algorithms

Article

Relaxed Data Types as Consistency Conditions [†]

Edward Talmage *,‡ **and Jennifer L. Welch** ‡

Department of Computer Science & Engineering, Texas A&M University, College Station, TX 77843, USA; welch@cse.tamu.edu

* Correspondence: etalmage@tamu.edu

† This paper is an extended version of our paper published in 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS).

‡ These authors contributed equally to this work.

Received: 27 February 2018; Accepted: 2 May 2018; Published: 4 May 2018

Abstract: In the quest for higher-performance shared data structures, weakening consistency conditions and relaxing the sequential specifications of data types are two of the primary tools available in the literature today. In this paper, we show that these two approaches are in many cases different ways to specify the same sets of allowed concurrent behaviors of a given shared data object. This equivalence allows us to use whichever description is clearer, simpler, or easier to achieve equivalent guarantees. Specifically, for three common data type relaxations, we define consistency conditions such that the combination of the new consistency condition and an unrelaxed type allows the same behaviors as Linearizability and the relaxed version of the data type. Conversely, for the consistency condition k-Atomicity, we define a new data type relaxation such that the behaviors allowed by the relaxed version of a data type, combined with Linearizability, are the same as those allowed by k-Atomicity and the original type. As an example of the possibilities opened by our new equivalence, we use standard techniques from the literature on consistency conditions to prove that the three data type relaxations we consider are not comparable to one another or to several similar known conditions. Finally, we show a particular class of data types where one of our newly-defined consistency conditions is comparable to, and stronger than, one of the known consistency conditions we consider.

Keywords: distributed data structures; relaxations; consistency conditions

1. Introduction and Background

Shared data types are an essential abstraction in distributed computing, as they provide a consistent interface for multiple processes to interact with shared data. Shared data access is more complex than local access, as multiple processes can concurrently access and change the stored values. Thus, a single process cannot assume that it will find the same value in the shared object as it last put there, which makes it non-trivial to interpret the value found in a shared object. The value of a shared object may also not be well-defined when another process is changing a stored value at the same time one is trying to read the value. A data type specification provides guarantees on the changes that other processes may make, and defines the states which a shared data object may take on. These guarantees ease the effort programmers building distributed systems must spend on coordinating different processes. In addition, by abstracting and efficiently implementing the oft-repeated tasks of shared data access and manipulation, overall program efficiency can be increased.

It is thus important to provide the best possible guarantees on the behavior of data types under concurrent access to shared data while maintaining the efficiency of those interfaces. The study of *consistency conditions* considers what guarantees may be provided or required on the behavior of shared data objects under concurrent access. The strongest consistency condition, Linearizability [1], requires that all operations on shared data appear to all processes as if they happened sequentially, in an order respecting the order of operations which do not overlap in real time. This makes program design and reasoning about program correctness relatively easy, as we are familiar with sequential program design and analysis. However, Linearizability is generally expensive to implement, in terms of computation and communication delays. For example, Attiya et al. [2] showed that it is impossible to linearizably implement many common data types without expensive synchronization steps; Attiya and Welch [3] showed that Linearizability is necessarily more expensive than sequential consistency; Lipton and Sandberg [4] showed that requiring even a consistency weaker than Linearizability is expensive; and Wang et al. [5] and Talmage and Welch [6] showed lower bounds on the worst-case and amortized cost of certain operations under Linearizability. To avoid this cost, many weaker consistency conditions have been proposed (see Viotti and Vukolic [7] for a review of consistency conditions in the literature), allowing more concurrent executions while providing weaker guarantees on the behavior of shared objects. Under these conditions, data types can be implemented more efficiently in terms of operation response time than they can under Linearizability (e.g., [3]). Some work has been done to explore classes of data types which, when implemented under a weak consistency condition, give stronger behavioral guarantees than those of the consistency condition (e.g., [8]).

Another approach for increasing the efficiency of distributed data type implementations that has recently gained popularity is to *relax* the sequential specification of the data type. Afek et al. [9] defined a weak consistency condition in a manner similar to how we consider relaxations. Henzinger et al. [10] formalized and generalized the notion of relaxations. Several papers have implemented a variety of relaxed data types (e.g., [11]). By allowing some specific behaviors that were otherwise illegal, particularly by allowing non-deterministic choices between possible behaviors, several papers have shown both empirically and formally that it is possible to reduce the (amortized) implementation costs of data types [6,10,12–14]. This does come at some cost to the computational strength of the implemented data type [15,16]. Proposed applications for relaxed data types include relaxed sets as work-stealing queues [17], counters that miss some increments but are statistically useful [9], and relaxed queues for webserver request handling [18]. Data type relaxations are generally (to date) implemented under Linearizability, so all new behaviors are specified sequentially. Sequential behaviors are often easier to understand, and thus use correctly, than complex conditions on concurrent executions, which are hard to visualize.

In this paper, we explore the relation of these two different methods for improving the performance of shared data type implementations. We show that the combination of Linearizability and three data type relaxations common in the literature, k-Out-of-Order, k-Lateness, and k-Stuttering [10], can be alternately defined as consistency conditions. That is, the set of concurrent executions which are considered legal under Linearizability when working with the relaxed type is the same as the set of concurrent executions which are legal under the new consistency condition and the original, unrelaxed type. Conversely, we show, with the example of k-Atomicity, that some consistency conditions can be separated into Linearizability and a sequential data type relaxation.

This partial equivalence means that for several common relaxations and consistency conditions, the relaxation and consistency condition definitions are interchangeable. As an example of the use of this interchangeability, we use ideas from the large body of work comparing the strengths of different consistency conditions [7,19–21] to show that the consistency conditions equivalent to k-Out-of-Order, k-Lateness, and k-Stuttering are distinct from similar previously known consistency conditions. Despite this general distinction, for some particular data types, we show that k-Stuttering is a strengthening of k-Atomicity.

A preliminary version of this work appeared as [22]. This paper extends that work by including proofs and adding Section 7's discussion of hybrid consistencies.

2. Model

2.1. Data Types

An *Abstract Data Type* specifies an interface for interacting with data, and defines how a data object instantiating that type will behave. Data type specifications consist of the possible operations which a process may invoke and a set of sequences of operation invocation-response pairs, called *instances*, which specify all possible return values an operation response may have, given a sequence of past operations and an invocation. We here consider only objects with sequential specifications, as relaxation of tasks without sequential specifications (see, e.g., [23,24]) has not been defined.

Definition 1. *An Abstract Data Type consists of*

1. *A set OPS of operations and the sets $args(OP)$ of valid arguments and $rets(OP)$ of valid return values for each $OP \in OPS$. An* instance *of an operation OP, denoted $OP(arg, ret)$, contains the argument(s) arg and the value(s) returned, ret. In a sequential environment, instances are indivisible, but we will consider them as a distinct invocation and matching response in the distributed setting.*

   When either $args(OP)$ or $rets(OP)$ contains only a null value ($\perp$), we condense the notation to $OP(arg)$ or $OP(ret)$, as appropriate.
2. *A set $\mathcal{L}$ of sequences of operation instances, called* legal sequences, *which satisfies two properties:*
   - *(a)* Prefix Closure: *If a sequence ρ is in $\mathcal{L}$, then every prefix of ρ is also in $\mathcal{L}$.*
   - *(b)* Completeness: *If a sequence ρ is in $\mathcal{L}$, then for every operation $OP \in OPS$ and every argument arg for OP, there is a response ret such that $\rho.OP(arg, ret)$ is in $\mathcal{L}$, where "." represents concatenation of sequences.*

The intuitive notion of the state of a shared object is determined by the sequence of past operation instances on that object. We say that two such, not necessarily equal, sequences π and ρ are *equivalent*, denoted $\pi \equiv \rho$, if for any sequence σ where either $\pi.\sigma$ or $\rho.\sigma$ is legal, then $\rho.\sigma$ or $\pi.\sigma$ is also legal, respectively. We classify operations by whether they change a shared object's state, return information about it, or both.

