Relaxed Queues and Stacks from Read/Write Operations

Armando Castañeda
Instituto de Matemáticas, UNAM, Mexico City, Mexico
armando.castaneda@im.unam.mx

Sergio Rajsbaum
Instituto de Matemáticas, UNAM, Mexico City, Mexico
rajsbaum@matem.unam.mx

Michel Raynal
Institut Universitaire de France, IRISA-Université de Rennes, France
Polytechnic University of Hong Kong, Hong Kong
michel.raynal@irisa.fr

Abstract

Considering asynchronous shared memory systems in which any number of processes may crash, this work identifies and formally defines relaxations of queues and stacks that can be non-blocking or wait-free while being implemented using only read/write operations. Set-linearizability and Interval-linearizability are used to specify the relaxations formally, and precisely identify the subset of executions which preserve the original sequential behavior. The relaxations allow for an item to be returned more than once by different operations, but only in case of concurrency; we call such a property *multiplicity.* The stack implementation is wait-free, while the queue implementation is non-blocking. Interval-linearizability is used to describe a queue with multiplicity, with the additional relaxation that a dequeue operation can return *weak-empty*, which means that the queue *might* be empty. We present a read/write wait-free interval-linearizable algorithm of a concurrent queue. As far as we know, this work is the first that provides formalizations of the notions of multiplicity and weak-emptiness, which can be implemented on top of read/write registers only.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Asynchrony, Correctness condition, Linearizability, Nonblocking, Process crash, Relaxed data type, Set-linearizability, Wait-freedom, Work-stealing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.13

Related Version A full version of the paper is available at [15]. `https://arxiv.org/abs/2005.05427`.

Funding *Armando Castañeda*: Supported by UNAM-PAPIIT project IN108720.
Sergio Rajsbaum: Supported by UNAM-PAPIIT project IN106520.
Michel Raynal: Supported by French ANR project DESCARTES (16-CE40-0023-03).

1 Introduction

In the context of asynchronous crash-prone systems where processes communicate by accessing a shared memory, linearizable implementations of concurrent counters, queues, stacks, pools, and other concurrent data structures [32] need extensive synchronization among processes, which in turn jeopardizes performance and scalability. Moreover, it has been formally shown that this cost is sometimes unavoidable, under various specific assumptions [11, 12, 19]. However, often applications do not require all guarantees offered by a linearizable sequential specification [38]. Thus, much research has focused on improving performance of concurrent data structures by relaxing their semantics. Furthermore, several works have focused on relaxations for queues and stacks, achieving significant performance improvements (e.g., [21, 22, 28, 38]).

It is impossible however to implement queues and stacks with only Read/Write operations, without relaxing their specification. This is because queues and stacks have consensus number two (i.e. they allow consensus to be solved among two processes but not three), while the consensus number of Read/Write operations is only one [23], hence too weak to wait-free implement queues and stacks. Thus, atomic Read-Modify-Write operations, such as Compare&Swap or Test&Set, are required in any queue or stack implementation. To the best of our knowledge, even relaxed versions of queues or stacks have not been designed that avoid the use of Read-Modify-Write operations.

In this article, we are interested in exploring if there are meaningful relaxations of queues and stacks that can be implemented using only simple Read/Write operations, namely, if there are non-trivial relaxations with consensus number one. Hence, this work is a theoretical investigation of the power of the crash Read/Write model for relaxed data structures.

Contributions

We identify and formally define relaxations of queues and stacks that can be implemented using only simple Read/Write operations. We consider queue and stack relaxations with *multiplicity*, where an item can be extracted by more than one dequeue or pop operation, instead of exactly once. However, this may happen only in the presence of concurrent operations. As already argued [30], this type of relaxation could be useful in a wide range of applications, such as parallel garbage collection, fixed point computations in program analysis, constraint solvers (e.g. SAT solvers), state space search exploration in model checking, as well as integer and mixed programming solvers.

One of the main challenges in designing relaxed data structures lies in the difficulty of formally specifying what is meant by “relaxed specification”. To provide a formal specification of our relaxations, we use *set-linearizability* [33] and *interval-linearizability* [14], specification methods that are useful to specify the behavior of a data structure in concurrent patterns of operation invocations, instead of only in sequential patterns. Using these specification methods, we are able to precisely state in which executions the relaxed behavior of the data structure should take place, and demand a strict behavior (not relaxed), in other executions, especially when operation invocations are sequential.

First Contribution. We define a *set-concurrent stack with multiplicity*, in which no items are lost, all items are pushed/popped in LIFO order but an item can be popped by multiple operations, which are then concurrent. We define a *set-concurrent queue with multiplicity* similarly. In both cases we present set-linearizable implementations based only on Read/Write operations. The stack implementation is wait-free [23], while the queue implementation is non-blocking [25]. Our set-concurrent implementations imply Read/Write solutions for *idempotent work-stealing* [30] and *k-FIFO* [28] queues and stacks.

Second Contribution. We define an interval-concurrent queue with a *weak-emptiness check*, which behaves like a classical sequential queue with the exception that a dequeue operation can return a control value denoted *weak-empty*. Intuitively, this value means that the operation was concurrent with dequeue operations that took the items that were in the queue when it started, thus the queue might be empty. First, we describe a wait-free interval-linearizable implementation based on Fetch&Inc and Swap operations. Then, using the techniques in our set-linearizable stack and queue implementations, we obtain a wait-free interval-linearizable implementation using only Read/Write operations.

Our interval-concurrent queue with weak-emptiness check is motivated by a theoretical question that has been open for more than two decades [4]: it is unknown if there is a wait-free linearizable queue implementation based on objects with consensus number two (e.g. Fetch&Inc or Swap), for any number of processes. There are only such non-blocking implementations in the literature, or wait-free implementations for restricted cases (e.g. [10, 18, 29, 16, 17]). Interestingly, our interval-concurrent queue allows us to go from non-blocking to wait-freedom.

Since we are interested in the computability power of Read/Write operations to implement relaxed concurrent objects (that otherwise are impossible), our algorithms are presented in an idealized shared-memory computational model. We hope these algorithms will help to develop a better understanding of fundamentals that can derive solutions for real multicore architectures, with good performance and scalability.

Related Work

It has been frequently pointed out that classic concurrent data structures have to be relaxed in order to support scalability, and examples are known showing how natural relaxations on the ordering guarantees of queues or stacks can result in higher performance and greater scalability [38]. Thus, for the past ten years there has been a surge of interest in relaxed concurrent data structures from practitioners (e.g. [34]). Also, theoreticians have identified inherent limitations in achieving high scalability in the implementation of linearizable objects [11, 12, 19].

Some articles relax the sequential specification of traditional data structures, while others relax their correctness condition requirements. As an example of relaxing the requirement of a sequential data structure, [22, 27, 28, 35] present a *k-FIFO* queue (called *out-of-order* in [22]) in which elements may be dequeued out of FIFO order up to a constant $k \geq 0$. A family of relaxed queues and stacks is introduced in [39], and studied from a computability point of view (consensus numbers). It is defined in [22] the *k-stuttering* relaxation of a queue/stack, where an item can be returned by a dequeue/pop operation without actually removing the item, up to $k \geq 0$ times, even in sequential executions. Our queue/stack with multiplicity is a stronger version of k-stuttering, in the sense that an item can be returned by two operations if and only if the operations are concurrent. Relaxed priority queues (in the flavor of [39]) and associated performance experiments are presented in [6, 42].

