CS6868 Research Mini-Project

Multi-Word Compare-and-Swap (MCAS) in OCaml 5

Sanjeev Reddy
CS25S013@smail.iitm.ac.in

Albin James
CS25S027@smail.iitm.ac.in

Kunal Umaji
CS25S020@smail.iitm.ac.in

May 1, 2026

Abstract

We implemented a software multi-word compare-and-swap (MCAS) primitive in OCaml 5 and used it to build concurrent data structures that require multi-location atomicity. The core artifact is a descriptor-based volatile MCAS following the efficient scheme of Guerraoui et al. [2], augmented with helping and deterministic address ordering. On top of this primitive, we built an MCAS-based atomic snapshot object, a lock-free sorted singly linked-list set, and an MCAS-based doubly linked-list set. We verified the core MCAS, the snapshot, and both linked-list variants using QCheck-Lin, and we also ran QCheck-STM and ThreadSanitizer-based stress tests for the STM-style support modules used during development. Our evaluation shows a clear trade-off. For snapshots, the software MCAS layer is substantially more expensive than double-collect, especially for scan-heavy workloads: at 32 domains, pure scans cost about $6.20\mu s$ per operation with MCAS versus $0.067\mu s$ for the baseline. However, MCAS becomes more attractive when a single operation must update several linked locations consistently. In our doubly linked-list benchmark, the MCAS version outperformed a coarse-grained mutex baseline on balanced and update-heavy workloads at moderate to high concurrency. Overall, MCAS is a useful abstraction for composable lock-free updates, but its performance cost must be justified by the need for true multi-location atomicity.

1 Goals

The project goal was to understand whether a software implementation of multi-word compare-and-swap can serve as a practical building block in OCaml 5, where the hardware-facing `Atomic` interface only offers single-word CAS. MCAS is interesting because many concurrent objects naturally need to update multiple words at once: linked-structure rewiring, record migration, and snapshots are standard examples in the non-blocking data-structure literature [4, 2]. The topic connects directly to the course themes of linearizability [5], non-blocking progress [3], and the gap between expressive software synchronization and limited hardware primitives.

We set out to build more than an isolated primitive. The repository contains three layers of artifacts:

1. a reusable volatile MCAS implementation in `volatile/mcas_volatile.ml`,
2. data structures implemented on top of it (`snapshot/mcas_snapshot_volatile.ml`, `linked_list/mcas_lockfree_linked_list.ml`, and `doubly_linked_list/mcas_doubly_linked_list.ml`),
3. correctness and performance infrastructure in `qlin/`, `stm/`, `tsan/`, and `benchmarking/`.

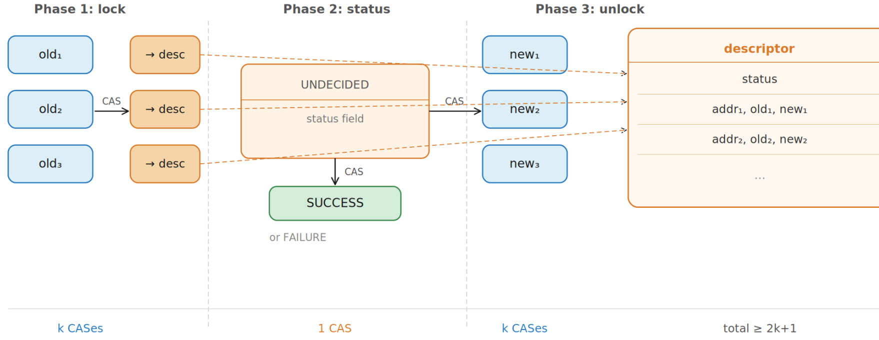

Figure 1: Descriptor lifecycle used by the MCAS implementation: acquire target words, decide the outcome, then interpret installed descriptors through the final status. Adapted from our project assets and based on the algorithmic structure in [2].

Research question. Can a software MCAS in OCaml 5 provide enough expressive power to simplify the construction of atomic snapshots and linked concurrent sets, and when does that extra atomicity justify its performance overhead relative to specialized single-word-CAS or lock-based baselines?

2 Background

An MCAS operation receives k triples $\langle addr_i, old_i, new_i \rangle$ and must atomically perform the following: if every target location still contains its expected old value, replace all old_i with new_i; otherwise, leave memory unchanged and return failure [2]. The challenge is that hardware does not provide such an operation directly.