Definition 2. *An operation OP of an abstract data type T is a* mutator *if there exists a legal sequence ρ of instances of operations of T and an instance op of OP such that $\rho \not\equiv \rho.op$. An operation OP is an* accessor *if there exist legal sequences ρ, ρ' of instances of operations of T and an instance aop of OP such that $\rho.aop$ is legal, but $\rho'.aop$ is not legal.*

An operation which is both an accessor and a mutator is a mixed *operation. An operation which is a mutator but not an accessor, or an accessor but not a mutator, is a* pure *mutator or accessor, respectively.*

For example, in an RMW register, *Read* is a pure accessor, *Write* is a pure mutator, and *Read-Modify-Write* is a mixed operation. In a FIFO queue augmented with *Peek*, *Enqueue* is a pure mutator, *Dequeue* is a mixed operation, and *Peek* is a pure accessor. A data type does not need to have all three kinds, as seen in a *Read*/*Write* register or classic queue without *Peek*.

Note that removing all instances of pure accessors from a sequence of operation instances π does not change the state represented, so we denote this equivalent sequence containing only mutator instances as $\pi|_m$.

We next give two example definitions of classic data types we will use in this paper.

Definition 3 (*Read*/*Write* Register)**.**

1. *$OPS = \{Read, Write\}$, $args(Read) = \{\perp\}$, $rets(Read) = \mathbb{Z}$,*
   $args(Write) = \mathbb{Z}$, and $rets(Write) = \{\perp\}$.

2. *$\mathcal{L}$ is the set of all sequences in which every Write instance returns $\perp$ and every Read instance returns the argument of the last Write instance preceding it.*

Definition 4 (FIFO Queue)**.**

1. *$OPS = \{Enqueue, Dequeue\}$,*
   $args(Enqueue) = \mathbb{Z}$, $rets(Enqueue) = \{\perp\}$,
   $args(Dequeue) = \{\perp\}$, and $rets(Dequeue) = \mathbb{Z}$.
2. *$\mathcal{L}$ is the set of all sequences in which every Enqueue instance returns $\perp$ and every Dequeue instance returns the argument of the first Enqueue instance whose argument has not yet been returned by a Dequeue or $\perp$ if no such Enqueue instance exists.*

This requires that each *Enqueue* has a unique argument, but this is straightforward to achieve by, e.g., applying logical timestamps to each *Enqueue* instance's argument. In the rest of the paper we assume that all operations invocations have unique arguments

2.2. Consistency Conditions

We consider an asynchronous, shared-memory model of computation among n processes. We split operation instances into separate invocations and responses. Processes interact by invoking operations, with arguments, on shared objects. Some time after an invocation, the object responds, giving the process a return value. Computation takes the form of *schedules*. A schedule of a data type T is a collection of sequences, one per process, of alternating invocations and responses of operations of T, each occurring at some real time and with each response being of the same operation as the previous invocation, so that they together form an instance of that operation. Each process' sequence is either infinite or ends in an operation response. In a schedule, we call two operation instances at different processes *overlapping* if the real time of one instance's invocation is between the real times of the invocation and response of the other instance. Ordering non-overlapping instances by their start times produces a partial order on all operation instances, called the *schedule order*.

Since data type specifications are inherently sequential, we need some way to relate a schedule of a distributed system, which is inherently concurrent, to those specifications. A consistency condition specifies what concurrent schedules are *legal* on a given data type.

Definition 5. *A* consistency condition *C is a set of schedules defined to be* legal *on any data type T under C.*

When discussing a consistency condition in conjunction with a particular data type, we implicitly consider only the subset of schedules for that type. This definition overloads the term "legal" to refer to schedules which correspond, by the consistency condition, to legal sequences on the given data type. Equality of consistency conditions is set equality between sets of legal schedules [7].

As an example, we define *Linearizability*, which is used throughout the literature in combination with relaxed data types, as it is the most intuitive consistency guarantee.

Definition 6 (Linearizability)**.** *A schedule E on a data type T is legal under Linearizability if there exists a permutation Π of all operation instances in E such that: (1) if an instance op precedes another instance op' in the schedule order, then op precedes op' in Π; and (2) Π is legal, according to the sequential specification of T.*

Weaker consistency conditions may allow some reordering with respect to the schedule order. For example, k-Atomicity for *Read*/*Write* registers, introduced in [25], allows *Read* operations to get a "stale" value, possibly missing some updates which overlap or even immediately precede the *Read* instance in the schedule order. This staleness is bounded by the constant k, ensuring that the behavior is not arbitrary. In practice, the values "missed" can reflect *Write* instances which the process invoking the *Read* has not yet heard about. That work gives probabilistic results showing that only requiring k-Atomicity can lead to implementations with higher proportions of operations which succeed, meaning that processes do not need to retry as often, improving performance.

3. Relaxed Data Types

We here present definitions of several relaxations introduced in [10]. We restate these definitions purely in terms of legal sequences of operation instances, where Henzinger et al. [10] combined equivalence classes of such sequences to develop a state machine notation. Several authors [6,12–16] used these and similar relaxations.

First, we consider the Out-of-Order relaxation. The definition of this relaxation does not immediately appear to have anything to do with ordering, but when instantiated on operations in ordered data structures such as *Dequeue* in queues and *Pop* in stacks, it causes those operations to return an element up to k places out of order. One way to think about this is to imagine that by deleting operation instances in the past, we are making the current instance act as if it is in a different place in the permutation of all instances.

Definition 7 (k-Out-of-Order Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 0$, a k-Out-of-Order relaxation of T, called T', is defined as follows:*

1. $OPS(T') = OPS(T)$
2. *A sequence Π of operation instances is legal in T' if for every instance op where $\Pi = \pi.op.\rho$, there is some sequence $u.v.w$, $|v| \leq k$, which is a minimum-length sequence equivalent in T to π, and there exists a sequence x, where*
   (a) *$u.w$ is legal in T and minimum-length among the set of sequences equivalent to it in T;*
   (b) *$u.w.op$ is legal in T; and*
   (c) i. *$u.w.op \equiv x.w$ and $\pi.op \equiv x.v.w$ or*
       ii. *$u.w.op \equiv u.x$ and $\pi.op \equiv u.v.x$.*

Intuitively, an instance op is allowed after some prefix π if some contiguous portion of the prefix can be ignored. The relaxation does not want to consider past actions which have since been undone, such as an overwritten write or removed element, so we replace π with a minimum-length sequence equivalent to it ($u.v.w$). We then delete up to k consecutive mutator instances (v), making $u.w.op$ legal in the base type. Now, $u.w.op$ being legal in T means that $\pi.op$ is legal in a k-Out-of-Order relaxation T' of an ADT T, but we need to specify what effect op had. We do this by saying that the set of sequences legal in T' after $\pi.op$ is the same set as those legal after reinserting the deleted sequence of instances ($x.v.w$ or $u.v.x$, as appropriate). In this and other relaxations, we refer to T, the type from which the relaxation is defined, as the *base type*.