Other works design a weakening of the consistency condition. For instance, *quasi-linearizability* [3], which models relaxed data structures through a distance function from valid sequential executions. This work provides examples of quasi-linearizable concurrent implementations that outperform state of the art standard implementations. A *quantitative* relaxation framework to formally specify relaxed objects is introduced in [21, 22] where relaxed queues, stacks and priority queues are studied. This framework is more powerful than quasi-linearizability. It is shown in [40] that linearizability and three data type relaxations studied in [22], k-Out-of-Order, k-Lateness, and k-Stuttering, can also be defined as consistency conditions. The notion of *local linearizability* is introduced in [20]. It is a relaxed consistency condition that is applicable to container-type concurrent data structures like pools, queues, and stacks. The notion of *distributional linearizability* [5] captures *randomized* relaxations. This formalism is applied to MultiQueues [37], a family of concurrent data structures implementing relaxed concurrent priority queues.

The previous works use relaxed specifications, but still sequential, while we relax the specification to make it concurrent (using set-linearizability and interval-linearizability).

The notion of *idempotent work stealing* is introduced in [30], where LIFO, FIFO and double-ended set implementations are presented; these implementations exploit the relaxed semantics to deliver better performance than usual work stealing algorithms. Similarly to our queues and stacks with multiplicity, the *idempotent* relaxation means that each inserted item is eventually extracted at least once, instead of exactly once. In contrast to our work, the algorithms presented in [30] use Compare&Swap (in the Steal operation). Being a practical-oriented work, formal specifications of the implemented data structures are not given.

Organization. The article is organized as follows. Section 2 presents the model of computation and the linearizability, set-linearizability and interval-linearizability correctness conditions. Sections 3 and 4 contain our Read/Write set-linearizable queue and stack implementations, and Section 5 explains some implications obtained from these implementations. Our interval-linearizable queue implementation is presented in Section 6. Finally, Section 7 ends the paper with some final remarks. Full proofs of our claims can be found in the full version of the paper [15].

2 Preliminaries

Model of Computation. We consider the standard concurrent system model with n *asynchronous* processes, $p_1, \ldots, p_n$, which may *crash* at any time during an execution. The *index* of process p_i is i. Processes communicate with each other by invoking *atomic* operations on shared *base objects*. A base object can provide atomic Read/Write operations (henceforth called a *register*), or more powerful atomic Read-Modify-Write operations, such as Fetch&Inc, Swap or Compare&Swap.

A *(high-level) concurrent object*, or *data type*, is, roughly speaking, defined by a state machine consisting of a set of states, a finite set of operations, and a set of transitions between states. The specification does not necessarily have to be *sequential*, namely, (1) a state might have pending operations and (2) state transitions might involve several invocations. The following subsections formalize this notion and the different types of objects.

An *implementation* of a concurrent object T is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which operations on base objects p_i executes in order to return a response when it invokes a high-level operation of T. A process is sequential: it can invoke a new high-level operations only when its previous operation has been responded. Each of these base objects operation invocations is a *step*. Thus, an *execution* of $\mathcal{A}$ is a possibly infinite sequence of steps, namely, executions of base objects operations, plus invocations and responses to high-level operations of the concurrent object T.

An operation in an execution is *complete* if both its invocation and response appear in the execution. An operation is *pending* if only its invocation appears in the execution. A process is *correct* in an execution if it takes infinitely many steps.

An implementation is *wait-free* if every process completes each operation it invokes [23]. An implementation is *non-blocking* if whenever processes take steps and at least one of them does not crash, at least one of them terminates its operation [25]. Thus, a wait-free implementation is non-blocking but not necessarily vice versa.

The *consensus number* of a shared object O is the maximum number of processes that can solve *consensus*, using any number of instances of O in addition to any number of Read/Write registers [23]. Consensus numbers induce the *consensus hierarchy* where objects are classified according their consensus numbers. The simple Read/Write operations stand at the bottom of the hierarchy, with consensus number one; these operations are the least expensive ones in real multicore architectures. At the top of the hierarchy we find operations with infinite consensus number, like Compare&Swap, that provide the maximum possible coordination.

Correctness Conditions. *Linearizability* [25] is the standard notion used to define a correct concurrent implementation of an object defined by a sequential specification. Intuitively, an execution is linearizable if its operations can be totally ordered, while respecting the execution order its non-concurrent operations (see below).

A *sequential specification* of a concurrent object T is a state machine specified through a transition function δ. Given a state q and an invocation $inv(\mathsf{op})$, $\delta(q, inv(\mathsf{op}))$ returns the tuple $(q', res(\mathsf{op}))$ (or a set of tuples if the machine is *non-deterministic*) indicating that the machine moves to state q' and the response to op is $res(\mathsf{op})$. In our specifications, $res(\mathsf{op})$ is written as a tuple $\langle \mathsf{op} : r\rangle$, where r is the output value of the operation. The sequences of invocation-response tuples, $\langle inv(\mathsf{op}) : res(\mathsf{op})\rangle$, produced by the state machine are its *sequential executions.*

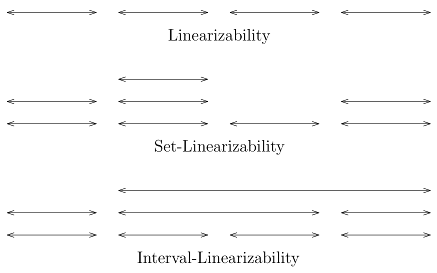

Figure 1 Linearizability requires a total order on the operations, set-linearizability allows several operations to be linearized at the same linearization point, while interval-linearizability allows an operation to be decomposed into several linearization points.

To formalize linearizability we define a partial order $<_\alpha$ on the completed operations of an execution α: $\mathsf{op} <_\alpha \mathsf{op}'$ if and only if $res(\mathsf{op})$ precedes $inv(\mathsf{op}')$ in α. Two operations are *concurrent*, denoted $\mathsf{op}||_\alpha\mathsf{op}'$, if they are incomparable by $<_\alpha$. The execution is *sequential* if $<_\alpha$ is a total order.

Let $\mathcal{A}$ be an implementation of a concurrent object T. An execution α of $\mathcal{A}$ is *linearizable* if there is a sequential execution S of T such that: (1) S contains every completed operation of α and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in α. (2) For every two completed operations op and op' in α, if $\mathsf{op} <_\alpha \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is *linearizable* if each of its executions is linearizable.

To formally specify our relaxed queues and stacks, we use the formalism provided by the set-linearizability and interval-linearizability consistency conditions [14, 33]. Roughly speaking, set-linearizability allows us to linearize several operations in the same point, namely, all these operations are executed concurrently, while interval-linearizability allows operations to be linearized concurrently with several non-concurrent operations. Figure 1 schematizes the differences between the three consistency conditions where each double-end arrow represents an operation execution.

