The Computability of Relaxed Data Structures: Queues and Stacks as Examples*

Nir Shavit† Gadi Taubenfeld‡

March 29, 2016

Abstract

Most concurrent data structures being designed today are versions of known sequential data structures. However, in various cases it makes sense to relax the semantics of traditional concurrent data structures in order to get simpler and possibly more efficient and scalable implementations. For example, when solving the classical producer-consumer problem by implementing a concurrent queue, it might be enough to allow the *dequeue* operation (by a consumer) to return and remove one of the two oldest values in the queue, and not necessarily the oldest one. We define infinitely many possible relaxations of several traditional data structures and objects: queues, stacks, multisets and registers, and examine their relative computational power.

Keywords: Relaxed data structure, consensus number, synchronization, wait-freedom, queue, stack, multiset, k-register.

1 Introduction

1.1 Motivation

Early in our computer science education, we learn how to implement sequential data structures. In the context of sequential data structures, implementing a queue in which it is fine for a dequeue operation to return one of the two oldest items in the queue, instead of always returning the oldest item, does not help in making the problem of efficiently implementing a queue easier to solve. Maybe for that reason, we sometimes tend to overlook the fact that in the context of concurrent programming, such relaxations might help a lot.

Assume that you need to solve the classical producer-consumer synchronization problem by implementing a concurrent queue. In some cases, it might be fine to allow the consumer to return and remove one of the two oldest items in the queue, and not necessarily the oldest one as is usually required. More generally, in some cases it makes senses to relax the semantics of traditional concurrent data structures in order to get more efficient and scalable concurrent implementations.

*A preliminary version of the results presented in this paper appeared in: (1) the Proceedings of the 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015) [28], and (2) the Proceedings of the 14th International Conf. on Distributed Computing and Networking (ICDCN 2013) [31].

†MIT and Tel-Aviv University. shanir@csail.mit.edu

‡The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel. tgadi@idc.ac.il

There is a trade-off between synchronization and the ability of an implementation to scale performance with the number of processors. Amdahl's law, implies that even a small fraction of inherently sequential code limits scaling. Using semantically weaker data structures may help in reducing the synchronization requirements and hence improves scalability for many-core systems. As a result, there is a recent trend towards implementing semantically weaker data structures for achieving better performance and scalability [27].

Important research has already been done on implementing semantically weaker data structure (see for example, [2, 3, 8, 16, 27, 29]). While these implementations address complexity issues, less research has been done on the computability of relaxed data structures. In this paper we investigate the computability of (wait-free) relaxed data structures, by considering infinitely many possible relaxations of several traditional data structures and objects: queues, stacks, multisets (i.e., bags) and registers, and examine their relative computational power. Our results demonstrate, for example, that for a concurrent queue small changes in its semantics dramatically effects its computational power, and that similar results do not apply for a concurrent stack.

1.2 Data structures with relaxed specifications

We will assume that processes can try to access a shared object at the same time, however, although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in their "real-time" order. This type of correctness requirement for shared objects is called *linearizability* [14].

A concurrent queue is a linearizable data structure that supports *enqueue*, *dequeue* and *peek* operations, by several processes, with the usual queue semantics.[1] Below we generalize this traditional notion of a concurrent queue.

A concurrent queue w.r.t. the numbers a, b and c, denoted $queue[a, b, c]$, is a linearizable data structure that supports the $enq.a(v)$, $deq.b()$ and $peek.c()$ operations, by several processes, with the following semantics: The $enq.a(v)$ operation inserts the value v at one of the a positions at the end of the queue;[2] the $deq.b()$ operation returns and removes one of the values at the b positions at the front of the queue; the $peek.c()$ operation returns one of the values at the c positions at the front of the queue without removing it. If the queue is empty the $deq.b()$ and the $peek.c()$ operations return a special symbol. We emphasize that the queue $queue[a, b, c]$, is implemented w.r.t. some fixed numbers a, b and c; these number are defined a priori and are *not* parameters that are passed at run time.

When defining the queue $queue[a, b, c]$, the numbers a, b and c can take the values of any positive integer, and the two special values 0 and $*$. When a, b or c equals 0, it means that the corresponding operation is not supported; when it equals $*$, it means that the corresponding operation can insert, remove or return (depending on the type of operation) a value at an arbitrary position (i.e., a position chosen by an adversary).

Thus, $queue[1, 1, 1]$ is the traditional FIFO queue (which is sometimes called augmented queue), where the values are dequeued in the order in which they were enqueued, and where the *peek* operation reads the oldest value in the queue without removing it; $queue[1, 1, 0]$ is a queue which supports the standard *enqueue* and *dequeue* operations but does not support a *peek* operation; $queue[1, 1, *]$

[1] The enqueue operation inserts a value to the queue and the dequeue operation returns and removes the oldest value in the queue. That is, the values are dequeued in the order in which they were enqueued. The *peek* operation reads the oldest value in the queue without removing it. If the queue is empty the dequeue and the peek operations return a special symbol.

[2] A position of an item in a queue or in a stack is simply the number of items which precede it plus one.

is a queue where the *peek* operation returns an arbitrary value that is currently in the queue; finally $queue[*, *, 0]$ is exactly a linearizable *multiset* object that supports *insert* and *remove* operations, by several processes, with the usual multiset semantics.

Relaxed versions of other traditional data structures are defined similarly. A relaxed concurrent stack, denoted $stack[a, b, c]$, is a linearizable data structure that supports the $push.a(v)$, $pop.b()$ and $top.c()$ operations, by several processes, with the obvious semantics. The object $stack[*, *, 0]$ is equivalent to the object $queue[*, *, 0]$ and corresponds to a *multiset* object. The object $stack[1, 0, 1]$ is exactly an *atomic read/write register*, where the push and top operations correspond to the write and read operations, respectively. For $k \geq 1$, the object $stack[1, 0, k]$ is exactly a k-atomic register as defined in Section 8.

1.3 Consensus numbers

A process executes correctly its algorithm until it possibly crashes. After it has crashed it executes no more steps. A process that crashes is said to be *faulty*, otherwise it is *non-faulty* (i.e., *correct*).

The (binary) consensus problem is to design an algorithm in which all non-faulty processes reach a common decision based on their initial opinions. The problem is defined as follows: There are n processes $p_1, p_2, \ldots, p_n$. Each process p_i has an input value $x_i \in \{0, 1\}$. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) each non-faulty process eventually decides on v, and (2) $v \in \{x_1, x_2, \ldots, x_n\}$. In particular, if all input values are the same, then that value must be the decision value.