Note that Henzinger et al. [10] defined the k-Out-of-Order relaxation to allow either deleting (as above) or inserting up to k operation instances. We only allow deletions, as some operations could have arbitrary behavior if we allowed arbitrary additions. For example, *Dequeue* on a relaxed queue can return any value x, if we add $Enqueue(x)$ to the beginning of the history. To avoid such problems, we restrict our attention to Out-of-Order with respect to deleting past instances.

The next relaxation we consider is *Lateness*. This name comes because one way to view the relaxed data type is that operations may act as out-of-order, each for any finite relaxation parameter, except that each time an instance does not satisfy the specification of the base type, we increase a lateness counter. That counter can never exceed k, and resets when an instance acts by the specification of the base type. Thus, we can have instances arbitrarily far from the base type's behavior, but are guaranteed that at least one in every k consecutive instances behaves normally. For example, a relaxed *Dequeue* may return and remove any element in the queue, as long as one in every k *Dequeue*s returns the head.

Definition 8 (k-Lateness Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 1$, a k-Lateness relaxation of T, T', is defined as follows:*

1. *$OPS(T') = OPS(T)$*
2. *A sequence Π of operation instances is legal in T' if for every instance op such that $\Pi = \pi.op.\rho$, there exists $l \geq 0$ such that $\pi.op$ is legal by the semantics of an l-Out-of-Order relaxed T, and at least one in every k consecutive mutator instances in $\Pi|_m$ must have $l = 0$.*

Finally, we consider a relaxation with a different flavor. Instead of allowing operations to act slightly incorrectly, this relaxation allows some mutator instances to have no effect on the state of the shared object. That is, some mutators may "stutter" on the current object state, failing to change it. Here, we only require that some fraction of mutator instances successfully change the object, while others may fail to take effect. All instances must still return a value that is legal based on the current state of the object. To do this, we track the subsequence of mutator instances in the schedule that do not stutter. This subsequence, represented by π'_i, is the history that determines the next operation instance's behavior. For example, a stuttering counter may hold the same value after up to k consecutive *increment*() instances before increasing. Here, π'_i consists only of those *increment* instances which actually increased the counter's value.

Definition 9 (k-Stuttering Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 1$, a k-Stuttering relaxation of T, T' is defined as follows:*

1. *$OPS(T') = OPS(T)$.*
2. *A sequence $\Pi = op_1.op_2...$ of operation instances is legal in T' if for every op_i, with $\Pi = \pi_i.op_i.\rho_i$, op_i returns a value such that $\pi'_i.op_i$ is legal in T, where π'_i is a sequence of mutator instances such that:*
   - *(a) $\pi'_1 = \varepsilon$, the empty sequence;*
   - *(b) $\pi'_i \in \{\pi'_{i-1}, \pi'_{i-1}.op_{i-1}\}$ for $i > 1$; and*
   - *(c) π'_i includes at least one of every k consecutive mutators in π_i.*

4. Converting Relaxations to Consistency Conditions

Relaxing data types and weakening consistency conditions have so far been largely separate methods of improving the performance of shared data types. In the next two sections, we show by example that some relaxed data types under Linearizability can be equivalently defined as their base types under weaker consistency conditions and vice versa.

The basic idea is to think of both consistency conditions and relaxations as functions. A consistency condition reduces concurrent schedules to one or more sequences of operation instances, which can be compared to the legal sequences of a given data type. We can view this as a function from the space of possible concurrent schedules to the power set of possible operation instance sequences. A data type relaxation takes a sequence of operation instances and transforms it to a sequence legal in the base type. This is a function from the space of possible operation instance sequences to itself. Since the codomain of consistency conditions is sets of elements of the domain of relaxations, we can compose the two "functions". The consistency condition can map a concurrent schedule to sequences that may not be legal by the base type, but then we may transform them by the rules of a relaxation to be legal. Thus, both collapsing concurrency and allowing some variance from the base set of legal sequences can occur in the consistency condition.

Similarly, if a consistency condition requires a global ordering respecting the schedule order, then adds other conditions, we show in Section 5 that we can split these conditions apart to have Linearizability for the consistency condition and a relaxation of the original data type, while still allowing the same set of concurrent schedules.

We will start by defining several consistency conditions which are equivalent to the data type relaxations introduced in Section 3. For each, the set of linearizable schedules legal for the relaxed version of a data type is equal to the set of schedules legal for the original data type and the weaker consistency condition. First, we discuss the Out-of-Order relaxation. This enables operations to return values which are not legal by the specification of the base type T, but would be legal if a few other instances had not occurred.

Definition 10 (OutofOrderCC(k))**.** *A schedule of any ADT T satisfies OutofOrderCC(k), for an integer $k \geq 0$, if*

1. *There exists a permutation Π of all operation instances in the schedule, which respects the schedule order of non-overlapping instances.*
2. *For every $op \in \Pi$, with $\Pi = \pi.op.\rho$, there is some sequence $u.v.w$, $|v| \leq k$, which is a minimum-length sequence equivalent in T to π, and there exists a sequence x, such that:*
   - *(a)* *$u.w$ is legal in T and minimum-length among the set of sequences equivalent to it in T;*
   - *(b)* *$u.w.op$ is legal in T; and*
   - *(c)*
     - *i.* *$u.w.op \equiv x.w$ and $\pi.op \equiv x.v.w$, or*
     - *ii.* *$u.w.op \equiv u.x$ and $\pi.op \equiv u.v.x$.*

Theorem 1. *For any integer $k \geq 0$, the set of schedules legal on a k-Out-of-Order relaxation of any ADT T under Linearizability is the same as the set of schedules legal on T under OutofOrderCC(k).*

We can similarly define consistency conditions LatenessCC(k) and StutteringCC(k) equivalent to k-Lateness and k-Stuttering relaxed versions of a type T under Linearizability. By rolling the relaxation into the consistency condition, it follows that the schedules legal on these relaxed data types under Linearizability are those legal on the base type under a weaker consistency condition. Theorems 1, 2, and 3 all hold by construction.