A *set-concurrent specification* of a concurrent object differs from a sequential execution in that δ receives as input the current state q of the machine and a set $Inv = \{inv(\mathsf{op}_1), \ldots, inv(\mathsf{op}_t)\}$ of operation invocations, and $\delta(q, Inv)$ returns (q', Res), where q' is the next state and $Res = \{res(\mathsf{op}_1), \ldots, res(\mathsf{op}_t)\}$ are the responses to the invocations in Inv. Intuitively, all operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$ are performed concurrently and move the machine from state q to q'. The sets Inv and Res are called *concurrency classes.* Observe that a set-concurrent specification in which all concurrency classes have a single element corresponds to a sequential specification.

Let $\mathcal{A}$ be an implementation of a concurrent object T. An execution α of $\mathcal{A}$ is *set-linearizable* if there is a set-concurrent execution S of T such that: (1) S contains every completed operation of α and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in α. (2) For every two completed operations op and op' in α, if $\mathsf{op} <_\alpha \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is *set-linearizable* if each of its executions is set-linearizable.

In an *interval-concurrent specification*, some operations might be pending in a given state q, namely, the state records that there is an operation of a process without response. We now have that in $(q', Res) = \delta(q, Inv)$, some of the operations that are pending in q might still be pending in q' and operations invoked in Inv may be pending in q', therefore Res contains the responses to the operations that are completed when moving from q to q'.

Let $\mathcal{A}$ be an implementation of a concurrent object T. An execution α of $\mathcal{A}$ is *interval-linearizable* if there is an interval-concurrent execution S of T such that: (1) S contains every completed operation of α and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in α. (2) For every two completed operations op and op' in α, if $\mathsf{op} <_\alpha \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is *interval-linearizable* if each of its executions is interval-linearizable.

3 Set-Concurrent Stacks with Multiplicity

By the *universality* of consensus [23], we know that, for every sequential object there is a linearizable wait-free implementation of it, for any number of processes, using Read/Write registers and base objects with consensus number ∞, e.g. Compare&Swap [24, 36, 41]. However, the resulting implementation might not be efficient because first, as it is universal, the construction does not exploit the semantics of the particular object, and Compare&Swap may be an expensive base operation. Moreover, such an approach would prevent us from investigating the power and the limit of the Read/Write model (as it was done for Snapshot object for which there are several linearizable wait-free Read/Write efficient implementations, e.g. [1, 8, 26]) and find accordingly meaningful Read/Write-based specifications of relaxed sequential specifications with efficient implementations.

A Wait-free Linearizable Stack from Consensus Number Two. Afek, Gafni and Morisson proposed in [2] a simple linearizable wait-free stack implementation for $n \geq 2$ processes, using Fetch&Inc and Test&Set base objects, whose consensus number is 2. Figure 2 contains a slight variant of this algorithm that uses Swap and *readable* Fetch&Inc objects, both with consensus number 2 (the authors explain in [2] how to replace Test&Set with Swap).

A Push operation reserves a slot in $Item$ by atomically reading and incrementing Top (Line 01) and then places its item in the corresponding position (Line 02). A Pop operation simply reads the Top of the stack (Line 04) and scans down $Items$ from that position (Line 05), trying to obtain an item with the help of a Swap operation (Lines 06 and 07); if the operation cannot get a item (a non-$\perp$ value), it returns empty (Line 09). In what follows, we call this implementation Seq-Stack. It is worth mentioning that, although Seq-Stack has a simple structure, its linearizability proof is far from trivial, the difficult part being proving that items are taken in LIFO order.

In a formal sense, Seq-Stack is the best we can do, from the perspective of the consensus hierarchy: if there were a wait-free (or non-blocking) linearizable implementation based only on Read/Write registers, we could solve consensus among two processes in the standard way, by popping a value from the stack initialized to a single item containing a predefined value `winner`;

```
Shared Variables:
  $Top$ : Fetch&Inc object initialized to 1
  $Items[1,\ldots]$ : Swap objects initialized to $\perp$

Operation Push($x_i$) is
(01)  $top_i \leftarrow Top$.Fetch&Inc()
(02)  $Items[top_i]$.Write($x_i$)
(03)  return true
end Push

Operation Pop() is
(04)  $top_i \leftarrow Top$.Read() $- 1$
(05)  for $r_i \leftarrow top_i$ down to 1 do
(06)      $x_i \leftarrow Items[r_i]$.Swap($\perp$)
(07)      if $x_i \neq \perp$ then return $x_i$ end if
(08)  end for
(09)  return $\epsilon$
end Pop
```

Figure 2 Stack implementation Seq-Stack of Afek, Gafni and Morisson [2] (code for process p_i).

this is a contradiction as consensus cannot be solved from Read/Write registers [24, 36, 41]. Therefore, there is no *exact* wait-free linearizable stack implementation from Read/Write registers only. However, we could search for *approximate* solutions. Below, we show a formal definition of the notion of a *relaxed* set-concurrent stack and prove that it can be wait-free implemented from Read/Write registers. Informally, our solution consists in implementing relaxed versions of Fetch&Inc and Swap with Read/Write registers, and plug these implementations in Seq-Stack.

A Set-linearizable Read/Write **Stack with Multiplicity.** Roughly speaking, our relaxed stack allows concurrent Pop operations to obtain the same item, but all items are returned in LIFO order, and no pushed item is lost. Formally, our set-concurrent stack is specified as follows:

▶ **Definition 1** (Set-Concurrent Stack with Multiplicity). *The universe of items that can be pushed is $\mathbf{N} = \{1, 2, \ldots\}$, and the set of states Q is the infinite set of strings $\mathbf{N}^*$. The initial state is the empty string, denoted ϵ. In state q, the first element in q represents the top of the stack, which might be empty if q is the empty string. The transitions are the following:*

1. *For $q \in Q$, $\delta(q, \mathsf{Push}(x)) = (x \cdot q, \langle \mathsf{Push}(x) : \mathsf{true} \rangle)$.*
2. *For $q \in Q$, $1 \leq t \leq n$ and $x \in \mathbf{N}$: $\delta(x \cdot q, \{\mathsf{Pop}_1(), \ldots, \mathsf{Pop}_t()\}) = (q, \{\langle \mathsf{Pop}_1() : x \rangle, \ldots, \langle \mathsf{Pop}_t() : x \rangle\})$.*
3. *$\delta(\epsilon, \mathsf{Pop}()) = (\epsilon, \langle \mathsf{Pop}() : \epsilon \rangle)$.*

▶ Remark 2. Every execution of the set-concurrent stack with all its concurrency classes containing a single operation boils down to an execution of a sequential stack.

The following lemma shows that any algorithm implementing the set-concurrent stack keeps the behavior of a sequential stack in several cases. In fact, the only reason the implementation does not provide linearizability is due only to the Pop operations that are concurrent.

