Scal: A Benchmarking Suite for Concurrent Data Structures

Andreas Haas, Thomas Hütter, Christoph M. Kirsch,
Michael Lippautz, Mario Preishuber, and Ana Sokolova

University of Salzburg
`firstname.lastname@cs.uni-salzburg.at`

Abstract. Concurrent data structures such as concurrent queues, stacks, and pools are widely used for concurrent programming of shared-memory multiprocessor and multicore machines. The key challenge is to develop data structures that are not only fast on a given machine but whose performance scales, ideally linearly, with the number of threads, cores, and processors on even bigger machines. Part of that challenge is to provide a common ground for systematically evaluating the performance and scalability of new concurrent data structures and comparing the results with the performance and scalability of existing solutions. For this purpose, we have developed Scal which is an open-source benchmarking framework that provides (1) software infrastructure for executing concurrent data structure algorithms, (2) workloads for benchmarking their performance and scalability, and (3) implementations of a large set of concurrent data structures. We discuss the Scal infrastructure, workloads, and implementations, and encourage further use and development of Scal in the design and implementation of ever faster concurrent data structures.

1 Introduction

We describe Scal[1], an open-source benchmarking framework for evaluating performance and multicore scalability of concurrent data structures such as concurrent queues, stacks, and pools. With (multicore) scalability we mean that performance grows (ideally linearly) with the number of threads increasing.

Scal provides:

1. Infrastructural software for scalable memory allocation and computational load generation, as well as tagging for atomicity and operation logging. Here, by scalable memory allocation and computational load generation we mean constant overhead independent of the number of threads.
2. Workloads for benchmarking concurrent data structures such as, for example, producer-consumer scenarios.
3. Concurrent data structure implementations, like (relaxed) queues, stacks, and pools, listed in Table 1.

[1] The Scal homepage is at `http://scal.cs.uni-salzburg.at`, the Scal code is publicly available at `http://github.com/cksystemsgroup/scal`.

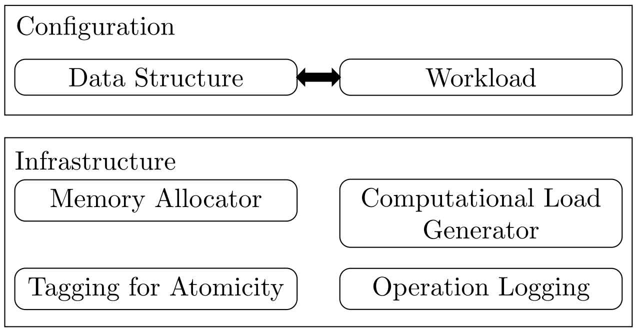


Fig. 1: Architecture of Scal

Each pair of a workload and a concurrent data structure defines a *configuration*. Hence, Scal provides infrastructure and configurations, as shown in Figure 1.

Scal requires an x86 machine and Posix threads. It has been successfully run on Intel and AMD machines with Linux Ubuntu 12.04 and 14.04. Porting Scal to other architectures and operating systems should be easily possible.

Scal reports temporal performance by measuring total execution time and calculating throughput. Time is measured (using standard OS primitives) from the moment that all threads are created and configured, until the moment that the last thread terminates. Creation and configuration is done sequentially by a single initialization thread. All other threads wait on a barrier after they have been created. As soon as the configuration is complete, the initialization thread records the (start) time and releases the barrier. The last thread that terminates records the (end) time again right before terminating. Total execution time is then reported as the difference between the end time and the start time. Throughput is the total execution time divided by the total number of data-structure operations performed by all threads. All Scal workloads determine (configure) the total number of operations to be performed, which enables throughput calculation. Obtaining meaningful temporal performance results requires disabling CPU frequency scaling (in addition to using scalable memory allocation and load generation).

