Saturn: a library of verified concurrent data structures for OCaml 5

Clément Allain (INRIA)
Vesa Karvonen (Tarides)
Carine Morel (Tarides)

August 1, 2024

1 Abstract

We present Saturn, a new OCaml 5 library available on `opam`. Saturn offers a collection of efficient concurrent data structures: stack, queue, skiplist, hash table, work-stealing deque, etc. It is well tested, benchmarked and in part formally verified.

2 Motivation

Sharing data between multiple threads or cores is a well-known problem. A naive approach is to take a sequential data structure and protect it with a lock. However, this approach is often inefficient in terms of performance, as locks introduce significant contention. Additionally, it may not be a sound solution as it can lead to liveness issues such as deadlock, starvation, and priority inversion.

In contrast, *lock-free* implementations, which rely on fine-grained synchronization instead of locks, are typically faster and guarantee system-wide progress. However, they are also more complex and come with their own set of bugs, such as the ABA problem (largely mitigated in garbage-collected languages), data races, and unexpected behaviors due to non-linearizability.

In this context, Saturn provides a collection of standard lock-free data structures, saving OCaml 5 programmers the trouble of designing their own. Currently, there is no similar project available for OCaml 5 in opam. Most OCaml 5 developers currently choose to write their own data structures, which is error-prone and time-consuming.

3 Library design

Saturn aims at covering a wide range of use cases, from simple stacks and queues to more complex data structures like skiplists and hash tables. More precisely, it currently features: (A) numerous queues: a queue based on the well-known Michael-Scott queue [3], a single-producer single-consumer queue, a multiple-producer single-consumer queue and a bounded queue; (B) a stack based on the Treiber stack [1]; (C) a work-stealing deque; (D) a bag; (E) a hash table; (F) a skiplist.

Most implementations are based on well-known algorithms. They have been adapted to work with and take advantage of the OCaml 5 memory model. For instance, we had to rework the Michael-Scott queue to avoid memory leaks.

Regarding performance, we are working on providing benchmarks for each Saturn's data structure (see section 4), and significant effort has been dedicated to micro-

optimization. In particular, we worked on (A) preventing false sharing[1], (B) adding fenceless atomic reads when possible, which improves performance on ARM processors, and (C) avoiding the extra indirection in arrays of atomics to reduce memory consumption. The feedback we produced while optimizing Saturn has highlighted some missing features in OCaml 5 and led to improvements in upstream OCaml (padded atomics, CSE bug fixed).

To explore some of these optimizations, we use unsafe features of the language (e.g., `Obj.magic`). Although we design our code to be memory-safe under regular use (e.g. only one domain can push at any given time in a single-consumer single-producer queue), we cannot offer the same level of guarantee as with the standard implementations. Consequently, some of Saturn's data structures have two versions: (1) a version that does not use any unsafe features of OCaml and (2) an optimized version. While most users should find the regular version efficient enough for their needs, adventurous users may prefer the optimized version, provided they encapsulate it correctly and verify their code somehow.

4 Benchmarks

As we are still in the experimental phase, we provide rough preliminary numbers to give an idea of the library performance. The following tables show the throughput of various queues and stacks implementations. The queue implementations benchmarked are: (1) the `Stdlib` queue (with one domain only), (2) the `Stdlib` queue protected with a mutex, (3) the lock-free Michael-Scott queue from Saturn (safe version), (4) a Michael-Scott two-stack-based queue (currently in this PR in Saturn). The stack implementations benchmarked are (1) the `Stdlib` stack (with one domain only), (2) the `Stdlib` stack protected with a mutex, (3) a concurrent stack implemented with an atomic list, (4) a lock-free Treiber stack from Saturn. The tests were run on an Intel i7-1270P (4P+8E cores) and an Apple M3 Max (6P+6P+4E cores) using OCaml 5.2.0 (see this repository if you want to run your own benchmarks).