▶ **Lemma 3.** *Let A be any set-linearizable implementation of the set-concurrent stack with multiplicity. Then,*

1. *All sequential executions of A are executions of the sequential stack.*
2. *All executions with no concurrent* Pop *operations are linearizable with respect to the sequential stack.*

3. *All executions with* Pop *operations returning distinct values are linearizable with respect to the sequential stack.*
4. *If* Pop *operations return the same value in an execution, then they are concurrent.*

The algorithm in Figure 3 is a set-linearizable Read/Write wait-free implementation of the stack with multiplicity, which we call Set-Conc-Stack. This implementation is a modification of Seq-Stack. The Fetch&Inc operation in Line 01 in Seq-Stack is replaced by a Read and Increment operations of a Read/Write wait-free linearizable Counter, in Lines 01 and 02 in Set-Conc-Stack. This causes a problem as two Push operations can set the same value in their top_i local variables. This problem is resolved with the help of a two-dimensional array $Items$ in Line 03, which guarantees that no pushed item is lost: each row of $Items$ now has n entries, each of them associated with one and only process. Similarly, the Swap operation in Line 06 in Seq-Stack is replaced by Read and Write operations in Lines 08 and 10 in Set-Conc-Stack, together with the test in Line 09 which ensures that a Pop operation modifies an entry in $Items$ only if an item has been written in it. Thus, it is now possible that two distinct Pop operations get the same non-$\perp$ value, which is fine because this can only happen if the operations are concurrent. Object Top in Set-Conc-Stack can be any of the known Read/Write wait-free linearizable Counter implementations[1].

```
Shared Variables:
   $Top$ : Read/Write Counter object initialized to 1
   $Items[1,\ldots][1,\ldots,n]$ : Read/Write registers init. to $\perp$

Operation Push($x$) is
(01)  $top_i \leftarrow Top$.Read()
(02)  $Top$.Increment()
(03)  $Items[top_i, i]$.Write($x$)
(04)  return  true
end Push

Operation Pop() is
(05)  $top_i \leftarrow Top$.Read() $- 1$
(06)  for $r_i \leftarrow top_i$ down to 1 do
(07)      for $s_i \leftarrow n$ down to 1 do
(08)          $x_i \leftarrow Items[r_i][s_i]$.Read()
(09)          if $x_i \neq \perp$ then
(10)              $Items[r_i][s_i]$.Write($\perp$)
(11)              return $x_i$
(12)          end if
(13)      end for
(14)  end for
(15)  return $\epsilon$
end Pop
```

Figure 3 Read/Write wait-free set-concurrent stack Set-Conc-Stack with multiplicity (code for process p_i).

▶ **Theorem 4.** *The algorithm* Set-Conc-Stack *(Figure* 3*) is a* Read/Write *wait-free set-linearizable implementation of the stack with multiplicity.*

Proof sketch. The set-linearizability proof is a "reduction" that proceeds as follows. For any execution E, we modify it and remove some of its operations to obtain another execution G of the algorithm. Then, from G, we obtain an execution H of Seq-Stack, and show that we can obtain a set-linearization SetLin(G) of G from any linearization Lin(H) of H. Finally, we add to SetLin(G) the operations of E that were removed to obtain a set-linearization

[1] To the best of our knowledge, the best implementation is in [7] with polylogarithmic step complexity, on the number of processes, provided that the number of increments is polynomial.

$\mathsf{SetLin}(E)$ of E. To obtain the execution G, we first obtain intermediate executions F and then F', from which we derive G. For any value $y \neq \epsilon$ that is returned by at least two concurrent Pop operations in E, we remove all these operations (invocations, responses and steps) except for the first one that executes Line 10, i.e., the first among these operations that marks y as taken in $Items$. Let F be the resulting execution of Set-Conc-Stack. Since there are no two Pop operations in F popping the same item $y \neq \epsilon$, then for every Pop operation we can safely move backward each of its steps in Line 10 next to its previous step in Line 08 (which corresponds to the same iteration of the for loop in Line 07). Thus, for every Pop operation, Lines 08 to 10 correspond to a Swap operation. Let F' denote the resulting equivalent execution.

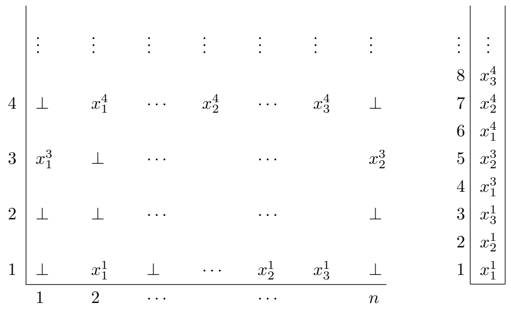

Figure 4 An example of the codification of the one-dimensional array $Items$ of Seq-Stack in the two-dimensional array $Items$ in Set-Conc-Stack. The untouched entries are represented with $\perp$.

We now permute the order of some steps in F' to obtain G. For each integer $b \geq 0$, let $t(b) \in [0, \ldots, n]$ be the number of Push operations in F' that store their items in row $Items[b]$. Namely, each of these operations obtains b in its Read steps in Line 01. Let $\mathsf{Push}_1^b, \ldots, \mathsf{Push}_{t(b)}^b$ denote all these operations. For each Push_j^b, let x_j^b denote the item the operation pushes, let e_j^b denote its Read step in Line 01, and let ind_j^b be the index of the process that performs operation Push_j^b. Hence, Push_j^b stores its item x_j^b in $Items[b][ind_j^b]$ when performs Line 03. Without loss of generality, let us suppose that $ind_1^b < ind_2^b < \ldots < ind_{t(b)}^b$. Observe that $\mathsf{Push}_1^b, \ldots, \mathsf{Push}_{t(b)}^b$ are concurrent.

Let f^b be the first among the steps $e_1^b, \ldots, e_{t(b)}^b$ that appears in F'. As explained in the full proof, moving forward each e_j^b right after f^b produces another execution equivalent to F'. Thus, we obtain G by moving forward all steps $e_1^b, \ldots, e_{t(b)}^b$ up to the position of f^b, and place them in that order, $e_1^b, \ldots, e_{t(b)}^b$, for every $b \geq 0$. Intuitively, all $\mathsf{Push}_1^b, \ldots, \mathsf{Push}_{t(b)}^b$ concurrently read Top and then concurrently increment it.

The main observation now is that G already corresponds to an execution of Seq-Stack, if we consider the entries in $Items$ in their usual order (first row, then column). We say that $Items[r][s]$ is *touched* in G if there is a Push operation that writes its item in that entry; otherwise, $Items[r][s]$ is *untouched*. Now, for every $b \geq 0$, in G all $\mathsf{Push}_1^b, \ldots, \mathsf{Push}_{t(b)}^b$ execute Line 01 one right after the other, in order $e_1^b, \ldots, e_{t(b)}^b$. Also, the items they push appear in row $Items[b]$ from left to right in order $\mathsf{Push}_1^b, \ldots, \mathsf{Push}_{t(b)}^b$. Thus, we can think of the touched entries in row $Items[b]$ as a column with the left most element at the bottom, and pile all rows of $Items$ with $Items[0]$ at the bottom. Figure 4 depicts an example of the transformation. In this way, each e_j^b corresponds to a Fetch&Inc operation and every Pop operations scans the touched entries of $Items$ in the order Seq-Stack does (note that it does not matter if the operation start scanning in a row of $Items$ with no touched entries, since untouched entries are immaterial). Thus, from G we can obtain an execution H of Seq-Stack.

