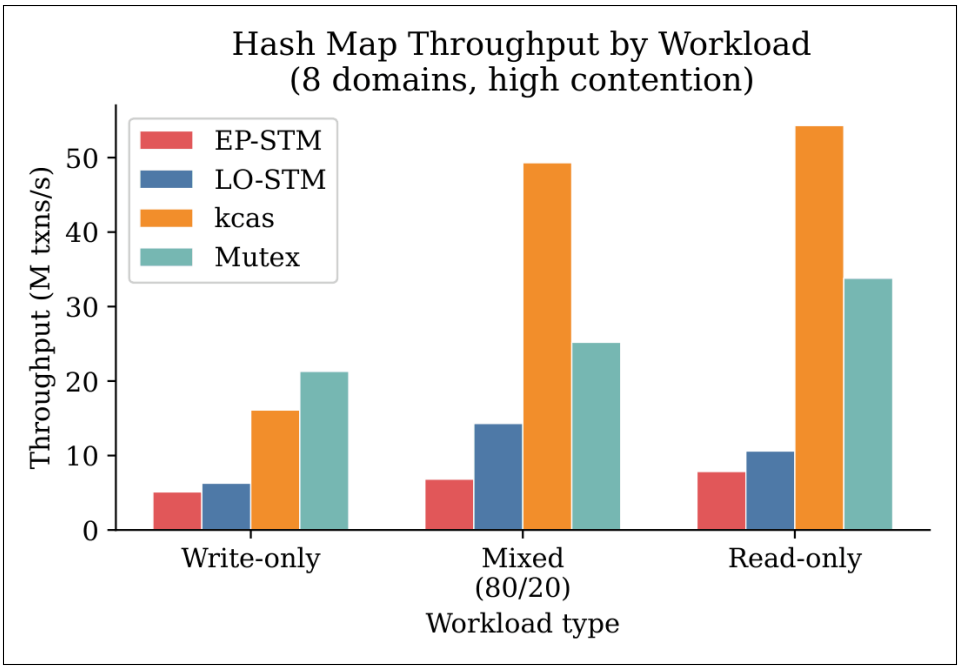

Figure 5: Hash map throughput (M txns/s) under write-only, mixed (80% read / 20% write), and read-only workloads at 8 domains under high contention (16-key range).

The *mixed* workload (80/20 read/write) is where the STM designs are intended to shine: reads that do not conflict should proceed in parallel, reducing effective contention. LO-STM and EP-STM both improved over write-only, and both closed the gap on `Mutex` on this workload. This clearly demonstrates the value of fine-grained conflict detection. The `Mutex` implementation acquires a per-bucket lock on every operation including reads, whereas both STMs allow non-conflicting reads to proceed without blocking writers in other buckets. `kcas` again led all implementations, benefiting from its lock-free single-CAS read path.

Under the *read-only* workload, `kcas` outperformed all other systems by a wide margin. LO-STM read-only transactions use the fast path in `commit` (no locks, a single validation pass), but still allocate a transaction descriptor and populate a read set on every call to `atomically`. `kcas` exposes a wait-free single-location read that bypasses the transaction machinery entirely for non-modifying accesses. In the benchmark, read-only operations call this directly rather than going through `Kcas.Xt.commit`, giving kcas a significant structural advantage on this workload. EP-STM performed comparably to LO-STM on read-only workloads because its read path has similar cost.

What Limited Performance

The results point to four distinct bottlenecks, in rough order of impact:

1. **Transaction descriptor overhead** - Both STMs allocate a fresh `Hashtbl` on every call to `atomically`, regardless of transaction size. For the counter benchmark, where each transaction touches exactly one `tvar`, this allocation dominates execution time. `kcas` avoids this entirely for single-location operations. A fixed-size stack-allocated descriptor (as in hardware TM or optimized STM libraries) would substantially reduce this cost.

2. **Global clock serialization** - Every committing write transaction increments a single shared atomic counter. On an 8-core machine with a 64-byte cache line, this becomes a bottleneck since the cache line holding the clock must migrate between cores on every commit. This is likely the primary reason neither STM scales on the counter benchmark and why throughput is flat from 1 to 8 domains. A distributed clock would reduce this pressure but requires a more complex validation protocol.
3. **Abort and retry cost under high contention** - EP-STM's immediate-abort contention manager avoids wasted work per abort but creates high retry frequency. LO-STM's optimistic approach avoids aborts at low contention but wastes more work per abort (an entire transaction completes before the conflict is discovered). Both strategies converge to poor throughput under maximum contention. A priority-based or backoff-tuned contention manager could reduce the retry frequency.
4. **Read-set and write-set scan at commit.** LO-STM's commit must iterate the entire read set to validate it. For the hash map under low contention, transactions may read multiple buckets before writing one, making the read set larger than the write set. This scan is $O(|\text{read set}|)$ and holds write-set locks while doing so, increasing the window during which other transactions are blocked.

Machine Target

OCaml 5.2 on multi-core x86-64 was the appropriate platform for this project. OCaml 5's `Domain` abstraction maps directly to OS threads with true shared-memory parallelism (the global interpreter lock was removed in OCaml 5.0), and the `Atomic` module provides sequentially consistent `fetch_and_add` and `compare_and_swap`, which happen to be the exact primitives needed for version clocks and per-`tvar` ownership flags. OCaml's garbage collector eliminated many memory safety bugs, allowing us to focus on concurrency correctness rather than manual memory management. A GPU would have been unsuitable for the reasons described above.

References

- D. Dice, O. Shalev, N. Shavit. *Transactional Locking II.* DISC 2006.
- K. Moore, J. Bobba, M. Moravan, M. Hill, D. Wood. *LogTM: Log-based Transactional Memory.* HPCA 2006.
- V. Karvonen et al. `kcas`: Software Transactional Memory for OCaml. `https://github.com/ocaml-multicore/kcas`
- `multicoretests`: Multicore Testing Tools for OCaml. `https://github.com/ocaml-multicore/multicoretests`

Distribution of Work

The distribution of work was balanced, and we both were always in communication about it, and are happy with the result. While the split didn't exactly follow the one outlined in the midterm report, we did try to both get involved in each step. Vishant was the lead on the LO-STM section, while Eddie was the lead on the EP-STM section.