The notion of a consensus number is central to our investigation and is formally defined below. A *wait-free* implementation of an object guarantees that any process can complete any operation in a finite number of steps, regardless of the speed of the other processes. A *register* is an object that supports read and write operations. A register can be atomic or non-atomic. With an *atomic* register, it is assumed that operations on the register (i.e, on the same memory location) occur in some definite order. That is, reading or writing an atomic register is an indivisible action. When reading or writing a non-atomic register (also called *safe* register), a process may be reading a register while another is writing into it, and in that event, the value returned to the reader is arbitrary.

The *consensus number* of an object of type o, denoted $CN(o)$, is the largest n for which it is possible to solve consensus for n processes in a wait-free manner using any number of objects of type o and any number of atomic registers. If no largest n exists, the consensus number of o is infinite (denoted ∞) [11]. Classifying objects by their consensus numbers is a powerful technique for understanding the relative computational power of shared objects.

The *consensus hierarchy* is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i. It is known that, in the consensus hierarchy, for every positive i, in a system with i processes: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level [11].

1.4 Contributions

New definitions. The definitions of concurrent queues and stacks with relaxed specifications together with the following technical results provide a deeper understanding of the computability issues which are involved in the development of relaxed data structures.

Relaxing the enqueue operation. First we show that, while $CN(queue[1,1,1]) = \infty$, the consensus number drops to *two* when the *enqueue* operation is allowed to insert an item at any position at random, regardless whether the *peek* and *dequeue* operations are relaxed or not. That is,

$$CN(queue[*,1,1]) = 2. \tag{R1}$$

It follows from R1 and the known result that $CN(queue[*,*,0]) = 2$ (i.e., that the consensus number of a *multiset* object is 2), that: for every $b \in Z^+ \cup \{*\}$, $c \in Z^+ \cup \{0,*\} : CN(queue[*,b,c]) = 2$. ($Z^+$ is the set of all positive integers.) Next, we show that the consensus number of all the queues in which the *peek* operation is not relaxed (i.e., peek always returns the element at the front of the queue) is *infinity*, even when the *enqueue* operation is allowed to insert an item at any one of the last k positions for any fixed k. That is,

$$\text{For every } a \in Z^+ : CN(queue[a,0,1]) = \infty \tag{R2}$$

In contrast with R2, the consensus numbers of *all* possible relaxations of a concurrent *stack* are at most 2. In particular, $CN(stack[1,1,1]) = 2$ and $CN(stack[1,0,1]) = 1$ [7, 11, 21] (as already mentioned, the object $stack[1,0,1]$ is exactly an atomic read/write register).

Relaxing the peek operation. Next, we show that the consensus number of all the queues in which the *peek* operation is relaxed (i.e., peek is not required to always return the oldest value in the queue), is exactly two, regardless of how far the *enqueue* and *dequeue* operations are relaxed, as long as these operations are supported. That is,

$$CN(queue[1,1,2]) = 2. \tag{R3}$$

It follows from R3 and the known result that the consensus number of a *multiset* object is 2 [15], that: for every $a \in Z^+ \cup \{*\}$, $b \in Z^+ \cup \{*\}$, $c \neq 1 : CN(queue[a,b,c]) = 2$.

Not supporting the dequeue operation. The situation changes dramatically when dequeue is not supported. The consensus number of all the queues where the dequeue operation is not supported *and* the peek operation is slightly relaxed, is just 1. That is,

$$CN(queue[1,0,2]) = 1. \tag{R4}$$

Thus, while $CN(queue[1,0,1]) = \infty$ and $CN(queue[1,1,2]) = 2$, by removing the dequeue operation from the object $queue[1,1,2]$, we get an object with consensus number one. It follows from R4 that: for every $a \in Z^+ \cup \{0,*\} : CN(queue[a,0,2]) = 1$.

Atomic registers vs. relaxed queues. It is known that $CN(atomic\ register) = 1$ [21]. It is easy to see that a $queue[*,0,2]$ has a trivial wait-free implementation from a single atomic register. While, for every $a \in Z^+$, atomic registers and $queue[a,0,2]$ both have consensus number 1, we observe that,

$$\text{A } queue[a,0,c] \text{ has no wait-free implementation from atomic registers, for every two positive integers } a \text{ and } c. \tag{R5}$$

All the above results hold for both an initialized queue and an uninitialized queue. These cases differ, for example, when the enqueue operation is not supported. For an initialized queue, $CN(queue[0,1,0]) = 2$ [11], while for an uninitialized queue, it is obvious that $CN(queue[0,1,1]) = 1$.

Relaxed registers. It is common to assume that operations on the same memory location are atomic – they occur in some definite order. However, this assumption can be relaxed allowing the possibility of concurrent operation on the same memory location. In [20], three classes of shared registers are defined, which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. We consider relaxations of these notions, called k-safe, k-regular and k-atomic, and prove that,

> It is possible to wait-free implement multi-writer multi-reader multi-valued 1-atomic registers (the strongest type) using single-writer single-reader k-safe bits (the weakest type), for every $k \geq 1$. (R6)

Thus, for every $k \geq 1$, k-safe registers and 1-atomic registers have the same computational power.

2 Preliminaries

2.1 Model of computation

Our model of computation consists of an asynchronous collection of $n \geq 2$ processes that communicate via shared objects. We use P to denote the set of all processes. Asynchrony means that there is no assumption on the relative speeds of the processes. Processes may fail by crashing, which means that a failed process stops taking steps forever. In an asynchronous system there is no way to distinguish between a faulty and a very slow process.

An *event* corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but do not return a value. We use the notation e_p to denote an instance of an arbitrary event at a process p.

A *run* is a pair $(f,\ R)$ where f is a function that assigns initial states (values) to the objects and R is a finite or infinite sequence of events. An implementation of an object from a set of other objects, consists of a non-empty set C of runs, a set P of processes, and a set of shared objects O. For any event e_p at a process p in any run in C, the object accessed in e_p must be in O. Let $x = (f,\ R)$ and $x' = (f',\ R')$ be runs. Run x' is a *prefix* of x (and x is an *extension* of x'), denoted $x' \leq x$, if R' is a prefix of R and $f = f'$. When $x' \leq x$, $(x - x')$ denotes the suffix of R obtained by removing R' from R. Let $R;T$ be the sequence obtained by concatenating the finite sequence R and the sequence T. Then $x;T$ is an abbreviation for $(f,\ R;T)$.