The Scal benchmarking framework was originally designed for the evaluation of concurrent (relaxed) data structures [9,16,12,15,8,3,7]. Note that Scal not only contains our newly developed data structure algorithms, but also many other state-of-the-art concurrent data structure implementations (cf. Table 1). We are aware of two other recently developed benchmarking frameworks for a similar purpose. The sim-universal-construction framework has been designed to develop and evaluate wait-free algorithms [4]. It has also been used to evaluate lock-free data structures [23]. The framework provides a subset of Scal's capabilities such as portable abstractions for atomic operations like `fetch-and-inc` or `compare-and-swap` (covered by tagging for atomicity in Scal) and implements lock-free queues such as the Michael-Scott queue [21]. However, there is no scalable memory allocation, computational load generation, and predefined work-

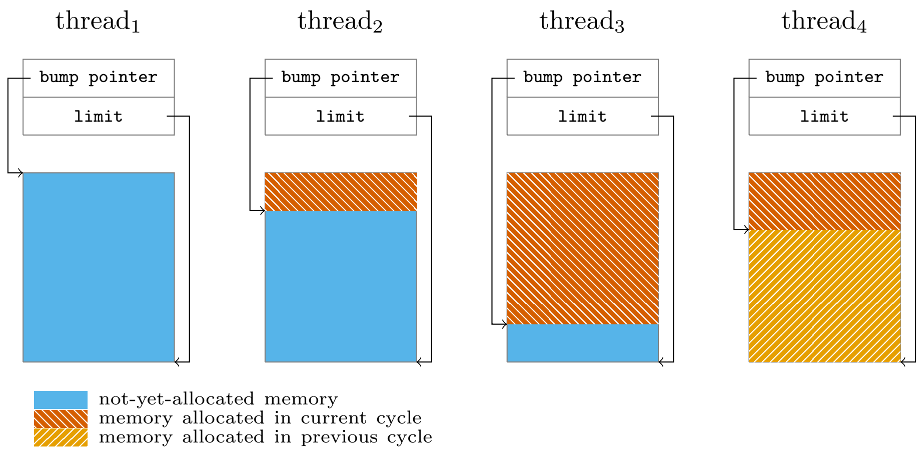


Fig. 2: Memory Allocation in Scal

loads. The SynchroBench [5] framework has been designed with the same goals as Scal. It provides a broad spectrum of concurrent data structure implementations such as linked lists, skiplists, trees, and hash tables. SynchroBench uses transactional memory libraries and does not provide scalable memory allocation, which makes the framework less suitable for high-performance benchmarking. SynchroBench also does not provide predefined workloads.

2 Scal Infrastructure

Next we describe the Scal infrastructure, in particular memory allocation, computational load generation, tagging for atomicity, and operation logging.

All experiments reported here ran on a unified memory architecture (UMA) machine with four 10-core 2GHz Intel Xeon E7-4850 processors supporting two hardware threads (hyperthreads) per core, 128GB of main memory, and Linux kernel version 3.8.0. All measurements are averaged over ten runs and reported as arithmetic mean including the 95% confidence interval (based on corrected sample standard deviation).

To this end, let us note that Scal is open-source and publicly available, and hence also open to improvement and extensions. New utilities may be developed as the demand grows.

2.1 Memory Allocator

The memory allocator of Scal is a special purpose concurrent allocator that performs cyclic allocation [24].

Ideally, in a benchmarking framework, memory management operations take zero time and do not depend on the number of threads. In Scal, memory allocation is done thread-locally with negligible overhead (that does not depend on the number of threads). Figure 2 shows the memory layout in Scal. Each thread in Scal gets its own heap for thread-local allocation. The heaps are initialized upon thread startup with preallocated fixed-size memory chunks that are optionally accessed (by reading single words) to warm up operating system pages. Each heap consists of a `bump pointer`, as well as a `start` and an `end` indicating the beginning and the end of the memory chunk.

Upon memory allocation, the bump pointer is incremented by the required size and the bump pointer value at the start of the allocation operation is returned as memory address[2]. Upon reaching the limit the bump pointer is reset to the beginning of the memory chunk.

Cyclic allocation is sound as long as no allocated and still live memory gets reallocated, i.e., the bump pointer returns addresses to dead memory. For any benchmark that terminates in finite time, there is a heap size such that cyclic allocation with that heap size is sound. In order to determine the heap size sufficient for sound cyclic allocation, Scal provides a configuration mode in which, instead of resetting the bump pointer (cyclic allocation), the heap expands by obtaining another memory chunk from the OS. This configuration mode is not scalable because of the overhead involved in obtaining memory chunks.