Any linearization $\mathsf{Lin}(H)$ of H is indeed a set-linearization of F and G with each concurrency class having a single operation. To obtain a set-linearization $\mathsf{SetLin}(E)$ of E, we put every Pop operation of E that is removed to obtain F, in the concurrency class of $\mathsf{Lin}(H)$ with the Pop operation that returns the same item. Therefore, E is set-linearizable. ◀

It is worth observing that indeed it is simple to prove that Set-Conc-Stack is an implementation of the *set-concurrent pool with multiplicity*, namely, Definition 1 without LIFO order (i.e. q is a set instead of a string). The hard part in the previous proof is the LIFO order, which is shown through a reduction to the (nontrivial) linearizability proof of Seq-Stack [2].

A Renaming-based Performance-related Improvement. When the contention on the shared memory accesses is small, a Pop operation in Set-Conc-Stack might perform several "useless" Read operations in Line 08, as it scans all entries of *Items* in every row while trying to get a non-$\perp$ value, and some of these entries might never store an item in the execution (called untouched in the proof of Theorem 4). This issue can be mitigated with the help of an array *Ren* with instances of any Read/Write $f(n)$-adaptive renaming. In $f(n)$-*adaptive renaming* [9], each process starts with its index as input and obtains a unique name in the space $\{1, \ldots, f(p)\}$, where p denotes the number of processes participating in the execution. Several adaptive renaming algorithms have been proposed (see e.g. [13]); a good candidate is the simple $(p^2/2)$-adaptive renaming algorithm of Moir and Anderson with $O(p)$ individual step complexity [31].

Push operations storing their items in the same row $Items[b]$, which has now infinite length, dynamically decide where in the row they store their items, with the help of $Ren[b].\mathsf{Rename}(\cdot)$ before performing Line 02. Additionally, these operations announce the number of operations that store values in row $Items[b]$ by incrementing a counter $NOPS[b]$ before incrementing Top in Line 02. In this way, a Pop operation first reads the value x of $NOPS[r_i]$ before the **for** loop in Line 06, and then scans only that segment of $Items[r_i]$ in the **for** loop in Line 07, namely, $Item[r_i][1, \ldots, f(x)]$.

Note that if the contention is small, say $O(\log^x n)$, every Pop operation scans only the first entries $O(\log^{2x} n)$ of row $Items[b]$ as the processes storing items in that row rename in the space $\{1, \ldots, (\log^{2x} n)/2\}$, using the Moir and Anderson $(p^2/2)$-adaptive renaming algorithm. Finally, observe that n does not to be known in the modified algorithm (as in Seq-Stack).

4 Set-Concurrent Queues with Multiplicity

We now consider the linearizable queue implementation in Figure 5, which uses objects with consensus number two. The idea of the implementation, which we call Seq-Queue, is similar to that of Seq-Stack in the previous section. Differently from Seq-Stack, whose operations are wait-free, Seq-Queue has a wait-free Enqueue and a non-blocking Dequeue.

Seq-Queue is a slight modification of the non-blocking queue implementation of Li [29], which in turn is a variation of the blocking queue implementation of Herlihy and Wing [25]. Each Enqueue operation simply reserves a slot for its item by performing Fetch&Inc to the tail of the queue, Line 01, and then stores it in *Items*, Line 02. A Dequeue operation repeatedly tries to obtain an item scanning *Items* from position 1 to the tail of the queue (from its perspective), Line 07; every time it sees an item has been stored in an entry of *Items*, Lines 09 and 10, it tries to obtain the item by atomically replacing it with $\top$, which signals that the item stored in that entry has been taken, Line 11. While scanning, the operation records the number of items that has been taken (from its perspective), Line 13, and if this number is equal to the number of items that were taken in the previous scan, it declares the queue is empty, Line 16. Despite its simplicity, Seq-Queue's linearizability proof is far from trivial.

```
Shared Variables:
  $Tail$ : Fetch&Inc object initialized to 1
  $Items[1,...]$ : Swap objects initialized to $\perp$

Operation Enqueue($x_i$) is
(01)  $tail_i \leftarrow Tail$.Fetch&Inc()
(02)  $Items[tail_i]$.Write($x_i$)
(03)  return true
end Enqueue

Operation Dequeue() is
(04)  $taken'_i \leftarrow 0$
(05)  while true do
(06)      $taken_i \leftarrow 0$
(07)      $tail_i \leftarrow Tail$.Read() $- 1$
(08)      for $r_i \leftarrow 1$ up to $tail_i$ do
(09)          $x_i \leftarrow Items[r_i]$.Read()
(10)          if $x_i \neq \perp$ then
(11)              $x_i \leftarrow Items[r_i]$.Swap($\top$)
(12)              if $x_i \neq \top$ then return $x_i$ end if
(13)              $taken_i \leftarrow taken_i + 1$
(14)          end if
(15)      end for
(16)      if $taken_i = taken'_i$ then return $\epsilon$
(17)      $taken'_i \leftarrow taken_i$
(18)  end while
end Dequeue
```

Figure 5 Non-blocking linearizable queue Seq-Queue from base objects with consensus number 2 (code for p_i).

Similarly to the case of the stack, Seq-Queue is optimal from the perspective of the consensus hierarchy as there is no non-blocking linearizable queue implementation from Read/Write operations only. However, as we will show below, we can obtain a Read/Write non-blocking implementation of a set-concurrent queue with multiplicity.

▶ **Definition 5** (Set-Concurrent Queue with Multiplicity). *The universe of items that can be enqueued is* $\mathbf{N} = \{1, 2, \ldots\}$*, and the set of states* Q *is the infinite set of strings* $\mathbf{N}^*$*. The initial state is the empty string, denoted* ϵ*. In state* q*, the first element in* q *represents the head of the queue, which might be empty if* q *is the empty string. The transitions are the following:*

1. *For* $q \in Q$, $\delta(q, \text{Enqueue}(x)) = (q \cdot x, \langle \text{Enqueue}(x) : \text{true} \rangle)$.
2. *For* $q \in Q$, $1 \leq t \leq n$, $x \in \mathbf{N}$: $\delta(x \cdot q, \{\text{Dequeue}_1(), \ldots, \text{Dequeue}_t()\}) = (q, \{\langle \text{Dequeue}_1() : x \rangle, \ldots, \langle \text{Dequeue}_t() : x \rangle\})$.
3. $\delta(\epsilon, \text{Dequeue}()) = (\epsilon, \langle \text{Dequeue}() : \epsilon \rangle)$.

▶ Remark 6. Every execution of the set-concurrent queue with all its concurrency classes containing a single operation is an execution of the sequential queue.