Process p is *enabled* at the end of run x if there exists an event e_p such that $x;e_p$ is a run. For simplicity, whenever we say that p is enabled at x we mean that p is enabled at *the end* of x. Also, we write xp to denote either $x;e_p$ when p is enabled in x, or x when p is not enabled in x. Register r is a *local* register of p if only p can access r. For any sequence R, let R_p be the subsequence of R containing all events in R which involve p. Runs $(f,\ R)$ and $(f',\ R')$ are *indistinguishable* for p, denoted by $(f,\ R)[p](f',\ R')$, iff $R_p = R'_p$ and $f(r) = f'(r)$ for every local register r of p.

The runs of an asynchronous implementation of an object must satisfy several properties. For example, if a *write* event which involves p is enabled at run x, then the same event is enabled at any

finite run that is indistinguishable to p from x. In the following proofs, we will implicitly make use of few such straightforward properties.

2.2 Three simple observations

The following lemmas are easy consequences of the above properties and definitions.

Lemma 2.1 *Let w, x and y be runs of an algorithm and p be a process such that (1) $w \leq x$ and $w[p]y$, and (2) the states of all the objects (local and shared) that p can access are the same in w and y, and $(x - w)$ contains only events of p. Then, $z = y;(x - w)$ is a run of the algorithm and $x[p]z$.*

Proof: By induction on the length of $(x - w)$. ∎

Next, we state two simple lemmas regarding relaxed queues. The first states that in any component, going from $a \in Z^+$ to $a + 1$ or to $*$ does not increase the power of the object since it just gives the adversary more choices of what to return. The second lemma states that going from $a \in Z^+ \cup \{0, *\}$ to 0 in any component does not increase the power of the object, since it just eliminates a possible operation.

Lemma 2.2 *For every $a_1, b_1, c_1, a_2, b_2, c_2$ in $Z^+ \cup \{0, *\}$,*
if $((a_2 = * \wedge a_1 \neq 0) \vee 0 < a_1 \leq a_2 \vee a_2 = 0) \wedge ((b_2 = * \wedge b_1 \neq 0) \vee 0 < b_1 \leq b_2 \vee b_2 = 0) \wedge ((c_2 = * \wedge c_1 \neq 0) \vee 0 < c_1 \leq c_2 \vee c_2 = 0)$
then $CN(queue[a_1, b_1, c_1]) \geq CN(queue[a_2, b_2, c_2])$.

Proof: The proof of the lemma follows immediately from the definitions. ∎

Lemma 2.3 *For every a, b, c in $Z^+ \cup \{0, *\}$,*

1. *$CN(queue[0, b, c]) \leq CN(queue[a, b, c])$, and*
2. *$CN(queue[a, 0, c]) \leq CN(queue[a, b, c])$, and*
3. *$CN(queue[a, b, 0]) \leq CN(queue[a, b, c])$.*

Proof: The proof of the lemma follows immediately from the definitions. ∎

We notice that the statement of Lemma 2.2 subsumes that of Lemma 2.3, since it covers the case of a_2, b_2 or c_2 being 0.

2.3 Known results

Lemma 2.4 *(a) $CN(queue[*, *, 0]) = 2$, (b) $CN(queue[1, 1, 0]) = 2$, and*
(c) $CN(stack[1, 1, 1]) = 2$.

The proofs that $CN(queue[*,*,0]) = 2$, $CN(queue[1,1,0]) = 2$, and $CN(stack[1,1,0]) = 2$ (with and without initialization) are from [11, 15]. The wait-free consensus algorithm which uses a single queue and registers from [15], is also correct when the queue is replaced with a stack or with a multiset. Proving that $CN(stack[1,1,1]) = 2$, can be establish by modifying the existing proof from [11], that $CN(queue[1,1,0]) = 2$.[3]

3 Basic properties of wait-free consensus algorithms

The first four lemmas below are known and have appeared (using different notations) or follow from known impossibility proofs for wait-free consensus. The definitions below refer to runs of a given consensus algorithm. A (finite) run x is *v-valent* if in all extensions of x where a decision is made, the decision value is v ($v \in \{0,1\}$). A run is *univalent* if it is either 0-valent or 1-valent, otherwise it is *bivalent*. We say that two univalent runs are *compatible* if they have the same valency, that is, either both runs are 0-valent or both are 1-valent. A run is *critical* if: (1) it is bivalent, and (2) any extension of the run is univalent. A run (f, R) is an *empty* run if the length of R is 0 (that is, no process has taken a step yet). Recall that $n \geq 2$.

Lemma 3.1 *In every wait-free consensus algorithm, if two univalent runs are indistinguishable for some process p, and the states of all the objects that p can access are the same at these runs, then these (univalent) runs must be compatible.*

Proof: Let w and y be univalent runs, such that $w[p]y$ and the states of all the objects (local and shared) that p can access are the same at w and y. By the wait-free property, w has an extension x such $x - w$ contains only events of process p, and p has decided in x. Let w be v-valent, for $v \in \{0,1\}$. Then p decide v in x. (The event in which p decides on v, may be implemented by p writing v into a special single-writer output register.) By Lemma 2.1, $z = y;(x - w)$ is a run of the algorithm such that $z[p]x$. Since p decides on v (i.e., p writes v to its output register) in z, z is v-valent. Hence, since $y \leq z$, y must also be v-valent. ∎

Lemma 3.2 *Every wait-free consensus algorithm has a bivalent empty run.*

Proof: We show that a bivalent empty run must exist. Assume to the contrary that every empty run is univalent. The empty run with all 0 inputs must be 0-valent, and similarly the empty run with all 1 inputs must be 1-valent. Thus, by Lemma 3.1, all the empty runs with all but one 0 inputs are 0-valent, and similarly all the empty runs with all but one 1 inputs are 1-valent. By repeatedly applying this argument i times we get that, all the empty runs with all but i 0 inputs are 0-valent, and similarly all the empty runs with all but i 1 inputs are 1-valent. Thus, when i is half the number of processes, we get that there are two empty runs x_0 and x_1 that differ only at the value of a single input, for process p, such that x_0 is 0-valent and x_1 is 1-valent. However, this contradicts Lemma 3.1. Hence, an empty bivalent run exists. ∎

Lemma 3.3 *Every wait-free consensus algorithm has a critical run.*

[3] To our surprise, we could not find any publication in which it is claimed that $CN(stack[1,1,1]) = 2$. Nevertheless, we consider it as a known result.