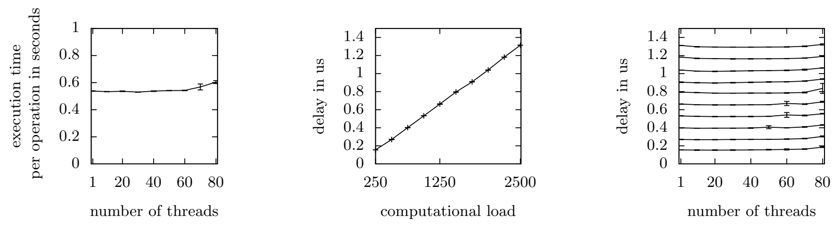
Cyclic allocation does not require explicit deallocation. Nevertheless, for saving memory, Scal provides a limited form of explicit deallocation: A free call (without arguments) rolls back the bump pointer to the value before the last allocation. Consecutive free calls without allocation in between have no effect. This form of explicit deallocation provides the benefit of keeping the size of the needed heap for sound cyclic allocation small, in particular with algorithms where many threads allocate memory within concurrent operations of which only one succeeds. The failing threads can then roll back, i.e., deallocate the most recently allocated memory.

In order to demonstrate the scalability of the allocator in Scal we designed a benchmark where each thread executes ten million allocation operations of the size required to accommodate a node of a Michael-Scott queue [21]. After each allocation operation, a computation is performed that simulates application activity and reduces the load on the allocator. Note that for many allocators, there is a computational load that renders them scalable. However, the smaller that load is, the more load can be put on the benchmarked concurrent data structure without introducing performance artifacts of the allocator. The computational load for which Scal's memory allocator is scalable is small. Figure 3a illustrates the scalability (constant overhead) of the allocator. We discuss computational load generation and the computational load used in Figure 3a in the next section.

2.2 Computational Load

Scal provides a primitive that simulates computational load resulting in a time delay. This enables exercising the concurrent data structures in different contention scenarios: In between any two data structure operations, the computa-

[2] This is standard bump pointer allocation.



(a) Memory allocation: Scalability for an increasing number of threads (b) Computational load: Delay for varying computational load (c) Computational load: Scalability for an increasing number of threads

Fig. 3: Evaluation of Scal's infrastructure

tional load imitates application behavior, i.e., a real computation. The higher the load, the lower the contention on the data structure.

The computational load primitive has a unit-less input that translates to a time delay that is close to linear in the input, as shown in Figure 3b. Moreover, the linear relationship remains the same independent of the number of threads concurrently using the primitive as shown in Figure 3c, even on hyper-threaded machines. In the figure we present many (pretty straight) curves, one for each data point of Figure 3b. Each curve shows the time delay for a given unit-less input value of the computational load for an increasing number of threads. The linearity is demonstrated as the curves are close to equally distant for equally distant inputs. The scalability is demonstrated by the fact that each curve is close to a constant line.

Our computational load primitive uses the x86 CPU instruction (RDTSC) that reads the time stamp counter (TSC) from the corresponding register of the processor. The counter represents the number of cycles since the last processor reset. Note that TSC only relates linearly to actual time for a constant clock speed. This is one of the reasons for disabling CPU frequency scaling in Scal.

The computational load primitive implements a busy wait on the value of TSC obtained from RDTSC. Additionally, the processor is informed that it currently executes a busy wait, potentially reducing CPU resources needed to execute the code fragment, making the computational load primitive scalable for hyper-threaded machines.

2.3 Pointer Tagging

Concurrent programs may be subject to the ABA problem: Finitely many names (memory addresses) for an infinite state space which requires eventual reuse of names for different states. Ideally, this is prevented (by hazard pointers [20]) or, less ideally but more practically, it is made less likely through versioning.

Hazard pointers solve the problem by only allowing name reuse when there is proof that the name is not in use any more. Versioning makes the occurrence of ABA less likely by increasing the set of names via adding version numbers. As implementing hazard pointers is costly, i.e., it requires significant bookkeeping, versioning is the common approach to fighting ABA.