Definition 11 (LatenessCC(k))**.** *A schedule of any ADT T satisfies LatenessCC(k), for an integer $k \geq 1$, if*

1. *There exists a permutation Π of all operation instances in the schedule which respects the schedule order of non-overlapping instances.*
2. *For every $op \in \Pi$, with $\Pi = \pi.op.\rho$, there exists $l \geq 0$ such that $\pi.op$ is legal by the semantics of an l-Out-of-Order relaxed T, and at least one in every k consecutive mutator instances in Π must have $l = 0$.*

Theorem 2. *For any integer $k \geq 1$, the set of schedules legal on a k-Lateness relaxation of any ADT T under Linearizability is the same as the set of schedules legal on T under LatenessCC(k).*

Definition 12 (StutteringCC(k))**.** *A schedule of any ADT T satisfies StutteringCC(k), for an integer $k \geq 1$, if*

1. *There exists a permutation $\Pi = op_1.op_2...$ of all operation instances in the schedule, respecting the schedule order of non-overlapping instances.*
2. *For every instance op_i in Π, let $\Pi = \pi_i.op_i.\rho_i$. op_i returns a value that such that $\pi'_i.op_i$ is legal in T, where π'_i is a sequence of mutator instances such that:*
   - *(a)* *$\pi'_1 = \varepsilon$;*
   - *(b)* *$\pi'_i \in \{\pi'_{i-1}, \pi'_{i-1}.op_{i-1}\}$, for $i > 1$; and*
   - *(c)* *π'_i includes at least one of every k consecutive mutators in π_i*

Theorem 3. *For any integer $k \geq 1$, the set of schedules legal on a k-Stuttering relaxation of any ADT T under Linearizability is the same as the set of schedules legal on T under StutteringCC(k).*

5. Consistency Condition to Relaxation

Thus far, we have shown that we can convert familiar relaxations to consistency conditions. The interest in relaxed data types is largely founded on their ease of use and understanding, compared to consistency conditions. Ideally, then, any consistency condition would be representable as a relaxed data type. This does not seem to be true, at least for our current understanding of relaxed data types, as relaxed data type specifications are sequential, while consistency conditions may be inherently concurrent, either with certain operations only available to certain processes, or by allowing different behavior in the presence of concurrency. Sequential specifications do not have any notion of processes or concurrency, so such conditions cannot be represented as a sequential relaxation.

For example, sequential consistency requires that there exist a permutation of all operation instances that is legal, and in which all instances invoked at a particular process appear in the order in which they were invoked. Because a sequential specification does not know about multiple processes, it is not well-defined for one to require or guarantee that all instances invoked at a single process have some desired relation.

Despite this conclusion that the sets of relaxations and consistency conditions are not equivalent, in this section we will show that at least one known consistency condition can be equivalently expressed as relaxed data types. We consider a well-established consistency condition from the literature, and define a generic data type relaxation equivalent to it.

k-Atomicity

Aiyer et al. defined k-Atomicity [25]. However, their definition only discusses registers and has, to our knowledge, not been generalized to other types. Since we are interested in arbitrary ADTs, we would like a more general definition. To do this, we generalize *Read*s to all pure accessors and *Write*s to all pure mutators. It is not well-defined how mixed operations should behave under k-Atomicity. They should be allowed to return a value as if they were out of order, but then the mutations they cause could seemingly cause previous operation instances to be illegal. Given these issues, we will limit our definition of k-Atomicity to data types which have only pure operations.

Definition 13 (k-Atomicity)**.** *A schedule E on a data type T with only pure operations is* k-atomic, *for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that, for every accessor instance op, with $\Pi = \pi.op.\rho$, there exists a sequence π' obtained by removing up to k consecutive instances from the end of $\pi|_m$ such that $\pi'.op$ is legal in T.*

We can now split this condition into two pieces. The first is the core of Linearizability, that there is an ordering of all operation instances in the schedule that respects the schedule order. The second condition expands the set of legal sequences beyond the set of legal sequences specified by T. The consistency condition requires that the sequence of all instances from the first part is in the set defined by the second part. By moving the second part into the data type, relaxing the data type specification, we are left with Linearizability for the consistency condition, and have the desired equivalence.

Definition 14 (k-Atomic-Equiv Relaxed ADT)**.** *Given any ADT T with no mixed operations and $k \geq 0$, a k-Atomic-Equiv relaxation of T is defined as follows:*

1. *$OPS(T') = OPS(T)$.*
2. *$\mathcal{L}_{T'}$ is the set of sequences Π, where for each accessor instance op, with $\Pi = \pi.op.\rho$, there exists a sequence π' such that $\pi'.op$ is legal in T, where π' is obtained by removing up to k consecutive instances from the end of $\pi|_m$.*

Theorem 4. *For $k \geq 0$, the set of schedules legal on a k-Atomic-Equiv relaxation of any ADT T with no mixed operations under Linearizability is the same as the set of schedules legal on T under k-Atomicity.*

Theorem 4 holds by construction.

Definition 14 is very similar to that of k-Out-of-Order, but they are not equivalent. Because it uses minimal equivalent sequences, a k-Out-of-Order relaxed data type cannot return a value which has been "deleted" from the data structure. For example, consider the following sequence:

$$Enqueue(1).Enqueue(2).Enqueue(3).Dequeue(1).Dequeue(x)$$

In a 2-Out-of-Order queue, x could be either 2 or 3. On the other hand, a k-Atomic type can return historical values that have been deleted or overwritten, so if the sequence in the previous example were executed on a 2-Atomic-Equiv queue, x could also be 1.

It is interesting to note that k-Regularity and k-Safety, other conditions from [25] very similar to k-Atomicity which we define below, cannot be directly converted into relaxed data types. This is because they allow operation instances to have different behaviors when they overlap with one or more mutators than when they do not overlap with any mutators. A sequential specification has no notion of concurrency, or overlapping operation instances, so cannot differentiate these two possibilities. Recent work (e.g., [24,26,27]) has begun exploring the concept of tasks or objects which do not have sequential specifications. These more general definitions may be able to represent consistency conditions which sequential specifications cannot.

6. Placing New Consistency Conditions

We have shown that some data type relaxations can be expressed as consistency conditions. We would like to know how these conditions compare to known consistency conditions. They neither appear to be equivalent to any common consistency conditions, nor do any of our new consistency conditions appear to be related to each other. In this section, we prove that these intuitions are correct.

Recall that consistency conditions are just sets of legal schedules [7]. Thus, to compare the strength of different consistency conditions, we can compare the sets of schedules over all data types.

Definition 15. *Given two consistency conditions C and D, we say that C is* stronger *than D, and D is* weaker *than C, if for all data types T, every schedule legal under C and T is also legal under D and T. That is, the set of legal schedules under C, for all data types, is a subset of the set of schedules legal under D.*

If neither C is stronger than D nor D is stronger than C, we say C and D are incomparable*. If C is stronger than, but not equal to, D, we say that C is* strictly stronger *than D and D is* strictly weaker *than C.*

Our conditions are in the "version staleness-based" family of consistency conditions in [7], referring to the fact that they may return a stale version of the data which is missing some recent updates, since these also have the requirements of Linearizability. Thus, we compare them to k-Atomicity, k-Regularity, and k-Safety, which are also version staleness-based. It is trivial to show that all of our conditions are weaker than Linearizability, since they start with the conditions of Linearizability, then allow some sequences that Linearizability does not.

First, we define generalized versions of k-Regularity and k-Safety, as we did for k-Atomicity. Because k-Regularity and k-Safety may behave exactly as k-Atomicity, we have the same restriction to data types without mixed operations.

Definition 16 (k-Regularity)**.** *A schedule E on a data type T with no mixed operations is k-regular, for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that for every instance op, $\Pi = \pi.op.\rho$,*

1. *if op is a mutator or overlaps with no mutator instances, $\pi|_m.op$ is legal by k-Atomicity; and*
2. *if op is an accessor overlapping with at least one other mutator, there exists a sequence π' such that $\pi'.op$ is legal in T, where π' is constructed either by deleting up to k instances from the end of $\pi|_m$ or by moving*

any subset of the mutator instances overlapping with op from after op in Π to before it and placing them in some order.