Proof: Let $Cons$ be an arbitrary wait-free consensus algorithm. By Lemma 3.2, $Cons$ has an empty bivalent run x_0. We begin with x_0 and pursue the following round-robin *bivalence-preserving scheduling* discipline (Recall that P denotes a set of processes, x and y denote runs and yp is an extension of the run y by one event of process p):

```
1  x := x_0; P := ∅; i := 0                                  /* initialization */
2  repeat
3        if x has a bivalent extension yp_i which involves p_i
4        then x := yp_i                                  /* bivalent extension of x */
5        else P := P ∪ {p_i}                        /* no such bivalent extension */
6        i := i + 1(mod n)                                      /* round-robin */
7  until |P| = n.
```

If the above procedure does not terminate, then there is an infinite run with only bivalent finite prefixes. However, the existence of such a run contradicts the definition of a wait-free consensus algorithm. Hence, the procedure will terminate with some critical run x. ∎

Lemma 3.4 *Let x be a critical run of a wait-free consensus algorithm and let p and q be two different processes such that the runs xp and xq are not compatible. Then, in their next events from x, p and q are accessing the same object, and this object is not a register.*

Proof: We consider the following three possible cases, and show that each one of them leads to a contradiction. We will assume that in the last event in xp process p is accessing some object, say o, and in the last event in xq process q is accessing some object, say o'.

Case 1. $o \neq o'$. Since the next events from x of p and q are independent, $xpq[p]xqp$, and the values of all objects are the same in both xpq and xqp. Hence, by Lemma 3.1, xpq and xqp are compatible; since xpq is an extension of xp and xqp is an extension of xq, it must be that xp and xq are also compatible. A contradiction.

Case 2. $o = o'$ is a register and in xp the last event is a *write* event by p to o. Since p *writes* to o in its next operation from x, the value of o must be the same in xp and xqp. (Here we use the fact that the write by p overwrites the possible changes of o made by q.) Hence, $xp[p]xqp$ and the values of all the objects, which are not local to q, are the same in xp and xqp. By Lemma 3.1, xp and xqp are compatible. Since xqp is an extension of xq, it must be that xp and xq are also compatible. A contradiction.

Case 3. $o = o'$ is a register and in xp the last event is a *read* event by p. Thus, $xpq[q]xq$, and the values of all the objects, which are not local to p, are the same in both xpq and xq. Hence by Lemma 3.1, xpq and xq are compatible. Since xpq is an extension of xp, it must be that xp and xq are also compatible. A contradiction.

Thus, it must be the case that $o = o'$ and o is not a register. ∎

Lemma 3.5 *Let x be a critical run of a wait-free consensus algorithm, and assume that the next event of p from x is a relaxed peek event which may return one of the two oldest items in a queue. Let xp^1 (resp. xp^2) denotes an extension of x by a peek event by p that has returned the oldest (resp. second oldest) item in a queue. Then, xp^1 and xp^2 are compatible.*

Proof: Let p and q be two different processes. Because the value the peek operation by p returns (i.e., the first or second) does not affect the state of the queue object visible to q, it follows that $xp^1[q]xp^2$ and the states of all the objects that q can access are the same at these runs. Thus, by Lemma 3.1, xp^1 and xp^2 are compatible. ∎

4 Relaxing the enqueue operation

It is obvious that $CN(queue[1, 0, 1]) = \infty$. Each process inserts its input value into the queue using an enqueue operation, and then uses a peek operation to find out what is the value at the front of the queue and decides on it. Also, it is obvious that, for an uninitialized queue, $CN(queue[0, 1, 1]) = 1$.[4] That is, a relaxed uninitialized queue where the enqueue operation is not supported is useless. Assume a queue object where only the enqueue operation may be relaxed. We show that only when the enqueue operation can insert a value at an arbitrary position, the consensus number drops to two; otherwise, in all other possible relaxations in which the enqueue operation is supported, the consensus number is not effected (i.e., it is ∞).

Theorem 4.1 $CN(queue[*, 1, 1]) = 2$.

Proof: It follows immediately from Lemma 2.2, Lemma 2.3, and Lemma 2.4(a) that $CN(queue[*, 1, 1]) \geq 2$. We prove that $CN(queue[*, 1, 1]) \leq 2$. A possible correct behavior of a $queue[*, 1, 1]$ object, is that every enqueue operation always inserts a data item at the head of the queue. In such a case, the $queue[*, 1, 1]$ object, behaves like a $stack[1, 1, 1]$ object. This implies that $CN(queue[*, 1, 1]) \leq CN(stack[1, 1, 1])$. Thus, by Lemma 2.4(c), $CN(queue[*, 1, 1]) \leq 2$. ∎

Corollary 4.2 *For every* $b \in Z^+ \cup \{*\}$, $c \in Z^+ \cup \{0, *\} : CN(queue[*, b, c]) = 2$.

Proof: The corollary follows from Lemma 2.2, Lemma 2.3, Lemma 2.4(a) and Theorem 4.1. ∎

Next we show that when the enqueue operation is relaxed but *can not* insert a value at an arbitrary position, the consensus number is infinity.

Theorem 4.3 *For every* $a \in Z^+ : CN(queue[a, 0, 1]) = \infty$.

Proof: For any given number $a \in Z^+$, we present a simple consensus algorithm for any number of processes using a singe $queue[a, 0, 1]$ object. Each process first enqueues its input value $a + 1$ times. Then, the process uses a peek operation to find out the value of the first item in the queue, and decides on that value. Clearly, once some process finishes to enqueue its input value $a + 1$ times, the value of the item at the head of the queue never changes. The result follows. ∎

Corollary 4.4 *For every* $a \in Z^+$, $b \in Z^+ \cup \{0, *\} : CN(queue[a, b, 1]) = \infty$.

Proof: The corollary follows immediately from Lemma 2.2, Lemma 2.3 and Theorem 4.3. ∎

[4] This is false, if the queue initially contains one element. In such a case, two processes can solve consensus, by deciding on the input of the process that successfully dequeues the element.

5 Relaxing the peek operation

Assume a queue object where only the *peek* operation may be relaxed. We show that in *all* possible relaxations of the peek operation the consensus number drops (from infinity) to two.

Theorem 5.1 $CN(queue[1,1,2]) = 2$.

Proof: It follows from Lemma 2.3 and Lemma 2.4(b) that $CN(queue[1,1,2]) \geq 2$. Below we prove that $CN(queue[1,1,2]) \leq 2$. By contradiction, assume that we have a wait-free consensus algorithm for *three* processes p, q and g using only $queue[1,1,2]$ objects and registers. By Lemma 3.3, the algorithm has a critical run x. By definition of a critical run, for two of the processes, say p and q, a run resulting by an extension of x by a single event of p and a run resulting by an extension of x by a single event of q are not compatible. Thus, by Lemma 3.4, in their next events from x, p and q are accessing the same object, which must be a $queue[1,1,2]$ object. By Lemma 3.5, if the next event of p (resp. q) from x is a relaxed peek event which may return one of the two oldest items in a queue, xp^1 and xp^2 (resp. xq^1 and xq^2) are compatible. Below, when the next event of p from x is a peek event, xp refers to xp^1 and xp^2.