Queue	Intel	Apple
Stdlib	61 M/s	64 M/s
Stdlib + mutex	24 M/s	19 M/s
Michael-Scott	22 M/s	32 M/s
Two-stack	37 M/s	56 M/s

Stack	Intel	Apple
Stdlib	66 M/s	72 M/s
Stdlib + mutex	24 M/s	24 M/s
Atomic list	52 M/s	66 M/s
Saturn Treiber	47 M/s	67 M/s

Table 1: Single domain benchmarks

There are several insights to be drawn from these results, but we will highlight a few key points. Firstly, for sequential programs, the `Stdlib` queue and stack are the fastest implementations as the concurrent implementations add significant overhead. However, the Saturn implementations consistently outperform the `Stdlib` ones protected with a single lock, even under low contention. Finally, the concurrent stack implemented with an atomic list performs comparably to the Treiber stack[2] from Saturn. In cases where a simple enough implementation exists, one might wonder why to use Saturn data structures instead of writing it oneself. However, even the atomic-list-based stack is optimized through (a) the use of `make_contended` to prevent false sharing, and (b) a backoff mechanism to reduce contention. Without these seemingly small optimizations, the atomic list implementation has a throughput of around 10 M/s regardless of contention (on the Intel

[1] False sharing occurs when different domains access different data items contained in the same cache line, forcing unnecessary synchronization. To prevent this, these data must be padded to ensure they are not in the same cache line.

[2] The Treiber stack is essentially a well-optimized atomic list.

machine), which is significantly lower than the Treiber stack's performance. In addition, the Saturn library provides thorough testing and could provide even more optimized implementations in the future.

Config	Queue	Intel	Apple	Stack	Intel	Apple
1 adder, 1 taker	Stdlib + mutex	6.1	14	Stdlib + mutex	2.7	18
	Michael-Scott	19	45	Atomic list	66	140
	Two-stack	40	110	Treiber	70	128
1 adder, 2 takers	Stdlib + mutex	3.1	3.2	Stdlib + mutex	3.1	4.0
	Michael-Scott	18	16	Atomic list	49	113
	Two-stack	36	102	Treiber	46	104
2 adders, 1 taker	Stdlib + mutex	5.8	5.8	Stdlib + mutex	6.5	7.7
	Michael-Scott	9.9	24	Atomic list	52	120
	Two-stack	17	89	Treiber	60	114
2 adders, 2 takers	Stdlib + mutex	3.6	6.0	Stdlib + mutex	3.6	7.7
	Michael-Scott	8.2	29	Atomic list	41	107
	Two-stack	17	97	Treiber	43	99

Table 2: Benchmarks with multiple domains in parallel (in millions of messages per second)

5 Tests

In multicore programming, it is essential to test not only the safety of the data structures but also to verify *linearizability*[3] [2] and *lock-freedom*[4] when expected. To achieve this, Saturn has been thoroughly tested using two primary tools: DSCheck and STM.

STM is used not only for unit testing but also for linearizability. It automatically generates random full programs using the provided API—in the case of Saturn, a data structure. These programs are executed in parallel with two domains and all results are checked against the postconditions of each function, providing unit testing. Simultaneously, STM verifies linearizability by ensuring that all intermediate states can be explained by a sequential execution of the calls. The STM test for the Treiber stack are a good example of how simple this is to write.

DSCheck is a model checker based on the DPOR[5] algorithm [4]. It is designed to compute all possible interleavings of instructions between multiple domains and verify that each one returns the expected result. This is particularly useful for catching elusive bugs that occur only in specific, rare interleavings. Additionally, DSCheck can be used to verify that a program is lock-free, as it will fail to terminate if any form of blocking is present. This is a bit more cumbersome to use than STM (see the DSCheck tests for the Treiber stack) but it is still a powerful tool. DSCheck implementation has been optimized[6] to make the tests quick enough to be used even on the more complex data structures of Saturn[7].

6 Formal verification

Lock-free algorithms are notoriously difficult to get right. To provide stronger guarantees, we have verified part of Saturn's data structures and aim at covering the entire library.

[3] See section 6 for the definition of linearizability.
[4] Roughly, lock-freedom guarantees system-wide progress. For more details, see Wikipedia.
[5] DPOR stands for Dynamic Partial-Order Reduction
[6] See the PRs about source sets and granular dependency relation.
[7] See the skiplist DSCheck tests.

At the time of writing, this effort essentially comprises concurrent bags, stacks and queues. One important benefit is that we get formal specifications for verified data structures.

The standard correctness criterion for concurrent data structures is *linearizability* [2]. It requires each operation on a data structure to appear to take effect instantaneously at some point during its execution, called the *linearization point*, such that the linearization points of all operations form a coherent sequential history.