Definition 17 (k-Safety)**.** *A schedule E on a data type T with no mixed operations is k-safe, for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that for every instance op, $\Pi = \pi.op.\rho$,*

1. *if op is a mutator or overlaps with no mutator instances, $\pi|_m.op$ is legal by k-Atomicity; and*
2. *if op is an accessor overlapping with at least one other mutator, it may return any value in $rets(OP)$.*

First, we state the following theorem relating k-Atomicity, k-Regularity, and k-Safety. This theorem is well established in the literature for registers, and directly generalizes for our new definitions. The proof is by definition, showing each is a strict subset of the previous.

Theorem 5 ([7,25,28,29])**.** *For all $k \geq 0$, k-Safety is strictly weaker than k-Regularity which is strictly weaker than k-Atomicity, which is strictly weaker than Linearizability, in the domain of data types with no mixed operations.*

Theorem 5 claims the following two statements for each pair of consistency conditions C and D, with C claimed strictly weaker than D: First, for every data type T for which D is defined, every schedule legal under D and T is legal under C and T. Second, there is some data type S for which there is a schedule legal under C and S but not under D and S. The proof follows immediately from the definitions, since linearizable behavior is legal under k-Atomicity, k-atomic behavior is legal under k-Regularity, and k-regular behavior is legal under k-Safety.

We next show that none of the three new consistency conditions we have introduced are comparable to any of these three previously known conditions. The naive approach to show that one new consistency condition is incomparable to these three known conditions would require six individual proofs, showing that the new condition neither contains nor is contained by each of the three. We leverage Theorem 5 to reduce this to two steps for each of our consistency conditions:

1. If we can show that a consistency condition C does not contain (is not weaker than) k-Atomicity, then we immediately know that C is not weaker than either k-Regularity or k-Safety. This follows from the fact that, whatever element in k-Atomicity is not in C is also in the supersets k-Regularity and k-Safety, showing that they are not contained in C.
2. Conversely, if k-Safety does not contain C, then neither k-Regularity nor k-Atomicity can contain C. This is because k-Regularity and k-Atomicity are subsets of k-Safety, and thus also do not contain any element which is not in k-Safety. Since k-Safety does not contain C, such an element must exist in C. Thus, we know that C is not stronger than any of the three.

Thus, by Theorem 5, to show a consistency condition C is incomparable with all of k-Atomicity, k-Regularity, and k-Safety, we choose a data type T and give a schedule which is legal on T under k-Atomicity, but not on T under C, and a data type T' and give a schedule which is legal on T' under C but not on T' under k-Safety. The proof of Theorem 6 uses this structure.

Theorem 6. *In the domain of data types which do not have mixed operations,*

1. *For all $k, l \geq 1$, $OutofOrderCC(k)$ is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*
2. *For all $k \geq 2$ and $l \geq 1$, $LatenessCC(k)$ is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*
3. *For all $k \geq 2$ and $l \geq 1$, $StutteringCC(k)$ is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*

Proof. Throughout this proof, when we use instances of *Enqueue* and *Peek*, we are referring to a restricted FIFO queue data type, which has no *Dequeue*, since that is a mixed operation. *Enqueue* is a pure mutator, since it has no return value, and *Peek* is a pure accessor, since it does not change the shared object.

1. OutofOrderCC(k):

   - To show that OutofOrderCC(k) does not contain l-Atomicity, consider the following sequential schedule of a register:

     $$Write(1).Write(2).Read(1)$$

     For every $l \geq 1$, this schedule is legal under l-Atomicity, since the *Read* can ignore the presence of the last preceding mutator instance, the $Write(2)$. This schedule is not legal under OutofOrderCC(k), for any $k \geq 1$, as the minimal-length equivalent sequence to $Write(1).Write(2)$ is simple $Write(2)$, and $Read(1)$ is not legal after any sequence obtained by deleting instances from this.
   - To show that OutofOrderCC(k) is not contained in l-Safety, consider the following sequential schedule of a restricted FIFO queue:

     $$Enqueue(1).Enqueue(2).Peek(2)$$

     This is legal under OutofOrderCC(k), for every $k \geq 1$, as the prefix $Enqueue(1).Enqueue(2)$ is a minimal-length sequence equivalent to itself, and $Peek(2)$ is legal after the sequence $Enqueue(2)$ obtained by removing one mutator instance. This schedule is not legal under l-Safety, for any $l \geq 1$, since none of the instances are concurrent, and $Peek(2)$ is not legal after any sequence obtained by deleting consecutive mutators from the end of the preceding sequence. Thus, OutofOrderCC(k) is not a subset of l-Safety and is thus not stronger than any of l-Safety, l-Regularity, and l-Atomicity.

2. LatenessCC(k):

   - Consider this sequential schedule of a register:

     $$Write(1).Write(2).Read(1)$$

     This schedule is legal under l-Atomicity, for $l \geq 1$, since the first two instances are legal in a register, and the $Read(1)$ is legal after the sequence obtained by ignoring the last previous mutator. This schedule is not legal under LatenessCC(k), $k \geq 2$, since the minimal-length equivalent sequence of $Write(1).Write(2)$ is $Write(2)$, so $Read(1)$ is not legal after any sequence obtained by deleting instances from a minimal sequence equivalent to the sequence of preceding instances.
   - Consider the following sequential schedule of a restricted FIFO queue:

     $$Enqueue(1).Enqueue(2).Peek(2)$$

     This schedule is legal under LatenessCC(k), for $k \geq 2$, since $Enqueue(1).Enqueue(2)$ is legal in a FIFO queue and $Peek(2)$ is legal after the sequence $Enqueue(2)$ obtained by removing one instance, which is allowed because $Enqueue(1).Enqueue(2)$ is a minimal-length equivalent sequence of itself. This schedule is not legal under l-Safety, for any $l \geq 1$, because $Peek(2)$ is not concurrent with any mutator and not legal after any sequence obtained by deleting instances from the end of $Enqueue(1).Enqueue(2)$.

3. StutteringCC(k):

   - Consider the following sequential schedule of a register:

     $$Write(1).Write(2).Read(1).Read(2)$$

     This schedule is legal under l-Atomicity, $l \geq 1$, since the first *Read* instance may ignore the last previous mutator, while the second *Read* may see it. For $k \geq 2$, this schedule is not legal under StutteringCC(k), as the first *Read* may only return 1 if $Write(2)$ stuttered, but then no succeeding *Read* can see the $Write(2)$.
   - Consider the following sequential schedule of a restricted FIFO queue:

     $$Enqueue(1).Enqueue(2)...Enqueue(k-1).Enqueue(k)...Enqueue(k+l).Peek(k)$$

     This schedule is legal under StutteringCC(k), $k \geq 2$ (Recall that StutteringCC(1) is merely Linearizability), since the first $k-1$ *Enqueue* instances may stutter, leaving $Peek(k)$ legal after the prefix $Enqueue(k)...Enqueue(k+l)$. This schedule is not legal under l-Safety, $l \geq 1$, since the *Peek* is not concurrent with any mutator and ignoring up to l of the last previous mutators will not allow *Peek* to return any value besides 1.