▶ **Lemma 7.** *Let* A *be any set-linearizable implementation of the set-concurrent queue with multiplicity. Then,*

1. *All sequential executions of* A *are executions of the sequential queue.*
2. *All executions with no concurrent* Dequeue *operations are linearizable with respect to the sequential queue.*
3. *All executions with* Dequeue *operations returning distinct values are linearizable with respect to the sequential queue.*
4. *If two* Dequeue *operations return the same value in an execution, then they are concurrent.*

Following a similar approach to that in the previous section, Seq-Queue can be modified to obtain a Read/Write non-blocking implementation of a queue with multiplicity. The algorithm is modified as follows: (1) replace the Fetch&Inc object in Seq-Queue with a Read/Write wait-free Counter, (2) extend *Items* to a matrix to handle collisions, and (3) simulate the Swap operation with a Read followed by a Write. The correctness proof of the modified algorithm is similar to the correctness proof of Set-Conc-Stack. In fact, proving that the algorithm implements the set-concurrent pool with multiplicity is simple, the difficulty comes from the FIFO order requirement of the queue, which is shown through a simulation argument.

▶ **Theorem 8.** *There is a* Read/Write *non-blocking set-linearizable implementation of the queue with multiplicity.*

5 Implications

Avoiding Costly Synchronization Operations/Patterns. It is worth observing that Set-Conc-Stack and Set-Conc-Queue allow us to circumvent the linearization-related impossibility results in [12], where it is shown that every linearizable implementation of a queue or a stack, as well as other concurrent operation executions as encountered for example in work-stealing, must use either expensive Read-Modify-Write operations (e.g. Fetch&Inc and Compare&Swap) or Read-After-Write patterns [12] (i.e. a process writing in a shared variable and then reading another shared variable, may be performing operation on other variables in between).

In the simplest Read/Write Counter implementation we are aware of, the object is represented via a shared array M with an entry per process; process p_i performs Increment by incrementing its entry, $M[i]$, and Read by reading, one by one, the entries of M and returning the sum. Using this simple Counter implementation, we obtain from Set-Conc-Stack a set-concurrent stack implementation with multiplicity, devoided of (1) Read-Modify-Write operations, as only Read/Write operations are used, and (2) Read-After-Write patterns, as in both operations, Push and Pop, a process first reads and then writes. It similarly happens with Set-Conc-Queue.

Work-stealing with multiplicity. Our implementations also provide relaxed *work-stealing* solutions without expensive synchronization operation/patterns. Work-stealing is a popular technique to implement load balancing in a distributed manner, in which each process maintains its own *pool* of tasks and occasionally *steals* tasks from the pool of another process. In more detail, a process can Put and Take tasks in its own pool and Steal tasks from another pool. To improve performance, [30] introduced the notion of *idempotent work-stealing* which allows a task to be taken/stolen at least once instead of exactly once as in previous work. Using this relaxed notion, three different solutions are presented in that paper where the Put and Take operations avoid Read-Modify-Write operations and Read-After-Write patterns; however, the Steal operation still uses costly Compare&Swap operations.

Our set-concurrent queue and stack implementations provide idempotent work-stealing solutions in which no operation uses Read-Modify-Write operations and Read-After-Write patterns. Moreover, in our solutions both Take and Steal are implemented by Pop (or Dequeue), hence any process can invoke those operations, allowing more concurrency. If we insist that Take and Steal can be invoked only by the owner, *Items* can be a 1-dimensional array. Additionally, differently from [30], whose approach is practical, our queues and stacks with multiplicity are formally defined, with a clear and simple semantics.

Out-of-order queues and stacks with multiplicity: The notion of a k*-FIFO queue* is introduced in [28] (called k-out-of-order queue in [22]), in which items can be dequeued out of FIFO order up to an integer $k \geq 0$. More precisely, dequeueing the oldest item may require up to $k+1$ dequeue operations, which may return elements not younger than the $k+1$ oldest elements in the queue, or nothing even if the queue is not empty. [28] presents also a simple way to implement a k-FIFO queue, through p independent FIFO queue linearizable implementations. When a process wants to perform an operation, it first uses a *load balancer* to pick one of the p queues and then performs its operation. The value of k depends on p and the load balancer. Examples of load balancers are round-robin load balancing, which requires the use of Read-Modify-Write operations, and randomized load balancing, which does not require coordination but can be computational locally expensive. As explained in [28], the notion of a k*-FIFO stack* can be defined and implemented similarly.

We can relax the k-FIFO queues and stacks to include multiplicity, namely, an item can be taken by several concurrent operations. Using p instances of our set-concurrent stack or queue Read/Write implementations, we can easily obtain set-concurrent implementations of k-FIFO queues and stacks with multiplicity, where the use of Read-Modify-Write operations or Read-After-Write patterns are in the load balancer.

6 Interval-Concurrent Queues with Weak-Emptiness Check

A natural question is if in Section 4 we could start with a wait-free linearizable queue implementation instead of Seq-Queue, which is only non-blocking, and hence derive a wait-free set-linearizable queue implementation with multiplicity. It turns out that it is an open question if there is a wait-free linearizable queue implementation from objects with consensus number two. (Concretely, such an algorithm would show that the queue belongs to the Common2 family of operations [4].) This question has been open for more than two decades [4] and there have been several papers proposing wait-free implementations of restricted queues [10, 18, 29, 16, 17], e.g., limiting the number of processes that can perform a type of operations.

```
Shared Variables:
  $Tail$ : Fetch&Inc object initialized to 1
  $Items[1,\ldots]$ : Swap objects initialized to $\perp$

Operation Enqueue($x_i$) is
(01)  $tail_i \leftarrow Tail$.Fetch&Inc()
(02)  $Items[tail_i]$.Write($x_i$)
(03)  return true
end Enqueue

Operation Dequeue() is
(04)  $tail_i \leftarrow Tail$.Read() $- 1$
(05)  for $r_i \leftarrow 1$ up to $tail_i$ do
(06)      $x_i \leftarrow Items[r_i]$.Swap($\perp$)
(07)      if $x_i \neq \perp$ then return $x_i$ end if
(08)  end for
(09)  return $\epsilon$
end Dequeue
```

Figure 6 A non-linearizable queue implementation (code for process p_i).

The Tail-Chasing Problem. One of the main difficulties to solve when trying to design such an implementations using objects with consensus number two is that of reading the current position of the tail. This problem, which we call as *tail-chasing*, can be easily exemplified with the help of the *non-linearizable* queue implementation in Figure 6. The implementation is similar to Seq-Stack with the difference that Dequeue operations scan *Items* in the opposite order, i.e. from the head to the tail.

The problem with this implementation is that once a Dequeue has scanned unsuccessfully *Item* (i.e., the items that were in the queue were taken by "faster" operations), it returns ϵ; however, while the operation was scanning, more items could have been enqueued, and indeed it is not safe to return ϵ as the queue might not be empty. Figure 7 describes an execution of the implementation that cannot be linearized because there is no moment in time during the execution of the Dequeue operation returning ϵ in which the queue is empty. Certainly, this problem can be solved as in Seq-Queue: read the tail and scan again; thus, in order to complete, a Dequeue operation is forced to *chase* the current position of the tail until it is sure there are no new items.