Most concurrent data-structure algorithms, in particular the lock-free ones, use versioning to address ABA [19]. To ease the programming of algorithms that require versioning, Scal provides a versioning utility, that we refer to as *pointer tagging*, as part of its infrastructure.

In particular, Scal provides 16-bit version tags for values with up to 48 bits, assembling value and tag in a single 64-bit word so that all standard atomic operations still work atomically on the pair of a value and a version tag. Note that this indeed enlarges the set of names, as current 64-bit operating systems limit address spaces to 48 bits.

Note that taking care of versioning by hand in concurrent algorithms is a common source of bugs. The pointer tagging of Scal relieves the programmer from such a burden, and the careful handling of version tags is implicitly done by Scal itself.

2.4 Operation Logging

In order to investigate the detailed behavior of a single run, and even individual operations, Scal provides the utility of *operation logging*.

For each concurrent data-structure operation, Scal provides functions for thread-locally logging the type of an operation, e.g., insert or remove, the invocation time, the response time, a linearization point. Some of these functions are added automatically as soon as operation logging is enabled, e.g., logging of invocation and response time. For others, the programmer can use the Scal functions to annotate the code of a concurrent algorithm, e.g., if linearization points are known within the code. Operation logging only incurs negligible overhead, since all data is stored in pre-allocated memory at runtime and only output into a file upon termination.

Operation logging allows to experimentally validate different metrics on concurrent executions. See [9] for definitions and experimental evaluation of such metrics. Last but not least, operation logging may be useful to the programmer (and has been useful to some of us) when debugging concurrent algorithms.

3 Scal Workloads

There are two generic configurable workloads in Scal, a classical producer-consumer workload, and a sequential alternating workload. We describe both below.

3.1 Producer-Consumer

In the producer-consumer workload, as usual, some threads are producers and some consumers. Scal allows configuring the number of producers, consumers, the computational load, and the number of elements to be produced per producer thread. Each producer then inserts its produced elements into the concurrent

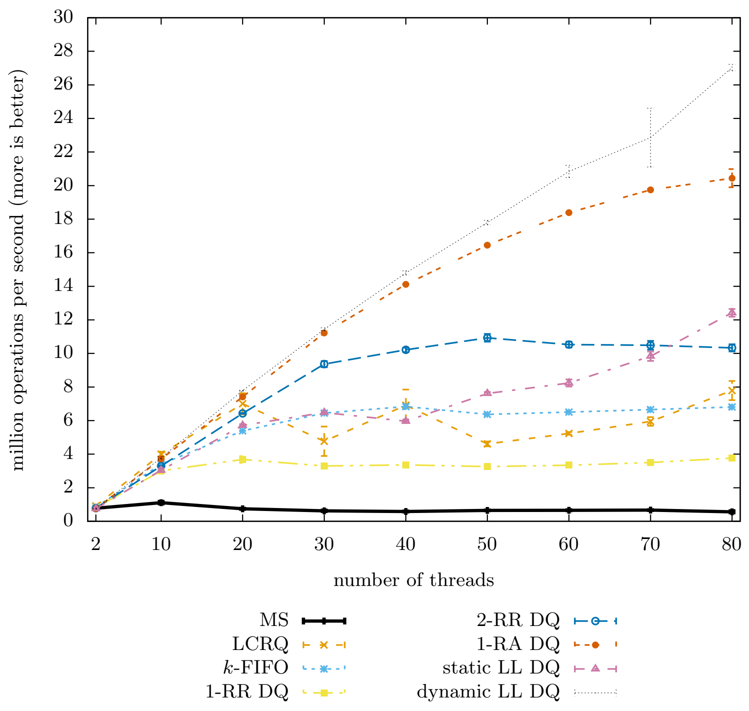


Fig. 4: Performance and scalability in a producer-consumer benchmark for a number of queue and queue-like data structures, for an increasing number of threads

data structure. Each consumer retrieves its fair share of elements (equal to the total number of elements produced divided by the number of consumers). Residual elements are discarded, i.e., they are left in the data structure. The configured computational load is executed in between any two operations performed. Additionally, the producer-consumer benchmark allows to insert a barrier between producing and consuming threads, for measuring either producing or consuming (or both) of elements separately.