□

While our new consistency conditions are all incomparable to these similar existing conditions in general, we observe that for some specific data types, they may not actually be distinct. We next show that, for a certain class of data types, StutteringCC(k) is stronger than k-Atomicity. We actually show that StutteringCC(k) is stronger than $(k-1)$-Atomicity, a special case of, and thus stronger than, k-Atomicity. This class of types contains those where all mutators are *overwriters*. An overwriter OP is an operation such that every sequence $\pi.op$, $op \in OP$, is equivalent to the singleton sequence op [5,30]. This means that the set of next operation instances which result in a legal sequence is determined entirely by the last previous mutator. In addition to a *Read*/*Write* register, this class includes other data types whose mutators are all overwriters, but which have accessors that return only parts of the state.

StutteringCC(k) and k-Atomicity both allow us to ignore some recent mutator instances. The difference, which makes the two consistency conditions distinct, is that a stuttering instance must be ignored by all subsequent operation instances, while in k-Atomicity, instances may be ignored by some subsequent instances, but seen by others.

Theorem 7. *If all mutators in a data type T, which has no mixed operations, are overwriters, then for all $k \geq 1$, StutteringCC(k) on T is stronger than $(k-1)$-Atomicity on T.*

Proof. We show that any schedule which is legal under StutteringCC(k) is also legal under $(k-1)$-Atomicity. Consider an arbitrary schedule E. Let Π be an ordering of all instances in E, which respects the schedule partial order of non-overlapping instances, as specified by the definition of StutteringCC(k). Let $\Pi = op_1.op_2....$ For each π'_i specified by the definition of StutteringCC(k), let m_i be the last mutator instance in π'_i. Because all mutators are overwriters, $\pi'_i \equiv m_i$. For each $op_i \in \Pi$, there cannot be more than $(k-1)$ mutator instances in Π strictly between m_i and op_i, by the definition of π'_i and m_i. Thus, by deleting up to $(k-1)$ of the last previous mutator instances before op_i in π, m_i will be the last mutator instance, and because it is a mutator, $op_1...m_i \equiv m_i \equiv \pi'_i$, so $op_1...op_i$ is legal under k-Atomicity. Thus, Π is legal under k-Atomicity. □

Finally, we show that the three new consistency conditions corresponding to data type relaxations we introduced in this paper are incomparable to one another. We no longer restrict the set of data types considered, since these relaxations are defined for all data types.

Theorem 8. *Considered on all data types and for all $k \geq 1$ and $l, m \geq 2$, OutofOrderCC(k), LatenessCC(l), and StutteringCC(m) are all incomparable to one another.*

Proof. 1. First, we compare OutofOrderCC(k) and LatenessCC(l), showing that neither condition contains the other.

- On a FIFO queue, the sequential schedule

$$Enqueue(1)...Enqueue(l+2).Dequeue(2)...Dequeue(l+2)$$

is legal under OutofOrderCC(k), because $Enqueue(1)...Enqueue(l+2)$ is legal in the base type, and for each instance $Dequeue(x)$, we can obtain a π' such that $\pi'.Dequeue(x)$ is legal by deleting $Enqueue(1)$ from the minimal-length equivalent sequence to the preceding sequence, which consists of all *Enqueue* instances whose argument has not yet been returned. The schedule is not legal under LatenessCC(l) because there are l consecutive $Dequeue(x)$ instances, for none of which is $\pi'.Dequeue(x)$ legal in a FIFO queue, when π' is a minimal-length equivalent sequence to the prior history, since all minimum-length sequences equivalent to a prefix of this schedule start with $Enqueue(1)$.
- On a FIFO queue, the sequential schedule

$$Enqueue(1)...Enqueue(k+2).Dequeue(k+2)$$

is legal under LatenessCC(l), but not under OutofOrderCC(k). In the base type, the prefix $Enqueue(1)...Enqueue(k+2)$ is legal and of minimum length among equivalent sequences. By deleting a finite number, $k+1$, of consecutive mutators from the preceding sequence, we have $Enqueue(k+2).Dequeue(k+2)$ which is legal in the base type. Under OutofOrderCC(k), the schedule is not legal, because deleting up to k consecutive mutators from the prefix before the *Dequeue* instance yields a sequence ending in $Enqueue(x).Enqueue(k+2)$, where $1 \leq x \leq k+1$, and appending $Dequeue(k+2)$ to such a sequence cannot give a sequence legal in a FIFO queue.

2. Next, we compare OutofOrderCC(k) and StutteringCC(m):

- Consider the following sequential schedule of a FIFO queue:

$$Enqueue(1).Enqueue(2).Dequeue(2).Dequeue(1)$$

This schedule is legal under OutofOrderCC(k), since removing $Enqueue(1)$, which is already minimal-length, from the preceding sequence gives $Enqueue(2).Dequeue(2)$, which is legal in a FIFO queue. The prefix of the first three instances is then equivalent to $Enqueue(1)$, and $Enqueue(1).Dequeue(1)$ is legal, so the entire sequence is legal. This schedule is not legal under StutteringCC(m), because π' for $Dequeue(2)$ must not include $Enqueue(1)$, so no later π' may include $Enqueue(1)$ and 1 cannot be returned by a *Dequeue*. In other words, $Enqueue(1)$ stutters, having no effect, so the second *Dequeue* instance cannot return 1.
- Consider the following sequential schedule of a FIFO queue:

$$Enqueue(1).Dequeue(1).Dequeue(1)$$

This schedule is legal under StutteringCC(m), but not under OutofOrderCC(k). In StutteringCC(m), π' for the first *Dequeue* instance is $Enqueue(1)$, leaving out no previous instances. This *Dequeue* instance stutters, having no effect, so π' for the second *Dequeue* is $Enqueue(1)$, and thus $\pi'.Dequeue(1)$ is legal. For OutofOrderCC(k), the empty sequence ε is the minimal-length equivalent sequence of $Enqueue(1).Dequeue(1)$ and, since $\varepsilon.Dequeue(1)$ is not legal in a FIFO queue, the original schedule is not legal.

3. Finally, compare LatenessCC(l) and StutteringCC(m):

    - On a FIFO queue, the sequential schedule

    $$Enqueue(1)...Enqueue(m+2).Dequeue(m+2)$$

    is legal under LatenessCC(l) but not under StutteringCC(m). For StutteringCC(m), the π' for the *Dequeue* instance must contain at least one of every m consecutive mutator instances in the preceding sequence. This means that it must contain at least 1 $Enqueue(x)$, where $x < m+2$, since there are $m+1$ consecutive such instances. Thus, $\pi'.Dequeue(m+2)$ is not legal, so this schedule is not legal under StutteringCC(m).
    - The sequential schedule

    $$Enqueue(1).Dequeue(1).Dequeue(1)$$

    on a FIFO queue is legal under StutteringCC(m), as argued above, but is not legal under LatenessCC(l). The second *Dequeue* instance must be legal after deleting a minimal-length sequence equivalent to the prefix $Enqueue(1).Dequeue(1)$. That is the empty sequence, though, and $\varepsilon.Dequeue(1)$ is not legal.