To verify this criterion, we rely on Iris [6], a state-of-the-art mechanized *concurrent separation logic*. Iris has been successfully used in the past to verify realistic data structures [10, 11, 9]. All proofs are formalized in Coq and available on `github`.

Concretely, we first translate the original code from Saturn to a deeply embedded language in Coq. At the time of writing, this translation is manual but could be automated. It preserves the essence of the implementation, focusing on the most important operations and omitting minor aspects not affecting the correctness. For instance, consider the following `push` function from the implementation of a concurrent stack:

```
let rec push t v =
  let old = Atomic.get t in
  let new_ = v :: old in
  if not (Atomic.compare_and_set t old new_) then (
    Domain.cpu_relax () ;
    push t v
  )
```

In Coq, `push` translates to the `stack_push` function:

```
Definition stack_push : val :=
  rec: "stack_push" "t" "v" =>
    let: "old" := !"t" in
    let: "new" := ‘Cons( "v", "old" ) in
    ifnot: CAS "t" "old" "new" then (
      Yield ;;
      "stack_push" "t" "v"
    ).
```

The Iris way to formulate linearizability is through *logically atomic specifications* [5], which have been proven [8] to be equivalent to linearizability in sequentially consistent memory models. For instance, the specification of `stack_push` takes the following form:

$$\frac{\frac{\frac{\frac{\{\,\text{stack-inv}\ t\,\}}{\langle\, vs.\,\text{stack-model}\ t\ vs\,\rangle}}{\texttt{stack\_push}\ t\ v}}{\langle\,\text{stack-model}\ t\ (v :: vs)\,\rangle}}{\{\,().\,\text{True}\,\}}$$

Similarly to Hoare triples, the two assertions inside curly brackets represent the precondition and postcondition for the caller. For this particular operation, the postcondition is trivial. The stack-inv t precondition is the stack invariant. Intuitively, it asserts that t is a valid concurrent stack. More precisely, it enforces a set of logical constraints—a concurrent protocol—that t must respect at all times.

The other two assertions inside angle brackets represent the *atomic precondition* and *atomic postcondition*. They specify the linearization point of the operation: during the execution of `stack_push`, the abstract state of the stack held by stack-model is atomically updated from vs to $v :: vs$; in other words, v is atomically pushed at the top of the stack.

As a final note, we emphasize that our verification assumes a sequentially consistent memory model. However, OCaml 5's weak memory model has been formalized [7] in Iris. Prior work [9] has shown how to extend logically atomic specifications in this setting. Adapting our specifications and proofs should be rather straightforward and is future work.

7 Acknowledgments

Saturn has been primarily developed and maintained by Vesa Karvonen and Carine Morel, and previously by Bartosz Modelski. The work on formal verification has been conducted by Clément Allain. We would like to thank Gabriel Scherer and the anonymous reviewers for their feedback.

References

[1] Robert K. Treiber. *Systems programming: Coping with parallelism*. Tech. rep. RJ 5118. IBM Almaden Research Center, 1986.

[2] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. URL: https://doi.org/10.1145/78969.78972.

[3] Maged M. Michael and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), pp. 267–275. URL: https://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf.

[4] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 110–121. ISBN: 158113830X. URL: https://doi.org/10.1145/1040305.1040315.

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 207–231. URL: https://doi.org/10.1007/978-3-662-44202-9%5C_9.

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

[7] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Cosmo: a concurrent separation logic for multicore OCaml". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 96:1–96:29. URL: https://doi.org/10.1145/3408978.

[8] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. "Theorems for free from separation logic specifications". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. URL: https://doi.org/10.1145/3473586.

[9] Glen Mével and Jacques-Henri Jourdan. "Formal verification of a concurrent bounded queue in a weak memory model". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. URL: https://doi.org/10.1145/3473571.

[10] Simon Friis Vindum and Lars Birkedal. "Contextual refinement of the Michael-Scott queue (proof pearl)". In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. Ed. by Catalin Hritcu and Andrei Popescu. ACM, 2021, pp. 76–90. URL: https://doi.org/10.1145/3437992.3439930.

[11] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. "Mechanized verification of a fine-grained concurrent queue from meta's folly library". In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 100–115. URL: https://doi.org/10.1145/3497775.3503689.