Figure 4 shows the results of an exemplary scalability measurement of the producer-consumer benchmark for an increasing number of threads of which half are producers and half consumers, for a computational load of 1000, and 1 million elements inserted per producer thread.

In this benchmark, Scal reports the total number of performed operations divided by the total execution time. To this end, we note that changing Scal to report other data, e.g. (average) number of operations per thread per unit of time, is a matter of changing one line of code.

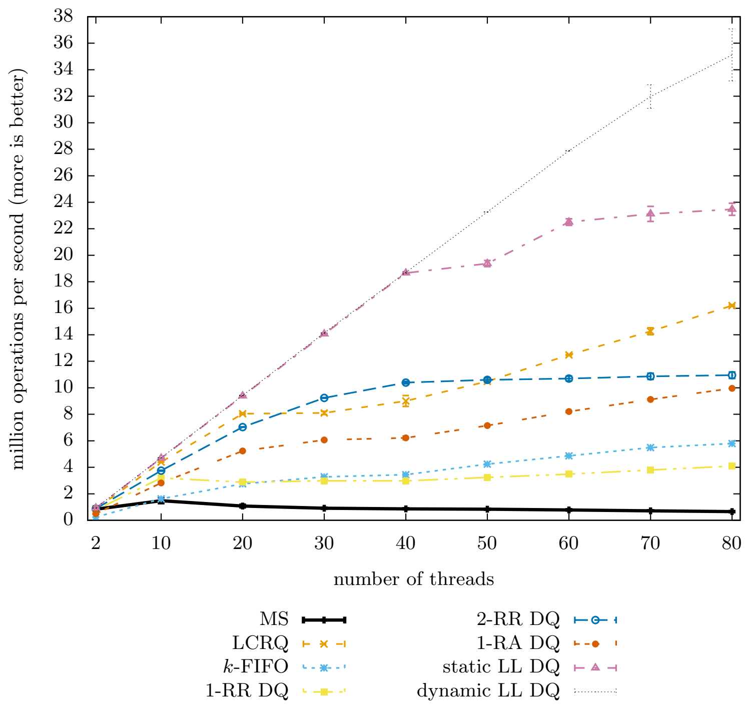


Fig. 5: Performance and scalability in a sequential alternating benchmark for a number of queue and queue-like data structures, for an increasing number of threads

3.2 Sequential Alternating

The sequential alternating workload is designed so that each thread alternates between an insert operation and a remove operation.

For sequential alternating, similar to the producer-consumer workload, Scal allows configuring the number of threads, the computational load, and the number of operations (pairs of consecutive insert and remove operation) per thread. Additionally, the sequential alternating workload allows the data structure to be prefilled with a specified amount of elements. Such an option could easily be added to the producer-consumer benchmark as well, it was just never needed for our experimental purposes. The computational load is again computed in between any two operations.

Figure 5 shows the results of an exemplary scalability measurement of the sequential alternating benchmark for an increasing number of threads, computational load of 1000, and 1 million pairs of consecutive insert and remove operations per thread.

Name	Semantics	Year	Ref.
Lock-based Singly-linked List Queue	strict queue	1968	[17]
Michael Scott (MS) Queue	strict queue	1996	[21]
Flat Combining Queue	strict queue	2010	[10]
Wait-free Queue	strict queue	2012	[18]
Linked Cyclic Ring Queue (LCRQ)	strict queue	2013	[23]
Timestamped (TS) Queue	strict queue	2015	[3]
Cooperative TS Queue	strict queue	2015	[6]
Segment Queue	k-relaxed queue [12,1]	2010	[1]
Random Dequeue (RD) Queue	k-relaxed queue [12,1]	2010	[1]
Bounded Size k-FIFO Queue	k-relaxed queue [12,1], pool	2013	[15]
Unbounded Size k-FIFO Queue	k-relaxed queue [12,1], pool	2013	[15]
b-RR Distributed Queue (DQ)	k-relaxed queue [12], pool	2013	[8]
Least-Recently-Used (LRU) DQ	k-relaxed queue [12], pool	2013	[8]
Locally Linearizable DQ (static, dynamic)	locally linearizable queue [7], pool	2015	[7]
Locally Linearizable k-FIFO Queue	locally linearizable queue [7], k-relaxed queue [12], pool	2015	[7]
Relaxed TS Queue	quiescently consistent queue (conjectured)	2015	[6]
Lock-based Singly-linked List Stack	strict stack	1968	[17]
Treiber Stack	strict stack	1986	[26]
Elimination-backoff Stack	strict stack	2004	[11]
Timestamped (TS) Stack	strict stack	2015	[3]
k-Stack	k-relaxed stack [12]	2013	[12]
b-RR Distributed Stack (DS)	k-relaxed stack [12], pool	2013	[8]
Least-Recently-Used (LRU) DS	k-relaxed stack [12], pool	2013	[8]
Locally Linearizable DS (static, dynamic)	locally linearizable stack [7], pool	2015	[7]
Locally Linearizable k-Stack	locally linearizable stack [7], k-relaxed queue [12], pool	2015	[7]
Timestamped (TS) Deque	strict deque (conjectured)	2015	[6]
d-RA DQ and DS	strict pool	2013	[8]