In each case, we have shown that the sets of schedules legal under each pair of consistency conditions are not related by subset or superset, so the three consistency conditions are pairwise incomparable. □

7. Application: Hybrid Consistencies

One application of this work is in hybrid consistency conditions. Hybrid consistency conditions are formed by placing the requirements of different consistency conditions on different operations of an ADT, often by requiring "strong" behavior from some operations but only "weak" behavior from others [31]. In general, hybrid consistencies allow implementations where some operations, whose behavior is perhaps less critical, run faster, while we can require some operations to behave more strictly, even if this reduces their performance. When comprised of consistency conditions which can be expressed as relaxations, these specifications of different per-operation behaviors are very natural to express in the data type specification. Such per-operation relaxations have been used in several works in the literature (e.g., Talmage and Welch [6], Henzinger et al. [10], Wimmer et al. [14], Shavit and Taubenfeld [15]), although not explicitly described as hybrid consistency conditions. As a sequential description, which is potentially more intuitive, moving to data type relaxations could greatly reduce the complexity of programming with hybrid consistencies.

We will briefly describe two different data types under hybrid consistency conditions and show how they are naturally expressed as data types with only some operations relaxed.

First, consider a read/write register where some *Write* instances are strong and must respect each other's real-time order, but some are weak and may be reordered among other weak operations. As a hybrid consistency condition, this could be expressed with a flag on the *Write* operation, distinguishing $Write_s$ from $Write_w$, which gives the following consistency condition:

Definition 18 (Consistency Condition for Register with Strong and Weak *Writes*)**.** *A concurrent schedule is legal if there is a permutation* Π *of all instances in the schedule such that*

1. *Each Read instance returns the argument of the previous Write instance in* Π *or* $\perp$ *if there is none.*
2. *If* op_1 *and* op_2 *are instances in* Π *where* op_1 *precedes* op_2 *in the schedule order and either at least one is a* $Write_s$ *instance or both are Read instances, then* op_1 *precedes* op_2 *in* Π.

As in Section 5, we can equivalently express this consistency condition as the following relaxation of a register. This is not technically a relaxation of a simple *Read*/*Write* register, since we need two different *Write* operations, but arguably the differentiation between types of *Write* instances in the consistency condition is already expressing two different operations. The equivalence holds by construction.

Definition 19 (Register with Strong and Weak *Write*s)**.**

1. $OPS = \{Read, Write_s, Write_w\}$
2. *$\mathcal{L}$ is the set of all sequences of instances of operations in OPS such that each Read instance rop returns the argument of any Write instance wop ($Write_s$ or $Write_w$) starting with the latest preceding $Write_s$ (or $\perp$ if there is no preceding $Write_s$) up to and not including the next $Write_s$, such that no previous Read returned the argument of a Write instance later than wop.*

Another example, which does not add new operations to the data type, is a FIFO queue where the *Enqueue* and *Dequeue* operations are strict, so that removing elements exactly follows insertion order, but the *Peek* operation is relaxed to speed up attempts to just read the structure.

Definition 20 (Hybrid Consistency for Queues with Weak *Peek*)**.** *A concurrent schedule is legal if there is a permutation Π of all instances in the schedule which respects the schedule partial order of non-overlapping instances where*

1. *$\Pi|_{\{Enqueue,Dequeue\}}$ is legal by the specification of a FIFO queue and*
2. *Each Peek instance op with $\Pi = \pi.op.\rho$ returns a value such that $\pi'.op$ is legal, where π' is obtained by deleting a contiguous sequence of up to k instances from a minimal-length sequence equivalent to π. That is, op returns one of the k oldest elements currently in the queue and may return $\perp$ if there are fewer than k elements currently in the queue.*

Exactly as with the Out-of-Order consistency condition, we can express this as a relaxation, but, in this case, we only need to relax the *Peek* operation, not all operations.

Definition 21 (Augmented FIFO Queue with Relaxed *Peek*)**.**

1. $OPS = \{Enqueue, Dequeue, Peek\}$
2. *$\mathcal{L}$ is the set of all sequences of instances of operations in OPS such that each Dequeue instance returns the argument of the first Enqueue instance whose argument has not previously been returned by a Dequeue instance or $\perp$ if none exists. Each Peek instance returns the argument of one of the first k Enqueue instances whose arguments have not been returned by a previous Dequeue instance or possibly $\perp$ if there are fewer than k such Enqueue instances.*

By construction, this relaxation is equivalent, under Linearizability, to an unrelaxed queue under the previous consistency condition.

In both of these cases, hybrid consistency is arguably attempting to express something that is probably more naturally expressed as a relaxation of certain operations. In fact, the queue with relaxed *Peek* has appeared in the relaxed data type literature [15,16] already. It seems intuitive that per-operation conditions like these hybrid consistencies are easier as relaxations of operations of the underlying data type. We are still limited in expressing them as such by the fact that we cannot currently express all consistency conditions as relaxations, but this provides both incentive and possible insight for extending the equivalence between the two models.

8. Conclusions and Future Work

In exploring the relation between relaxations for abstract data types and consistency conditions, we have shown that, in several cases, the ideas in each may be expressed equivalently by the other.

Specifically, we showed that the k-Out-of-Order, k-Lateness, and k-Stuttering relaxations may be equivalently expressed as consistency conditions and that the consistency condition k-Atomicity can be equivalently expressed as a relaxation. For each of these, we define the equivalent consistency condition or relaxation. We then explore how the newly-defined consistency conditions fit into the space of consistency conditions, related by the conditions' strength, by showing that they are distinct from several previously-known similar conditions.

In the future, we need to define or quantify the spaces of possible data type relaxations and consistency conditions. This would allow more general conclusions about the relation of the two fields. For example, it seems that every data type relaxation can be expressed as a consistency condition, while only some consistency conditions can be expressed as relaxations. If we could define the space of possible relaxations, we could formally show that the space of relaxations would be a subset of the space of consistency conditions. There is also the question of relaxing tasks and other distributed problems and data operations which cannot be sequentially specified. Such relaxations are not yet defined, but could lead to broader equivalences with consistency conditions.

In this paper, we did not consider relaxing particular operations in a data type. It is possible, and common in the literature [6,10], to relax the behavior of certain operations, while requiring that others behave as in the base type. In the case of per-operation relaxations, our result in Section 6 regarding data types where all mutators are overwriters would extend to all data types where all overwriting operations were k-Stuttering relaxed, greatly increasing their scope.

Finally, it is not obvious how the complexity of implementations of shared data types would depend on which of the two approaches we use. Past work has improved efficiency by relaxing data types, so this work may enable us to more easily compare the complexity of consistency conditions, known and new, perhaps enabling us to distinguish incomparable conditions by performance.

Author Contributions: E.T. and J.L.W. developed the research questions and content together. E.T. led the writing effort, with both authors involved in final preparation.

Acknowledgments: This work was partially supported by NSF Grant 1526725. We would like to thank the editors and anonymous reviewers for their insightful and detailed suggestions and feedback.