The classic software idea is to replace plain values temporarily with *descriptors*. A descriptor records the participating words and the operation's status. Threads that encounter a descriptor do not block; instead, they *help* finish the pending operation, which is the standard route to lock-freedom [3, 1]. A successful read must therefore distinguish between a stable value and an installed word descriptor, and, in the latter case, interpret the logical value from the parent operation's status.

The key invariants behind our implementation are:

1. target references are sorted by a unique identifier before installation, preventing cyclic interference between overlapping operations;
2. every installed word descriptor points to one parent MCAS descriptor whose status is in {`Active`, `Successful`, `Failed`};
3. a read that sees an active descriptor helps complete it before returning, ensuring system-wide progress;
4. the logical contents of a claimed word are derived from the descriptor status: `new_value` after success and `old_value` after failure.

This design is especially attractive for operations that need to maintain cross-pointer consistency. A doubly linked-list insertion, for example, must update a predecessor's `next` pointer and a successor's `prev` pointer together; doing this with only single-word CAS usually requires extra marking states or more complicated retry logic.

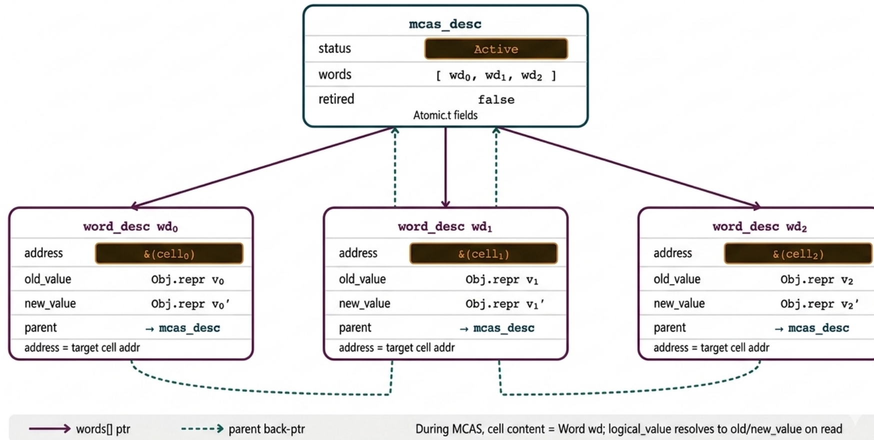

Figure 2: Descriptor object graph for one MCAS operation. A parent descriptor stores the global status, while each claimed word keeps a backpointer to the same parent. This organization explains how helping and logical reads are coordinated in the implementation.

3 Tasks Undertaken

3.1 Implementation

Core volatile MCAS. The working implementation resides in `volatile/mcas_volatile.ml`. Each MCAS-capable location is a cell whose state is either `Value of Obj.t` or `Word of word_desc`. A `word_desc` stores the address, old value, new value, and a backpointer to its parent descriptor. The parent `mcas_desc` stores an atomic status flag and the array of word descriptors.

The main routines are compact but subtle:

- `read_internal` dereferences a cell, helps any active foreign descriptor, and returns the logical value visible at that address.
- `acquire_word` installs a `Word` descriptor into one location if the current logical value still matches the expected old value.
- `mcas_desc` iterates through the sorted word list, attempts to acquire all locations, then flips the descriptor status atomically to either `Successful` or `Failed`.

The code uses physical equality on boxed values (`Obj.t`) and a unique integer `id` per reference for sorting and duplicate detection. The design is intentionally minimalist: there is no separate persistent-memory layer, epoch management, or helping queue. Figure 2 is a useful way to read the code: the parent descriptor owns the decision, while the per-word descriptors only record how that decision applies at individual cells.

MCAS-based snapshot. In `snapshot/mcas_snapshot_volatile.ml`, each register is an MCAS reference. An update is a one-word MCAS that retries until it succeeds. A scan first reads all slots and then validates the whole snapshot by issuing an MCAS whose expected and desired values are the same for every slot. If any slot has changed in the meantime, validation fails and the scan retries. Compared with the double-collect baseline in `snapshot/snapshot.ml`, this makes the linearization argument conceptually uniform but adds a full k-word MCAS to every scan.

Lock-free singly linked list. The repository contains both a standard CAS-based Harris-style ordered set in `linked_list/lockfree_linked_list.ml` and an MCAS variant in `linked_list/mcas_lockfree_linked_list.ml`. The MCAS version stores for each node a `next` pointer and a

separate `deleted` flag, both as MCAS references. Insertions and deletions use one multi-word update to couple pointer changes with consistency checks on neighboring nodes. This avoids the usual mark-bit encoding, but it pays the MCAS overhead even for operations that the single-word-CAS baseline handles with one or two CAS instructions.

