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

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

May 31, 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 is often not the best solution. First, if performance is a concern, this approach is likely to be inefficient as locks introduce significant contention. Second, 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 relying on fine-grained synchronisation rather than locks are typically faster and guaranty system-wide progress. Yet, they are also more complex and come with their own set of bugs: ABA problem (largely mitigated in garbage-collected languages), data races, 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.

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 [2], 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,

[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.

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, (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 Implementation	**Intel i7-1270P**	**Apple M3**
Stdlib queue	61 M/s	64 M/s
Stdlib queue with mutex	24 M/s	19 M/s
Saturn Michael-Scott queue	22 M/s	32 M/s
Two_stack queue	37 M/s	56 M/s

Table 1: Message over time (million messages per second) for several queue implementations on a single domain

There are many 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 largely 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]. There is still a benefit to using Saturn data structures in this case: even this basic implementation 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, which is significantly lower than the Treiber stack's performance.

5 Tests

In multicore programming, it is essential to test not only the safety of the data structures but also to verify linearizability and lock-freedom when expected. To achieve this,

[2]The Treiber stack is essentially an atomic list.

Stack Implementation	Intel i7-1270P	Apple M3
Stdlib stack	66 M/s	72 M/s
Stdlib stack with mutex	24 M/s	24 M/s
Atomic list	52 M/s	66 M/s
Saturn Treiber stack	47 M/s	67 M/s

Table 2: Message over time (million messages per second) for several stack implementations on a single domain

Configuration	Queue Implementation	Intel i7-1270P	Apple M3
1 adder, 1 taker	Stdlib queue with mutex	6.1 M/s	14 M/s
	Saturn Michael-Scott queue	19 M/s	45 M/s
	Two_stack queue	40 M/s	110 M/s
1 adder, 2 takers	Stdlib queue with mutex	3.1 M/s	3.2 M/s
	Saturn Michael-Scott queue	18 M/s	16 M/s
	Two_stack queue	36 M/s	102 M/s
2 adders, 1 taker	Stdlib queue with mutex	5.8 M/s	5.8 M/s
	Saturn Michael-Scott queue	9.9 M/s	24 M/s
	Two_stack queue	17 M/s	89 M/s
2 adders, 2 takers	Stdlib queue with mutex	3.6 M/s	6.0 M/s
	Saturn Michael-Scott queue	8.2 M/s	29 M/s
	Two_stack queue	17 M/s	97 M/s

Table 3: Message over time (million messages per second) for several stack implementations on a multiple domain running in parallel

Configuration	Stack Implementation	Intel i7-1270P	Apple M3
1 adder, 1 taker	Stdlib stack with mutex	2.7 M/s	18 M/s
	Atomic list	66 M/s	140 M/s
	Saturn Treiber stack	70 M/s	128 M/s
1 adder, 2 takers	Stdlib stack with mutex	3.1 M/s	4.0 M/s
	Atomic list	49 M/s	113 M/s
	Saturn Treiber stack	46 M/s	104 M/s
2 adders, 1 taker	Stdlib stack with mutex	6.5 M/s	7.7 M/s
	Atomic list	52 M/s	120 M/s
	Saturn Treiber stack	60 M/s	114 M/s
2 adders, 2 takers	Stdlib stack with mutex	3.6 M/s	7.7 M/s
	Atomic list	41 M/s	107 M/s
	Saturn Treiber stack	43 M/s	99 M/s

Table 4: Messages over time (million messages per second) for several stack implementations running in parallel

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[3] algorithm [3]. It is designed to compute all possible interleavings of a given program 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[4] to make the tests quick enough to be used even on the more complex data structures of Saturn (see the skiplist DSCheck tests).

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. Moreover, one important benefit is that we get formal specifications for verified data structures.

This verification effort has been conducted using Iris [4], a state-of-the-art mechanized *concurrent separation logic*. In particular, all proofs are formalized in Coq.

A common criterion to specify concurrent operations on shared data structures is *linearizability*. The equivalent Iris notion is *logical atomicity*: there exists a point in time when a concurrent operation atomically takes effect (the linearization point). This statement takes the form of an *atomic specification*:

$$\frac{\frac{\frac{\frac{\{\,\text{queue-inv } t\,\}}{\langle\,\forall vs.\,\text{queue-model } t\ vs\,\rangle}}{\texttt{queue\_push}\ t\ v}}{\langle\,\text{queue-model } t\ (vs +\!\!+ [v])\,\rangle}}{\{\,().\,\text{True}\,\}}$$

In this example, we specify the `queue_push` operation from an implementation of a concurrent queue. Similarly to Hoare triples, the two assertions inside curly brackets express the precondition and postcondition. Here, the queue-inv t precondition represents the queue invariant. As it is persistent, we do not need to give it back in the postcondition. The other two assertions inside angle brackets express the *atomic precondition* and *atomic postcondition*. Basically, they specify the linearization point of the operation: during the execution of `queue_push`, the logical model of the queue is atomically updated from vs to $vs +\!\!+ [v]$, in other words v is atomically pushed at the back of the queue.

As a final note, we emphasize that our verification assumes a sequentially consistent memory model. Nevertheless, OCaml 5's relaxed memory model has been formalized [5] in Iris. It should be possible and is future work to adapt our specifications and proofs to support it.

[3] DPOR stands for Dynamic Partial-Order Reduction

[4] See the PRs about source sets and granular dependency relation.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.