Without loss of generality, we can assume the xp is 0-valent and xq is 1-valent. Since xp is 0-valent also xpq is 0-valent. Since xq is 1-valent also xqp is 1-valent. Thus, xpq and xqp are not compatible. Next, we consider all the possible cases, regarding the next two events of p and q from x and show that each one of these cases leads to a contradiction.

Case 1. Both events are *peek* events. Because a peek operation does not have any effect on the state of a $queue[1,1,2]$ object, it follows that $xpq[g]xqp$ and the states of all the objects that g can access are the same at these runs. Thus, by Lemma 3.1, xpq and xqp must be compatible, a contradiction. Notice that we do not really care what value a peek operation returns (i.e., the oldest or second oldest), since this will not affect the state of the object visible to g.

Case 2. Exactly one of the two events is a *peek* event. Because the peek operation does not have any effect on the state of a $queue[1,1,2]$ object and the other operation has the same effect in both xpq and xqp, it follows that $xpq[g]xqp$ and the states of all the objects that g can access are the same at these runs. Thus, by Lemma 3.1, xpq and xqp must be compatible, a contradiction. Notice that again we do not really care what value the peek operation returns.

Case 3. Both events are *dequeue* events. In the last two events in xpq and xqp the same two items were removed from the queue, thus, $xpq[g]xqp$ and the states of all the objects that g can access are the same at these runs. Thus, by Lemma 3.1, xpq and xqp must be compatible, a contradiction.

Case 4. One event is a *enqueue* and the other is a *dequeue*. Assume w.l.o.g. that the enqueue event is by p and the dequeue event is be q. If the queue is nonempty, the two events commute since each operates on a different end of the queue. Thus, xpq and xqp are indistinguishable for all the processes and the states of all the objects is the same in xpq and xqp, and thus by Lemma 3.1 the contradiction is immediate. If the queue is empty, $xp[g]xqp$ and the states of all the objects that g can access are the same at these runs. Thus, by Lemma 3.1, xp and xqp must be compatible, a contradiction.

Case 5. Both events are *enqueue* events. Assume that p enqueues the value v_p and q enqueues the value v_q. Consider the runs xpq and xqp. The valency of each one of these two runs is determined by the process that has taken the first step from x. If p or q runs uninterrupted starting from either xpq or xqp, the only way for each one of them to observe the queue's state is via a *dequeue* or a *peek* operation. However, since the peek operation can return one of the first two items at the

head of the queue, a *peek* can not be used to determine which process enqueue operation was first. More precisely, after the two enqueue events by p and q, in all extension of x,

1. as long as the values v_p and v_q are not at the head of queue, the adversary can force every peek event to always return the item at the head of the queue, and thus it is not possible to decide which process enqueue event was first;

2. if the values v_p and v_q are at the head of queue (in some order), the adversary can force every peek operation to always return the *same* value, say v_p, regardless whether v_p is the first or second element. Thus, again, it is not possible to decide which process enqueue event was first.

Thus, the only way for a process to determine which process went first is via dequeue operations. Next we consider the following two extensions of xpq and xqp.

- Let y be an extension of xpq that results from the following execution: Starting from x let p enqueue v_p and then let q enqueue v_q. Run p uninterrupted until it dequeues v_p (as explained above this is the only way for p to observe which process went first). Then, run q uninterrupted until it dequeues v_q.

- Let y' be an extension of xqp that results from the following execution: Starting from x let q enqueue v_q and then let p enqueue v_p. Run p uninterrupted until it dequeues v_q Then, run q uninterrupted until it dequeues v_p.

Since y is and extension of xpq, y is 0-valent, and since y' is and extension of xqp, y' is 1-valent. Clearly, $y[g]y'$ and the states of all the objects that g can access are the same at these runs. Thus, by Lemma 3.1, y and y' must be compatible, a contradiction. ∎

Corollary 5.2 *For every* $a \in Z^+ \cup \{*\}$, $b \in Z^+ \cup \{*\}$, $c \neq 1 : CN(queue[a, b, c]) = 2$.

Proof: The corollary follows from Lemma 2.2, Lemma 2.4(a) and Theorem 5.1. ∎

6 Not supporting the dequeue operation

The consensus number of all the queues where the dequeue operation is not supported *and* the peek operation is relaxed, is just 1. Put another way, while $CN(queue[1, 0, 1]) = \infty$ and $CN(queue[1, 1, 2]) = 2$, by removing the dequeue operation from the object $queue[1, 1, 2]$, we get an object with consensus number one. That is,

Theorem 6.1 $CN(queue[1, 0, 2]) = 1$.

Proof: By contradiction, assume that we have a wait-free consensus algorithm for *two* processes p and q using only $queue[1, 0, 2]$ objects and registers. By Lemma 3.3, the algorithm has a critical run x. By definition of a critical run, a run resulting by an extension of x by a single event of p and a run resulting by an extension of x by a single event of q are not compatible. By Lemma 3.4, in their next events from x, p and q are accessing the same object, which must be a $queue[1, 0, 2]$ object. By Lemma 3.5, if the next event of p (resp. q) from x is a relaxed peek event which may return one

of the two oldest items in a queue, xp^1 and xp^2 (resp. xq^1 and xq^2) are compatible. Below, when the next event of p from x is a peek event, xp refers to xp^1 and xp^2.

Without loss of generality, we assume the xp is 0-valent and xq is 1-valent. Since xp is 0-valent also xpq is 0-valent. Since xq is 1-valent also xqp is 1-valent. Thus, xpq and xqp are not compatible. Next, we consider all the possible cases, regarding the next events of p and q from x and show that each one of these cases leads to a contradiction.

Case 1. Both events are *peek* events. Because a peek operation does not have any effect on the states of the $queue[1, 0, 2]$ object, it follows that $xp^1[p]xqp^1$ and the states of all the objects that p can access are the same at these runs. Thus, by Lemma 3.1, xp^1 and xqp^1 must be compatible, a contradiction. We do not really care what value a peek operation by q returns since this will not affect the state of the object visible to p.[5]

Case 2. Exactly one of the two events is a *peek* event. Assume w.l.o.g. that the peek event is by process q. Because the peek operation does not have any effect on the states of the $queue[1, 0, 2]$ object and the operation by p has the same effect in both xp and xqp, it follows that $xp[p]xqp$ and the states of all the objects that p can access are the same at these runs. Thus, by Lemma 3.1, xp and xqp must be compatible, a contradiction. Notice that again we do not really care what value the peek operation returns.