MCAS doubly linked list. The most compelling application is `doubly_linked_list/mcas_doubly_linked_list.ml`. Here MCAS lets us update `prev`, `next`, and deletion metadata in one atomic step. This is compared against a coarse-grained mutex baseline in `doubly_linked_list/coarse_doubly_linked_list.ml`. Unlike the singly linked-list case, the baseline is not lock-free; MCAS therefore tests whether better composability can compensate for its software overhead.

3.2 Testing and verification

We used several complementary validation techniques.

QCheck-Lin linearizability tests. The directory `qlin/` contains linearizability tests for the volatile MCAS itself, the MCAS snapshot, the single-word-CAS linked list, and the MCAS linked list. The command generators produce mixed concurrent histories such as `Get/Set/Cas` for MCAS and `Insert/Delete/Member` for sets. The tests are small but targeted: for example, the linked-list tests restrict keys to a tiny range $0 \ldots 15$ to maximize contention and expose races quickly.

QCheck-STM model-based tests. The `stm/` directory contains state-machine tests for the development support modules `Stm_volatile` and `Stm_snapshot`. These compare the implementations against simple sequential models and run both sequential and parallel test suites. While these modules are not the main artifact of the report, they gave us an extra sanity check on the auxiliary API design and the test harness structure.

ThreadSanitizer stress tests. The `tsan/` programs repeatedly exercise shared references and snapshots from several domains. TSan cannot prove linearizability, but it is effective at exposing unsynchronized memory accesses that property-based tests might miss.

In this code base, the main bugs such testing is designed to catch are incorrect helping behavior, forgetting to validate neighbors before rewiring linked-list nodes, and retry loops that can return values without first stabilizing an in-progress descriptor. We did not find evidence in the recorded artifacts that the final checked-in versions violate these properties, but the tests are still bounded and therefore not a proof.

4 Evaluation

Experimental setup. Benchmarks were run from the repository's `benchmarking/` programs on the project machine available in the current environment: an Intel Core i5-10300H with 8 logical CPUs, running OCaml 5.4.0. The snapshot benchmark uses 16 registers and 200,000 operations per domain for domain counts $2, 4, 8, 16, 32$. The linked-list benchmarks prefill 128 keys from a key space of 256 and run 150,000 operations per domain for the same domain counts. The recorded CSV files in `presentation/data/` appear to contain one run per configuration, reporting elapsed time and average microseconds per operation. This means the numbers are best interpreted as directional rather than as a full statistical study. In particular, 16 and 32 domains oversubscribe the available hardware threads, so tail-end slowdowns are expected.

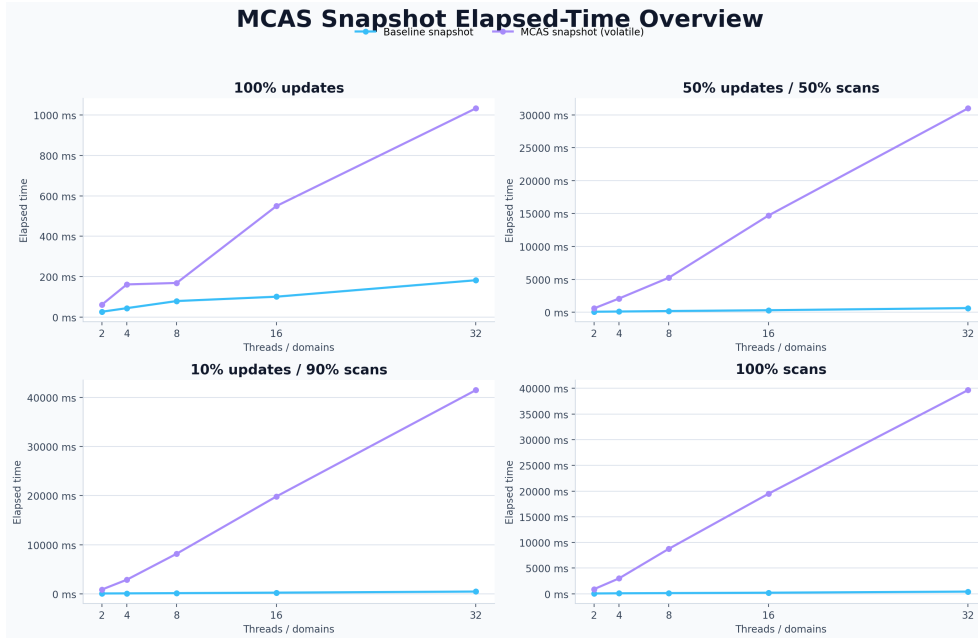

Figure 3: Snapshot benchmark summary from the project artifacts. The MCAS snapshot remains correct and compositional, but the extra validation work makes it significantly slower than double-collect except in the conceptual sense of providing a stronger primitive.