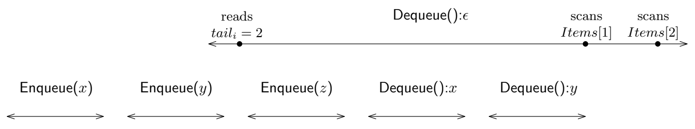

Figure 7 An example of the tail-chasing problem.

Inspired by this problem, below we introduce a relaxed interval-concurrent queue that allows a Dequeue operation to return a weak-empty value, with the meaning that the operation was not able take any of the items that were in the queue when it started but it was concurrent with all the Dequeue operation that took those items, i.e., it has a sort of *certificate* that the items were taken, and the queue might be empty. Then, we show that such a relaxed queue can be wait-free implemented from objects with consensus number two.

A Wait-Free Interval-Concurrent Queue with Weak-Emptiness. Roughly speaking, in our relaxed interval-concurrent queue, the state is a tuple (q, P), where q denotes the state of the queue and P denotes the *pending* Dequeue operations that eventually return *weak-empty*, denoted ϵ^{w}. More precisely, $P[i] \neq \perp$ means that process p_i has a pending Dequeue operation. $P[i]$ is a prefix of q and represents the remaining items that have to be dequeued so that the current Dequeue operation of p_i can return ϵ^{w}. Dequeue operations taking items from the queue, also remove the items from $P[i]$, and the operation of p_i can return ϵ^{w} only if $P[i]$ is ϵ. Intuitively, the semantics of ϵ^{w} is that the queue *could* be empty as all items that were in the queue when the operations started have been taken. So this Dequeue operation virtually occurs after all the items have been dequeued.

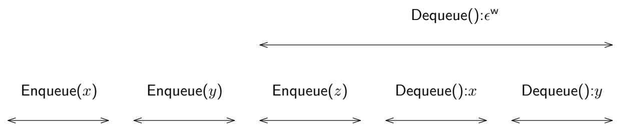

Figure 8 An interval-concurrent execution with a Dequeue operations returning weak-empty.

Figure 8 shows an example of an interval-concurrent execution of our relaxed queue where the Dequeue operation returning ϵ^{w} is allowed to return only when x and y have been dequeued. Observe that this execution is an interval-linearization of the execution obtained from Figure 7 by replacing ϵ with ϵ^{w}.

▶ **Definition 9** (Interval-Concurrent Queue with Weak-Empty)**.** *The universe of items that can be enqueued is* $\mathbf{N} = \{1, 2, \ldots\}$ *and the set of states is* $Q = \mathbf{N}^* \times (\mathbf{N}^* \cup \{\perp\})^n$*, with the initial state being* $(\epsilon, \perp, \ldots, \perp)$*. Below, a subscript denotes the ID of the process invoking an operation. The transitions are the following:*

1. *For* $(q, P) \in Q$*,* $0 \le t, \ell \le n-1$*,* $\delta(q, P, \mathsf{Enqueue}(x), \mathsf{Dequeue}_{i(1)}(), \ldots, \mathsf{Dequeue}_{i(t)}())$ *contains the transition* $(q \cdot x, S, \langle \mathsf{Enqueue}(x) : \mathsf{true}\rangle, \langle \mathsf{Dequeue}_{j(1)}() : \epsilon^{\mathsf{w}}\rangle, \ldots, \langle \mathsf{Dequeue}_{j(\ell)}() : \epsilon^{\mathsf{w}}\rangle)$*, satisfying that*
   a. $i(k) \neq i(k')$*,* $i(k) \neq$ *the id of the process invoking* Enqueue(x)*, and* $j(k) \neq j(k')$*,*
   b. *for each* $i(k)$*,* $P[i(k)] = \perp$*,*
   c. *for each* $j(k)$*, either* $P[j(k)] = \epsilon$*, or* $P[j(k)] = \perp$ *and* $q = \epsilon$ *and* $j(k) = i(k')$ *for some* k'*,*
   d. *for each* $1 \le s \le n$*, if there is a* k *with* $s = j(k)$*, then* $S[s] = \perp$*; otherwise, if there is* k' *with* $s = i(k')$*,* $S[s] = q$*, else* $S[s] = P[s]$*.*
2. *For* $(x \cdot q, P) \in Q$*,* $0 \le t, \ell \le n-1$*,* $\delta(x \cdot q, P, \mathsf{Dequeue}(), \mathsf{Dequeue}_{i(1)}(), \ldots, \mathsf{Dequeue}_{i(t)}())$ *contains the transition* $(q, S, \langle \mathsf{Dequeue}() : x\rangle, \langle \mathsf{Dequeue}_{j(1)}() : \epsilon^{\mathsf{w}}\rangle, \ldots, \langle \mathsf{Dequeue}_{j(\ell)}() : \epsilon^{\mathsf{w}}\rangle)$*, satisfying that*
   a. $i(k) \neq i(k')$*,* $i(k) \neq$ *the id of the process invoking* Dequeue()*, and* $j(k) \neq j(k')$*,*
   b. *for each* $i(k)$*,* $P[i(k)] = \perp$*,*
   c. *for each* $j(k)$*, either* $P[j(k)] = x$*, or* $P[j(k)] = \perp$ *and* $q = \epsilon$ *and* $j(k) = i(k')$ *for some* k'*,*
   d. *for each* $1 \le s \le n$*, if there is a* k *with* $s = j(k)$*, then* $S[s] = \perp$*; otherwise, if there is* k' *with* $s = i(k')$*,* $S[s] = q$*, else* $S[s]$ *is the string obtained by removing the first symbol of* $P[s]$ *(which must be* x*).*
   e. *if* $x \cdot q = \epsilon$ *and* $t, \ell = 0$*, then* $x \in \{\epsilon, \epsilon^{\mathsf{w}}\}$*.*

▶ Remark 10. Every execution of the interval-concurrent queue with no dequeue operation returning ϵ^{w} is an execution of the sequential queue.

▶ **Lemma 11.** *Let* A *be any interval-linearizable implementation of the interval-concurrent queue with weak-empty. Then, (1) all sequential executions of* A *are executions of the sequential queue, and (2) all executions in which no* Dequeue *operation is concurrent with any other operation are linearizable with respect to the sequential queue.*

The algorithm in Figure 9, which we call Int-Conc-Queue, is an interval-linearizable wait-free implementation of a queue with weak-emptiness, which uses base objects with consensus number two. Int-Conc-Queue is a simple modification of Seq-Queue in which an Enqueue operation proceeds as in Seq-Queue, while a Dequeue operation scans *Items* at most two times to obtain an item, in both cases recording the number of taken items. If the two numbers are the same (cf. *double clean* scan), then the operations return ϵ, otherwise it returns ϵ^{w}.