Table 1: Concurrent Data Structures in Scal

4 Concurrent Data Structure Implementations in Scal

All Scal implementations of concurrent data structures are listed in Table 1. We distinguish between strict queues, relaxed queues, strict stacks, relaxed stacks, a strict deque (conjectured), and strict pools. By strict we mean data structures that are linearizable [14] with respect to a sequential specification of a queue, stack, deque, or pool, respectively. Strict concurrent data structures often lack performance and scalability [25] as they require significant synchronization [2]. A common trend in the design of concurrent data structures chooses to relax the semantics for gain in performance and scalability. The relaxations could affect the sequential specification [12,1,22] or the consistency condition [7,13]. Note that most data structures in Scal are lock free [13]. In the sequel, we discuss all implemented data structures in some detail. The table shows references for each data structure, which we omit in the text below.

4.1 Strict Queues

As baseline for queues, there is a standard lock-based implementation of a queue based on a singly-linked list. As lock-free baseline for queues, we have implemented the Michael-Scott queue. The flat combining queue is another well-known strict queue. The wait-free queue is a strict queue based on the Michael-Scott queue whose wait freedom is achieved by faster threads helping slower ones to complete their operations. The linked list cyclic ring queue is a fast lock-free queue that operates on very large cyclic buffers and uses fetch-and-add as basic synchronization primitive. The (cooperative) timestamped queue is a fast lock-free queue that uses timestamps to achieve queue order.

4.2 Relaxed Queues

There are several variants of relaxed queues in Scal:

- A number of k-relaxed queues that are linearizable with respect to the k-out-of-order relaxation of the sequential specification of a queue [12]. In a k-relaxed queue, one of the $k+1$-oldest elements is returned upon a dequeue operation.
- Several queues that are locally linearizable with respect to the sequential specification of a queue, and a relaxed queue that is conjectured to be quiescently consistent.

The segment queue and the random dequeue queue are k-relaxed but do not provide a linearizable emptiness check and hence are not linearizable pools. All other relaxed queues in Scal are linearizable pools.

The bounded and unbounded size k-FIFO queues are lock-free k-relaxed queues related to the segment queue. They implement a Michael-Scott queue of segments of size k.

The b-RR distributed queue and the least-recently-used distributed queue are members of the distributed queues (DQ) family. All data structures in the DQ family implement an array of Michael-Scott queues which are accessed using various load balancers. For these particular DQs, the load balancers enable proving a bound k for a k-relaxation.

The locally linearizable queues are variants of DQ and the k-FIFO queue. The locally linearizable DQ comes in two variants: with a static or dynamic array size (number of Michael-Scott queues). It is the load balancer(s) that make them locally linearizable. Finally, the relaxed TS queue is a relaxed timestamped queue that is conjectured to provide quiescent consistency.

4.3 Strict Stacks

As baseline for stacks, we have implemented a lock-based stack based on a singly linked list. As lock-free baseline for stacks, there is the Treiber stack. The elimination-backoff stack is a fast stack that utilizes the possibility of elimination, i.e., popping any element that is being concurrently pushed. The timestamped stack is a fast lock-free stack that uses timestamps to achieve stack order and also benefits from elimination.