Snapshot results. Figure 3 shows that MCAS is much more expensive than double-collect for snapshots, especially when scans dominate. At 32 domains, the baseline needs only $0.029\mu s$ per update in the 100% update workload, whereas MCAS needs $0.161\mu s$, about $5.6\times$ slower. The gap widens dramatically for scan-heavy workloads because each scan in the MCAS version performs a full-array validation via MCAS: $6.197\mu s$ versus $0.067\mu s$ for 100% scans ($\approx 92\times$ slower). The balanced workload also shows a large penalty ($4.845\mu s$ versus $0.097\mu s$ at 32 domains).

Singly linked-list results. The singly linked-list experiment compares a specialized single-word-CAS design against an MCAS-based one. Here the specialized baseline wins consistently. In the balanced workload at 8 domains, the CAS list costs $0.093\mu s$ per operation, versus $0.470\mu s$ for the MCAS list ($\approx 5\times$ slower). At 32 domains, the gap grows to roughly $15\times$ ($0.088\mu s$ versus $1.299\mu s$). This result is not surprising: the CAS baseline is already a lock-free design tailored to the problem, so the extra generality of MCAS mostly appears as overhead.

Doubly linked-list results. The doubly linked-list benchmark is the most informative comparison because the baseline is a coarse mutex implementation. Here MCAS becomes competitive and, in several cases, clearly better. At 8 domains, the MCAS version outperforms the mutex baseline in all three workloads: for example, $0.494\mu s$ versus $1.061\mu s$ in the balanced workload and $0.574\mu s$ versus $1.156\mu s$ in the update-heavy workload. The advantage persists at 16 and 32 domains for balanced and update-heavy workloads, although pure-search at 32 domains slightly favors the mutex baseline. This supports the main thesis of the project: when one operation genuinely needs to update multiple linked locations atomically, MCAS can enable a cleaner non-blocking design and may beat a simple lock-based alternative once contention increases. Figure 4 makes this point more clearly than the snapshot plots, because the workload naturally benefits from a primitive that can change several pointers together.

Interpretation. The evaluation answers the research question with a nuanced "yes, but only when the abstraction is needed." MCAS is not a free substitute for a specialized algorithm. For

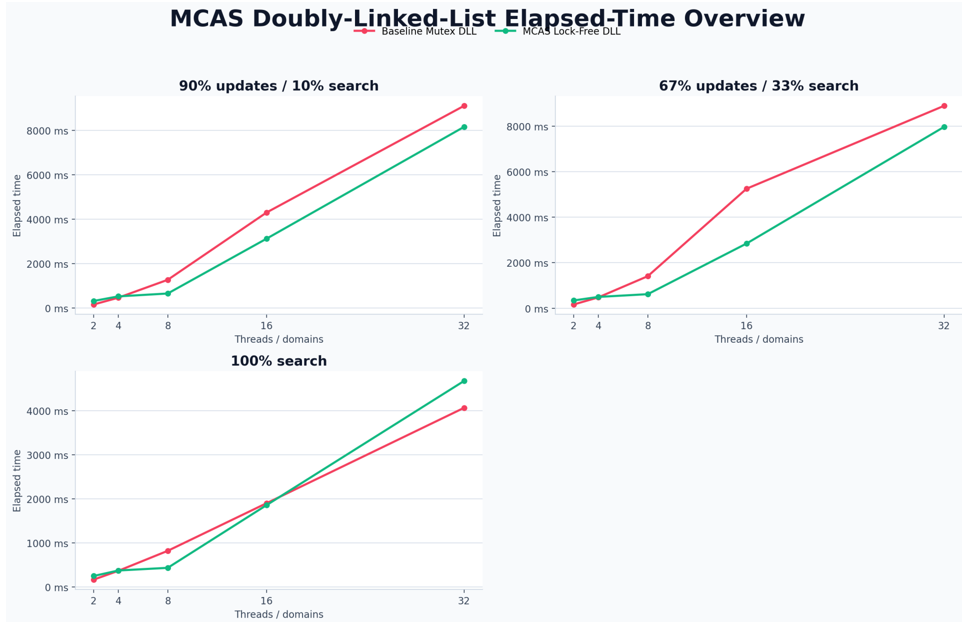

Figure 4: Overview of the doubly linked-list benchmark. This comparison is the strongest positive result for MCAS in our study: once updates touch several linked pointers, the lock-free MCAS design becomes competitive with, and often better than, the coarse-grained mutex baseline.