▶ **Theorem 12.** *The algorithm* Int-Conc-Queue *(Figure 9) is a wait-free interval-linearizable implementation of the queue with weak-empty, using objects with consensus number two.*

Interval-Concurrent Queue with Weak-emptiness and Multiplicity. Using the techniques in Sections 3 and 4, we can obtain a Read/Write wait-free implementation of a even more relaxed interval-concurrent queue in which an item can be taken by several dequeue operations, i.e., with multiplicity. In more detail, the interval-concurrent queue with weak-emptiness is modified such that concurrent Dequeue operations can return the same item and are set-linearized in the same concurrency class, as in Definitions 1 and 5.

```
Shared Variables:
  $Tail$ : Fetch&Inc object initialized to 1
  $Items[1, \ldots]$ : Swap objects initialized to $\bot$

Operation Enqueue($x_i$) is
  $tail_i \leftarrow Tail$.Fetch&Inc()
  $Items[tail_i]$.Write($x_i$)
  return true
end Enqueue

Operation Dequeue() is
  for $k \leftarrow 1$ up to 2 do
      $taken_i[k] \leftarrow 0$
      $tail_i \leftarrow Tail$.Read() $- 1$
      for $r_i \leftarrow 1$ up to $tail_i$ do
          $x_i \leftarrow Items[r_i]$.Read()
          if $x_i \neq \bot$ then
              $x_i \leftarrow Items[r_i]$.Swap($\top$)
              if $x_i \neq \top$ then return $x_i$ end if
              $taken_i[k] \leftarrow taken_i[k] + 1$
          end if
      end for
  end for
  if $taken_i[1] = taken_i[2]$ then return $\epsilon$
      else return $\epsilon^{\mathsf{w}}$
  end if
end Dequeue
```

Figure 9 Wait-free interval-concurrent queue from consensus number 2 (code for p_i).

We obtain a Read/Write wait-free interval-concurrent implementation of the queue with weak-emptiness and multiplicity by doing the following: (1) replace the Fetch&Inc object in Int-Conc-Queue with a Read/Write wait-free Counter, (2) extend *Items* to a matrix to handle collisions, and (3) simulate the Swap operation with a Read followed by a Write. Thus, we have:

▶ **Theorem 13.** *There is a* Read/Write *wait-free interval-linearizable implementation of the queue with weak-emptiness and multiplicity.*

7 Final Discussion

Considering classical data structures initially defined for sequential computing, this work has introduced new well-defined relaxations to adapt them to concurrency and investigated algorithms that implement them on top of "as weak as possible" base operations. It has first introduced the notion of set-concurrent queues and stacks with multiplicity, a relaxed version of queues and tasks in which an item can be dequeued more than once by concurrent operations. Non-blocking and wait-free set-linearizable implementations were presented, both based only on the simplest Read/Write operations. These are the first implementations of relaxed queues and stacks using only these operations. The implementations imply algorithms for idempotent work-stealing and out-of-order stacks and queues.

The paper also introduced a relaxed concurrent queue with weak-emptiness check, which allows a dequeue operation to return a "weak-empty certificate" reporting that the queue might be empty. A wait-free interval-linearizable implementation using objects with consensus number two was presented for such a relaxed queue. As there are only non-blocking linearizable (not relaxed) queue implementations using objects with consensus number two, it is an open question if there is such a wait-free implementation. The proposed queue relaxation allowed us to go from non-blocking to wait-freedom using only objects with consensus number two.

This work also can be seen as a work prolonging the results described in [14] where the notion of interval-linearizability was introduced and set-linearizability [33] is studied. It has shown that linearizability, set-linearizability and interval-linearizability constitute a hierarchy of consistency conditions that allow us to formally express the behavior of non-trivial (and still meaningful) relaxed queues and stacks on top of simple base objects such as Read/Write registers. An interesting extension to this work is to explore if the proposed relaxations can lead to practical efficient implementations. Another interesting extension is to explore if set-concurrent or interval-concurrent relaxations of other concurrent data structures would allow implementations to be designed without requiring the stronger computational power provided by atomic Read-Modify-Write operations.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. `doi:10.1145/153724.153741`.

2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Comput.*, 20(4):239–252, 2007. `doi:10.1007/s00446-007-0023-3`.

3 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010. `doi:10.1007/978-3-642-17653-1_29`.

4 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993*, pages 159–170, 1993. `doi:10.1145/164051.164071`.

5 Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 133–142, 2018. `doi:10.1145/3210377.3210411`.

6 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: a scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 11–20, 2015. `doi:10.1145/2688500.2688523`.

7 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. `doi:10.1145/2108242.2108244`.

8 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 62(1):3:1–3:22, 2015. `doi:10.1145/2732263`.

9 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990. `doi:10.1145/79147.79158`.

10 Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. *J. Parallel Distributed Comput.*, 121:1–14, 2018. `doi:10.1016/j.jpdc.2018.06.004`.

11 Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4):24:1–24:33, 2009. `doi:10.1145/1538902.1538908`.

12 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498, 2011. `doi:10.1145/1926385.1926442`.

13 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, 2011. `doi:10.1016/j.cosrev.2011.04.001`.

14 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

15 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. What can be done with consensus number one: Relaxed queues and stacks. *CoRR*, abs/2005.05427, 2020. arXiv:2005.05427.

16 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

17 Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 137–151, 2005. doi:10.1007/11561927_12.

18 David Eisenstat. A two-enqueuer queue. *CoRR*, abs/0805.0444, 2008. arXiv:0805.0444.

19 Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012. doi:10.1137/08072646X.

20 Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016. doi:10.4230/LIPIcs.CONCUR.2016.6.

21 Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013. doi:10.1145/2482767.2482789.

22 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 317–328, 2013. doi:10.1145/2429069.2429109.

23 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

24 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

25 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

26 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. *J. Parallel Distributed Comput.*, 72(1):1–12, 2012. doi:10.1016/j.jpdc.2011.08.005.

27 Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *Parallel Computing Technologies - 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30 - October 4, 2013. Proceedings*, pages 208–223, 2013. doi:10.1007/978-3-642-39958-9_18.

28 Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Algorithms and Architectures for Parallel Processing - 12th International Conference, ICA3PP 2012, Fukuoka, Japan, September 4-7, 2012, Proceedings, Part I*, pages 273–287, 2012. doi:10.1007/978-3-642-33078-0_20.

29 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001. URL: http://hdl.handle.net/1807/16583.

30 Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 45–54, 2009. doi:10.1145/1504176.1504186.

31 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995. doi:10.1016/0167-6423(95)00009-H.

32 Mark Moir and Nir Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.ch47.

33 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396, 1994. doi:10.1145/197917.198176.

34 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013. doi:10.1145/2517349.2522739.

35 Hannes Payer, Harald Röck, Christoph M. Kirsch, and Ana Sokolova. Scalability versus semantics of concurrent FIFO queues. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 331–332, 2011. doi:10.1145/1993806.1993869.

36 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. doi:10.1007/978-3-642-32027-9.

37 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 80–82, 2015. doi:10.1145/2755573.2755616.

38 Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011. doi:10.1145/1897852.1897873.

39 Nir Shavit and Gadi Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. *Distributed Comput.*, 29(5):395–407, 2016. doi:10.1007/s00446-016-0272-0.

40 Edward Talmage and Jennifer L. Welch. Relaxed data types as consistency conditions. *Algorithms*, 11(5):61, 2018. doi:10.3390/a11050061.

41 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Prentice Hall, 2006.

42 Tingzhe Zhou, Maged M. Michael, and Michael F. Spear. A practical, scalable, relaxed priority queue. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*, pages 57:1–57:10, 2019. doi:10.1145/3337821.3337911.