4.4 Relaxed Stacks

Just like relaxed queues, also relaxed stacks come in two flavors:

- k-Relaxed stacks that relax the sequential specification to a k-out-of-order stack that allows for removing one of the $k+1$-youngest elements in the stack.
- Locally linearizable stacks.

All relaxed stacks in Scal are linearizable pools, in particular they provide linearizable emptiness checks.

The k-Stack is a typical k-relaxed stack implemented as a Treiber stack of segments of size k. The k-Stack has a linearizable emptiness check and is hence a linearizable pool. Just like for queues, there is a family of distributed stacks (DS) implemented as an array of Treiber stacks with different load balancers of which b-RR DS and least-recently-used DS are proven to be k-relaxed for a particular bound k depending on the parameters of the data structure.

Also here we have the same locally linearizable variants of stacks, namely the locally linearizable DQ with static and dynamic array size, and the locally linearizable k-Stack.

4.5 Strict Deque

The implementation of a strict deque in Scal is a timestamped implementation, combining the timestamped stack and timestamped queue. Proving the correctness (linearizability with respect to the data structure) was a highly nontrivial task for the timestamped stack, leading to a new theorem that provides sufficient conditions for stack linearizability. We conjecture that this combined deque implementation is linearizable with respect to a deque. The proof still remains to be done.

4.6 Strict Pools

All other variants of DQ and DS are very much relaxed queue-like or stack-like data structures, i.e., they can only be proven to be linearizable with respect to a pool. Currently d-RA DQ and DS are implemented in Scal. We have experimented with other implementations of pools as well. Their code is currently not part of Scal but will be added in the future.

5 Conclusions

We have presented Scal, an open-source benchmarking framework for evaluating the performance and scalability of concurrent data structures. Scal provides implementations of many concurrent data structures as well as the necessary infrastructure and relevant workloads for executing and benchmarking them. The framework has already enabled research that has lead to some of the concurrent data structures mentioned here. Scal is nevertheless only the starting point of a comprehensive benchmarking suite for concurrent data structures. The code is open source and may easily be extended with implementations of other concurrent data structures and enhanced with more infrastructure and workloads.

Acknowledgements

This work has been supported by a Google PhD Fellowship and the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF): S11404-N23 and S11411-N23).

References

1. Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.
2. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M.M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.
3. M. Dodds, A. Haas, and C.M. Kirsch. A scalable, correct time-stamped stack. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 233–246. ACM, 2015.
4. P. Fatourou and N.D. Kallimanis. A highly-efficient wait-free universal construction. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334. ACM, 2011.
5. V. Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM, 2015.
6. A. Haas. *Fast Concurrent Data Structures Through Timestamping.* PhD thesis, University of Salzburg, Salzburg, Austria, 2015.
7. A. Haas, T.A. Henzinger, A. Holzer, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith. Local linearizability. *CoRR*, abs/1502.07118, 2015.
8. A. Haas, T.A. Henzinger, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory—multicore performance and scalability through quantitative relaxation. In *Proc. International Conference on Computing Frontiers (CF)*. ACM, 2013.
9. A. Haas, C.M. Kirsch, M. Lippautz, and H. Payer. How fifo is your concurrent fifo queue? In *Proc. Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, pages 1–8. ACM, 2012.

10. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.
11. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215. ACM, 2004.
12. T.A. Henzinger, C.M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 317–328. ACM, 2013.
13. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
14. M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
15. C.M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-fifo queues. In *Proc. International Conference on Parallel Computing Technologies (PaCT)*, LNCS, pages 208–223. Springer, 2013.
16. C.M. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS. Springer, 2012.
17. D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley, Redwood City, CA, USA, 1997.
18. A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 141–150. ACM, 2012.
19. M.M. Michael. Aba prevention using single-word instructions. Technical Report RC 23089, IBM Research Center, 2004.
20. M.M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
21. M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.
22. M.M. Michael, M.T. Vechev, and V.A. Saraswat. Idempotent work stealing. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.
23. A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 103–112. ACM, 2013.
24. H.H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proc. International Symposium on Memory Management (ISMM)*, pages 15–30. ACM, 2007.
25. N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.
26. R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ-5118, IBM Research Center, 1986.