Case 3. Both events are *enqueue* events. Assume that p enqueues the value v_p and q enqueues the value v_q. Consider the 0-valent run xpq and the 1-valent run xqp. The valency of each one of these two runs is determined by the process that has taken the first step from x. If p or q runs uninterrupted starting from either xpq or xqp, the only way for each one of them to observe the queue's state is via a *peek* operation. Since the peek operation can return one of the first two items at the head of the queue, a *peek* can not be used to determine which process enqueue operation was first. More precisely:

1. If the queue is *not* empty at x then after the two enqueue events by p and q, the adversary can force every peek event to always return the item at the head of the queue, and thus it is not possible for p or q to decide which process enqueue event was first.

2. If the queue is empty at x then after the two enqueue events by p and q, the values v_p and v_q are at the head of queue (in some order). Now the adversary can force every peek operation by p to always return v_p, and every peek operation by q can always return v_q. Thus, again, it is not possible to decide which process enqueue event was first.

Thus, it can not be that both events are enqueue events, a contradiction. ∎

Corollary 6.2 *For every* $a \in Z^+ \cup \{0, *\}$, $c \geq 2 : CN(queue[a, 0, c]) = 1$.

Proof: The corollary follows from Lemma 2.2, Lemma 2.3 and Theorem 6.1. ∎

[5]Notice that we reach a contradiction, by assuming that p's peek operation returns the first element in both passes. Since this implies that it cannot be the case that both events by p and q are *peek* events, there is no need to consider the sub-case where p's peek operation returns the first element in one path and the second element is the other path.

7 Atomic registers vs. relaxed queues

It is known that $CN(atomic\ register) = 1$ [21]. It is easy to see that a $queue[*, 0, 1]$ has a trivial wait-free implementation from a single atomic register, which raises the question whether also $queue[1, 0, 2]$ has a wait-free implementation from atomic registers. The answer to this question is negative. We prove the following general result:

Theorem 7.1 *A $queue[a, 0, c]$ has no wait-free implementation from atomic registers, for every two integers $a \geq 1$ and $c \geq 1$.*

Proof: The *(n, k)-set consensus problem* is to design an algorithm for n processes, where each process starts with an input value from some domain, and must choose some participating process' input as its output. All n processes together may choose no more than k distinct output values. An (n, k)-set consensus object (or algorithm) is an object which solves the (n, k)-set consensus problem. One of the most celebrated impossibility results in distributed computing is that, for any $1 \leq k < n$, a wait-free (n, k)-set consensus object can not be implemented using any number of wait-free $(n, k + 1)$-set consensus objects and atomic registers [5, 13, 26].

We show that, for any for every two positive integers a and c, a wait-free $(a+c, a+c-1)$-set consensus object has a simple wait-free implementation using a single (initially empty) $queue[a, 0, c]$ object, as follows. Each process p_i inserts its input value v_i into the queue using an $enq.a(v_i)$ operation, and then uses a $peek.c()$ operation to find a value in one of the c positions at the front of the queue, and decides on it. During the execution any one of the $a + c - 1$ values that are inserted first into the queue can occupy (at some point in time) one of the c positions at the front of the queue. Any value that is inserted later will never occupy one of the c positions at the front of the queue. Thus, processes together will never choose more than $a + c - 1$ distinct output values. Since, for every two positive integers a and c, it is possible to solve in a wait-free manner the $(a+c, a+c-1)$-set consensus problem using a $queue[a, 0, c]$ object, but it is not possible to solve $(a + c, a + c - 1)$-set consensus in a wait-free manner using atomic registers, the result follows. ∎

8 Relaxed registers

It is common to assume that operations on the same memory location are atomic – they occur in some definite order. However, this assumption can be relaxed allowing the possibility of concurrent operation on the same memory location. In [20], Lamport has defined three classes of shared registers which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. Below we consider natural generalizations of Lamport's notions.

8.1 k-safe, k-regular and k-atomic registers

Unless otherwise stated, for the rest of this section. it is assumed that each register is a single-writer multi-reader register. Such a register can be written by one predefined process and can be read by all the processes. Let k be a positive integer.

- The weakest possibility is a *k-safe* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. No assumption is made

about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register. We consider the initial value as the first written value.

- The next stronger possibility is a *k-regular* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. A read that overlaps a write obtains either the new value or one of the k most recently written values. That is, a read that overlaps any series of writes obtains either one of the values being written or one of the k most recently written values before the first of the writes.

- The final possibility is a *k-atomic* register, in which the reads and writes behave as if they occur in some definite order, and a read obtains one of the k most recently written values. In other words, for any execution, there is some way of totally ordering the overlapping reads and writes so that the value returned by the each read is one of the k most recently written values in the execution which has no overlapping. (Operations that do not overlap should take effect in their "real-time" order.)

We observe that Lamport's familiar notions of safe, regular and atomic registers are equivalent to the notions of 1-safe, 1-regular and 1-atomic registers, respectively. We will use the notion k-register as an abbreviation for k-safe, k-regular and k-atomic registers, when the exact type of a register is not important. We notice that, for $k \geq 1$, a k-atomic register is exactly the object $stack[1, 0, k]$, as defined in the Introduction.

8.2 Two constructions

We present two constructions of registers, by indicating how write operations and read operations are performed. A register can be either a single-writer single-reader (SWSR) register, a single-writer multi-reader (SWMR) register or a multi-writer multi-reader (MWMR) register. Unless explicitly stated, we assume that a register is a SWMR register. We require that the constructions satisfy the wait-freedom progress condition.

The first construction implements a single-writer *multi-reader* multi-valued k-safe, k-regular or $(k+1)$-atomic register, denoted r, from single-writer *single-reader* multi-valued k-safe, k-regular or k-atomic registers, respectively.

Construction 8.1 *Let k be an arbitrary natural number, and let $r_1, ..., r_n$ be SWSR multi-valued k-registers, where each r_i $(i \in \{1, ..., n\})$ can be written by the same single process and read by process p_i. We construct a SWMR multi-valued k-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** n **do** $r_i := value$;

- *The read operation of r by process p_i is performed by letting p_i read the value of r_i.*

The above construction is similar to Construction 8.1 from [20] which was designed for 1-registers. We prove the following theorem for the general case of k-registers.