Benchmark	Workload	Domains	Baseline μs/op	MCAS μs/op
Snapshot	scans_100	32	0.067	6.197
Snapshot	balanced_50_50	32	0.097	4.845
SLL set	balanced_34_33_33	32	0.088	1.299
DLL set	balanced_34_33_33	8	1.061	0.494
DLL set	update_heavy_45_45_10	32	2.310	1.715

Table 1: Representative benchmark points extracted from the recorded CSV results. “Baseline” refers respectively to double-collect snapshot, the single-word-CAS linked list, and the coarse-grained mutex doubly linked list.

snapshots, the software layer mostly hurts performance because a double-collect scan is already simple and efficient. For the singly linked-list set, the single-word-CAS design already matches the data structure well. But for the doubly linked list, MCAS gives a natural way to synchronize several pointer updates without global locking, and the benchmark evidence suggests that this can pay off under contention.

5 Reflection on the Use of LLMs

Tools and models used. For the final consolidation of this project and report, we used Codex CLI with GPT-5.2 to inspect the repository, summarize parts of the paper, compare benchmark outputs, and tighten the prose. We did not rely on the LLM as an authority on correctness: every algorithmic claim in the report was cross-checked against the source code, the benchmark CSV files, and the cited paper.

What worked well. The LLM was most useful as a synthesis tool. It helped connect the code in `volatile/`, `snapshot/`, and the linked-list directories with the figures and bibliography already present in the project slides, which sped up the report-writing phase. It was also helpful

for turning raw benchmark numbers into concise comparative statements and for spotting the high-level theme that the doubly linked-list benchmark was the strongest evidence in favor of MCAS.

What did not work. The main failure mode was overconfident inference from incomplete cues. For example, the repository README and top-level `mcas.ml` suggest an older Harris–Fraser–Pratt path, while the actual working implementation is the concrete code in `volatile/mcas_volatile.ml`. An LLM can easily state the wrong algorithmic lineage if we do not verify against the real source files. It also tends to generate plausible but unsupported performance explanations unless the CSV logs are checked carefully.

What was surprising. It was surprisingly good at surfacing the implicit structure of the repository: which modules are executable artifacts, which are scaffolds, and which figures already summarize the evaluation. It was much less reliable on subtle semantic questions such as whether a particular retry loop is enough to justify a linearization argument.

What was difficult. The parts that still required genuine understanding were the concurrency arguments: why helping is necessary, why sorting addresses matters, what value a reader should return when it sees an in-progress word descriptor, and why MCAS helps more for doubly linked structures than for snapshots. Those points came from reading code and paper side by side, not from accepting LLM output at face value.

Your overall assessment. The LLM was a net help for synthesis, navigation, and drafting, but not a substitute for understanding non-blocking algorithms. We would use it again in the same supporting role, while continuing to verify every concurrency-specific claim manually.

6 Conclusions

Our answer to the research question is that software MCAS in OCaml 5 is valuable primarily as an *expressive* primitive. It lets us implement operations that genuinely need to update multiple locations together, and the doubly linked-list results show that this can outperform a coarse lock-based baseline under contention. However, the abstraction is expensive. In workloads where a specialized single-word-CAS or read-mostly algorithm already fits the problem well, MCAS mainly adds overhead. This is especially clear for snapshots, where double-collect is dramatically faster.

The main limitations of our work are that the benchmarking methodology is light weight (single recorded runs per configuration, limited hardware reporting, no variance analysis), the correctness evidence is empirical rather than formal, and the evaluation focuses on volatile memory only. Given more time, the next steps would be to add repeated benchmark trials with confidence intervals, formalize linearization points more explicitly, and explore whether the same MCAS substrate can support more complex structures such as trees or durable objects [6].

Contributions

Member	Contribution (%)	Main responsibilities
Sanjeev Reddy	34%	Core MCAS design, snapshot implementation, verification setup
Albin James	33%	Benchmarking, slide/report assets, evaluation analysis, literature survey
Kunal Umaji	33%	Linked-list variants, literature survey, report integration
Total	**100%**	

References

[1] Keir Fraser. Practical lock-freedom. 2004.

[2] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[3] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[4] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[5] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[6] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. volume 3, pages 1–26. ACM, 2019.