Theorem 8.1 *The following claims are correct w.r.t. Construction 8.1, for any $k \geq 1$,*

1. *If $r_1, ..., r_n$ are SWSR k-safe registers or $r_1, ..., r_n$ are SWSR k-regular registers then r is a SWMR k-safe register or a SWMR k-regular register, respectively.*

2. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is a SWMR $(k+1)$-atomic register.*

3. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is not a SWMR k-atomic register.*

Proof: A read of r by process p_i that does not overlap a write of r, also does not overlap a write of r_i. If r_i is k-register (i.e., if r_i is k-safe, k-regular or k-atomic), then this read must obtain one of the k most recently written values into r. This is enough to show that if r_i is k-safe then r is k-safe. If a read of r_i by process p_i overlaps a write of r_i, then it overlaps a write of the same value to r. In such a case, if r_i is k-regular then this read must obtain either the last value written or one of the k most recently written values into r_i (and hence into r). This implies that if r_i is k-regular then r is k-regular.

Now, assume that r_i is k-atomic, and that a read of r_i by process p_i overlaps a write of the value v into r. Then (1) if v was already written into r_i, this read must obtain either the value v or one of the $k-1$ most recently written values into r_i before v; or (2) if v was not written into r_i yet, this read must one of the k most recently written values into r_i. Since the linearization point of write of v into r might be before the linearization point of the read of v, in both cases above, the returned value is one of the $k+1$ most recently written values into r. This implies that if r_i is a SWSR k-atomic register then r is a SWMR $k+1$-atomic register.

Assume that $r_1, ..., r_n$ are k-registers. If a read of r by two different processes p_i and p_j both overlap the same write of value v into r, it is possible for p_i to get the new value v and for p_j to get the kth written value into r before the value v was written. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a k-atomic register. Thus, r is not a k-atomic register. ∎

The second construction implements a SWMR multi-valued 1-safe or 1-regular register from SWMR multi-valued k-safe or k-regular registers, respectively.

Construction 8.2 *Let k be an arbitrary natural number, and let r' be a SWMR multi-valued k-registers. We construct a SWMR multi-valued 1-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** k **do** $r' := value$;
- *The read operation of r by process p is performed by letting p read the value of r'.*

Theorem 8.2 *The following claims are correct w.r.t. Construction 8.2, for any $k \geq 1$,*

1. *If r' is a SWMR k-safe register or a SWMR k-regular register then r is a SWMR 1-safe register or a SWMR 1-regular register, respectively.*
2. *If r' is a SWMR k-atomic register then r is not a SWMR 1-atomic register.*

Proof: A read of r by process p that does not overlap a write of r, also does not overlap any of the latest k writes of r'. Thus, all the k most recently written values into r' are identical and equal the most recent value written into r'. Since r' is k-register, in the case of no overlap, a read of r must obtain one of the k most recently written values into r', and thus it must obtain the most recent value written into r'. This is enough to show that if r' is k-safe then r is 1-safe.

If a read of r' by process p overlaps a write of r', then it overlaps a write of the same value to r. In such a case, if r' is k-regular then this read must obtain either the new value or one of the k most recently written values into r' (and hence into r). However, since each value is written k times, each of the k most recently written values equals either the new value or the most recent value written before the new value. This implies that if r' is k-regular then r is 1-regular.

We have assumed that r' is a k-register. Thus, during a write of r, the k most recently written values into r' equals either the new value or the most recent value written before the new value. If a read of r by two different processes p_i and p_j both overlap the same write of value v into r, it is possible for p_i to get the new value v and p_j the old value. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a 1-atomic register. Thus, r is not a 1-atomic register. ∎

We notice that Construction 8.2 can be used for implementing a 2-atomic register from k-atomic registers. It would be interesting to find a similar simple and efficient construction also for implementing 1-atomic register from k-atomic registers.

8.3 Main result regarding relaxed registers

Our result is about computability of using k-registers. We show that for every $k \geq 1$, k-safe registers and 1-atomic registers have the same computational power. More precisely, it is possible to wait-free implement multi-writer multi-reader multi-valued 1-atomic registers using single-writer single-reader k-safe bits, for every $k \geq 1$.

Theorem 8.3 *For every $k \geq 1$, it is possible to construct a MWMR 1-atomic register using SWSR k-safe bits.*

Proof: It follows from Construction 8.1 and Construction 8.2 that it is possible to implement a SWMR 1-safe bit using SWSR k-safe bits. A well known result is that it is possible to implement a MWMR multi-valued 1-atomic register from SWMR 1-safe bits (see Chapter 4 of [10]). The result follows. [6] ∎

For $k \geq 1$, the object $stack[1, 0, k]$ is exactly a k-atomic register. Thus, the following corollary follows immediately from Theorem 8.3:

Corollary 8.4 *For every $k \geq 1$, a $stack[1, 0, 1]$ object can be implemented from $stack[1, 0, k]$ objects.*

There are several classical synchronization algorithms, such as a mutual exclusion algorithm and a ℓ-exclusion algorithm, that only use SWMR 1-safe registers, for interprocess communication [30]. Using Construction 8.2, such algorithms can be very easily and efficiently modified to use only k-safe registers, for every $k \geq 1$. Examples of such modifications can be found in [31].

8.4 Practical implications

Various optimizations enable reordering memory references as it allows much better performance. When a correct operation depends on ordered memory references, memory barriers are used to prevent reordering [22, 23]. Memory barriers are required to enable good performance and scalability. The reason for that is the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access [24].

[6] The known constructions of a MWMR multi-valued 1-atomic register from SWMR 1-safe bits, are complicated and are not practically useful for transforming algorithms that use strong type of registers into algorithms that use weak type of registers.

Without using memory barriers, as a result of reordering, a read from an atomic register may obtain some older value when compared to the value this read would return in the in order execution of the original program code. Suppose that in some setting where reordering is possible, a read may obtain, in the worst case, one of the 5 most recently written values when compared to the value it would return in the in order execution. In such a case, no harm done, if the program which uses 1-atomic registers, was designed in the first place to work correctly assuming that communication is done via 5-atomic registers.

Consider the following design strategy: Design your algorithms to be correct when k-atomic registers are used for some $k > 1$. Now replace the k-atomic registers with the stronger 1-atomic registers. In such algorithms the use of memory barriers may not be necessary in some cases, even when reordering is possible. Thus, there is a tradeoff between the number of memory barriers needed to ensure correctness and the type of k-registers used. Put another way, proving correctness w.r.t. k-registers while actually using 1-registers provides some level of resiliency against memory reordering. Finding the exact level of resiliency provided using such a design strategy, as a function of k, is an interesting research topic which is not covered in this paper.

9 Related work

The design of concurrent data structures has been extensively studied [10, 30]. However, there are limitations in achieving high scalability in their design [4, 6]. Two progress conditions that have been proposed for data structures which avoid locking are wait-freedom [11] (defined earlier), and obstruction-freedom [12]. Obstruction-freedom, a weak form of implementations that do not use locks, guarantees that an active process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough.

It is shown in [4] that the worst-case operation time complexity of obstruction-free implementations is high, even in the absence of step contention. In [6], an $\Omega(n)$ lower bound is proven on the time to perform a single instance of an operation in any implementation of a large class of data structures shared by n processes, such as counters, stacks, and queues. It is suggested in [6] that "it might be beneficial to replace linearizable implementations of strongly ordered data structures, such as stacks and queues, with more relaxed data structures, such as pools and bags".

In [27], it is pointed out that concurrent data structures will have to go through a substantial "relaxation process" in order to support scalability: "The data structures of our childhood – stacks, queues, and heaps – will soon disappear, replaced by looser *unordered* concurrent constructs based on distribution and randomization". A few examples are given in [27] showing how relaxing a stack's LIFO ordering guarantees can result in higher performance and greater scalability.

Another approach to weaken the requirement of traditional data structures is not to change at all the definition of the data structures, but rather to relax the traditional correctness requirements (i.e., linearizability [14] and sequential consistency [17]). A tutorial which describes many issues related to memory consistency models can be found in [1]. In the context of relaxing the consistency condition linearizability [14], two relaxations of a queue were presented in [3]. In [16], a k-FIFO queue was implemented, which may dequeue elements out of FIFO order up to a constant $k \geq 0$. There are various implementations of relaxed data structures where insertion-order is of no importance, such as pools and bags, see for example [2, 29]. In [9], a systematic and formal framework is presented for obtaining new data structures by quantitatively relaxing existing ones.

The relative power of various shared objects has been studied extensively in shared memory environments where processes may fail benignly, and where every operation is wait-free. In [11],

a hierarchy of progressively stronger shared objects, is defined. Objects at each level are able to perform tasks which are impossible for objects at the lower levels.

The consensus problem was defined in [25]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure was first proved for the asynchronous message-passing model in [7], and later has been extended for the shared memory model with atomic registers in [21]. The impossibility result that, for $1 \leq k \leq n-1$ there is no k-resilient k-set-consensus algorithm for n processes using atomic registers, is from [5, 13, 26] (set-consensus is defined in section 6). It is shown in [11] that traditional data types, such as sets which support insert and remove operations, queues which support enqueue and dequeue operations (i.e., $queue[1,1,0]$), stacks which supports push and pop operations (i.e., $stack[1,1,0]$), all have consensus number exactly two. In the proofs of [11], it is assumed that the data structures are initialized. The same results also hold for the case of uninitialized data structures [15, 30]. It is trivial to show that $CN(queue[1,1,1]) = \infty$.

A formalism for reasoning about concurrent systems which does not assume that read and write are atomic operations is developed in [18, 19]. This formalism has been further developed in [20], where it is used to specify several classes of interprocess communication mechanism and to prove correctness of algorithms for implementing them. In particular, a very weak form of (non-atomic) shared register, called *safe* register, is defined [20]. A safe register can be written and read concurrently, but read errors may occur during the writing of a shared register. Following the publication of [20], many papers were published about implementing one type of shared register from another. In particular it is now known how to implement a MWMR atomic register from SWSR safe bits. A compilation of some of these results can be found in Chapter 4 of [10]. A discussion on how memory barriers are used to prevent reordering can be found in [22, 23, 24].

10 Discussion

Synchronization inherently limits parallelism. As a result, there is a recent trend towards implementing semantically weaker data structures which reduce the need for synchronization and thus achieve better performance and scalability. We have considered infinitely many possible relaxations of queues and stacks, and examined their relative computational power by determining their consensus numbers. We have also introduced the new notions of k-safe, k-regular and k-atomic registers, and showed how to implement 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type).

Our results demonstrate, somewhat surprisingly, that each one of the infinitely many relaxed objects considered has one of the following three consensus numbers: 1, 2 or ∞. Another conclusion is that a queue is more sensitive than a stack to changes in its semantics. It would be interesting to extend our results to other data structures. Various limitations in achieving high scalability in the design of traditional data structures have been demonstrated in [4, 6]. Can these results be extended to cover some of the relaxed data structures presented in this paper?

It would be interesting to find out the internal structure among relaxed objects in the same level of the consensus hierarchy. In particular, for $i \in Z^+$, is it possible to implement a $queue[1,1,i+1]$ object using $queue[1,1,i+2]$ objects and registers? Is it possible to implement a $queue[1,1,2]$ object using $queue[1,1,0]$ objects and registers?

Acknowledgements: We wish to thank the three anonymous referees and the associate editor Eric Ruppert for their constructive suggestions and corrections. Support is gratefully acknowledged from

the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, and the Department of Energy under grant ER26116/DE-SC0008923.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 151–162, 2010.

[3] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, OPODIS'10, pages 395–410, 2010.

[4] H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4):1–33, July 2009.

[5] E. Borowsky and E. Gafni. Generalizecl FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 91–100, 1993.

[6] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[7] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[8] E. Gidron, I. Keidar, D. Perelman, and Yonathan Y. Perez. Salsa: Scalable and low synchronization numa-aware algorithm for producer-consumer pools. In *Proc. 24th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '12, pages 151–160, 2012.

[9] T. Henzinger, C. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Not.*, 48(1):317–328, January 2013.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008. 508 pages.

[11] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[12] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, page 522, 2003.

[13] M. P. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, July 1999.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: LNCS 674*, pages 69–84, 1992.

[16] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent fifo queues. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, 2012. *LNCS 7439*, 273–287.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, September 1979.

[18] L. Lamport. The mutual exclusion problem: Part I – a theory of interprocess communication. *Journal of the ACM*, 33:313–326, 1986.

[19] L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33:327–348, 1986.

[20] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[21] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[22] P. E. McKenney. Memory ordering in modern microprocessors, part i. *Linux Journal*, 136, 2005. 2 page (Revised April 2009.).

[23] P. E. McKenney. Memory ordering in modern microprocessors, part ii. *Linux Journal*, 137, 2005. 2 page (Revised April 2009.).

[24] P. E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.

[25] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[26] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29, 2000.

[27] N. Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, March 2011.

[28] N. Shavit and G. Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. In *22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2015)*, July 2015.

[29] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, 2011.

[30] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

[31] G. Taubenfeld. Weak read/write registers. In *14th Inter. Conf. on Distributed Computing and Networking (ICDCN 2013)*, January 2013. *LNCS 7730* Springer Verlag 2013, 423–427. Full version is avaliable at: http://www.faculty.idc.ac.il/gadi/Publications.htm.