Conflicts of Interest: The authors declare no conflict of interest. The funding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

References

1. Herlihy, M.; Wing, J.M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* **1990**, *12*, 463–492. [CrossRef]
2. Attiya, H.; Guerraoui, R.; Hendler, D.; Kuznetsov, P.; Michael, M.M.; Vechev, M.T. 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, 26–28 January 2011; Ball, T., Sagiv, M., Eds.; ACM: New York, NY, USA, 2011; pp. 487–498.
3. Attiya, H.; Welch, J.L. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* **1994**, *12*, 91–122. [CrossRef]
4. Lipton, R.J.; Sandberg, J.S. *PRAM: A Scalable Shared Memory*; Technical Report CS-TR-180-88; Department of Computer Science, Princeton University: Princeton, NJ, USA, 1988.
5. Wang, J.; Talmage, E.; Lee, H.; Welch, J.L. Improved Time Bounds for Linearizable Implementations of Abstract Data Types. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Phoenix, AZ, USA, 19–23 May 2014; pp. 691–701.

6. Talmage, E.; Welch, J.L. Improving Average Performance by Relaxing Distributed Data Structures. In Proceedings of the 28th International Symposium on Distributed Computing, DISC 2014, Austin, TX, USA, 12–15 October 2014; Lecture Notes in Computer Science; Kuhn, F., Ed.; Springer: Berlin, Germany, 2014; Volume 8784, pp. 421–438.
7. Viotti, P.; Vukolic, M. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* **2016**, *49*, doi:10.1145/2926965. [CrossRef]
8. Shapiro, M.; Preguiça, N.M.; Baquero, C.; Zawirski, M. Conflict-Free Replicated Data Types. In Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Grenoble, France, 10–12 October 2011; Lecture Notes in Computer Science; Défago, X., Petit, F., Villain, V., Eds.; Springer: Berlin, Germany, 2011; Volume 6976, pp. 386–400.
9. Afek, Y.; Korland, G.; Yanovsky, E. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In Proceedings of the 14th International Conference on Principles of Distributed Systems, OPODIS 2010, Tozeur, Tunisia, 14–17 December 2010; Lecture Notes in Computer Science; Lu, C., Masuzawa, T., Mosbah, M., Eds.; Springer: Berlin, Germany, 2010; Volume 6490, pp. 395–410.
10. Henzinger, T.A.; Kirsch, C.M.; Payer, H.; Sezgin, A.; Sokolova, A. Quantitative relaxation of concurrent data structures. In Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, 23–25 January 2013; Giacobazzi, R., Cousot, R., Eds.; ACM: New York, NY, USA, 2013; pp. 317–328.
11. Alistarh, D.; Kopinsky, J.; Li, J.; Shavit, N. 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, 7–11 February 2015; Cohen, A., Grove, D., Eds.; ACM: New York, NY, USA 2015; pp. 11–20.
12. Kirsch, C.M.; Lippautz, M.; Payer, H. *Fast and Scalable k-FIFO Queues*; Technical Report 2012-04; Department of Computer Sciences, University of Salzburg: Salzburg, Austria, 2012.
13. Rihani, H.; Sanders, P.; Dementiev, R. 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, 13–15 June 2015; Blelloch, G.E., Agrawal, K., Eds.; ACM: New York, NY, USA, 2015; pp. 80–82.
14. Wimmer, M.; Gruber, J.; Träff, J.L.; Tsigas, P. The lock-free k-LSM relaxed priority queue. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, 7–11 February 2015; Cohen, A., Grove, D., Eds.; ACM: New York, NY, USA, 2015, pp. 277–278.
15. Shavit, N.; Taubenfeld, G. The computability of relaxed data structures: Queues and stacks as examples. *Distrib. Comput.* **2016**, *29*, 395–407. [CrossRef]
16. Talmage, E.; Welch, J.L. Anomalies and Similarities Among Consensus Numbers of Relaxed Queues. In Proceedings of the 5th Edition of the International Conference on Networked Systems, NETYS 2017, Marrakech, Morocco, 17–19 May 2017; Lecture Notes in Computer Science; Abbadi, A.E., Garbinato, B., Eds.; Springer: Berlin, Germany, 2017; Volume 10299.
17. Michael, M.M.; Vechev, M.T.; Saraswat, V.A. Idempotent work stealing. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, 14–18 February 2009; Reed, D.A., Sarkar, V., Eds.; ACM: New York, NY, USA, 2009; pp. 45–54.
18. Payer, H.; Röck, H.; Kirsch, C.M.; Sokolova, A. 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, 6–8 June 2011; Gavoille, C., Fraigniaud, P., Eds.; ACM: New York, NY, USA, 2011; pp. 331–332.
19. Bermbach, D.; Kuhlenkamp, J. Consistency in Distributed Storage Systems - An Overview of Models, Metrics and Measurement Approaches. In Proceedings of the First International Conference on Networked Systems, NETYS 2013, Marrakech, Morocco, 2–4 May 2013; Lecture Notes in Computer Science; Gramoli, V., Guerraoui, R., Eds.; Springer: Berlin, Germany, 2013; Volume 7853, pp. 175–189.
20. Friedman, R.; Vitenberg, R.; Chockler, G.V. On the composability of consistency conditions. *Inf. Process. Lett.* **2003**, *86*, 169–176. [CrossRef]

21. Vitenberg, R.; Friedman, R. On the Locality of Consistency Conditions. In Proceedings of the 17th International Conference on Distributed Computing, DISC 2003, Sorrento, Italy, 1–3 October 2003; Lecture Notes in Computer Science; Fich, F.E., Ed.; Springer: Berlin, Germany, 2003; Volume 2848, pp. 92–105.
22. Talmage, E.; Welch, J.L. Relaxed Data Types as Consistency Conditions. In Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Boston, MA, USA, 5–8 November 2017; Lecture Notes in Computer Science; Spirakis, P., Tsigas, P., Eds.; Springer: Berlin, Germany, 2017.
23. Neiger, G. Set-Linearizability. In Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, CA, USA, 14–17 August 1994; Anderson, J.H., Peleg, D., Borowsky, E., Eds.; ACM: New York, NY, USA, 1994; p. 396.
24. Castañeda, A.; Rajsbaum, S.; Raynal, M. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks—(Extended Abstract). In Proceedings of the 29th International Symposium on Distributed Computing, DISC 2015, Tokyo, Japan, 7–9 October 2015; Lecture Notes in Computer Science; Moses, Y., Ed.; Springer: Berlin, Germany, 2015; Volume 9363, pp. 420–435.
25. Aiyer, A.; Alvisi, L.; Bazzi, R.A. On the Availability of Non-strict Quorum Systems. In *Distributed Computing*; Lecture Notes in Computer Science; Fraigniaud, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3724, pp. 48–62.
26. Hemed, N.; Rinetzky, N. Brief announcement: Concurrency-Aware Linearizability. In Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC'14, Paris, France, 15–18 July 2014; Halldórsson, M.M., Dolev, S., Eds.; ACM: New York, NY, USA, 2014; pp. 209–211.
27. Hemed, N.; Rinetzky, N.; Vafeiadis, V. Modular Verification of Concurrency-Aware Linearizability. In Proceedings of the 29th International Symposium on Distributed Computing, DISC 2015, Tokyo, Japan, 7–9 October 2015; Lecture Notes in Computer Science; Moses, Y., Ed.; Springer: Berlin, Germany, 2015; Volume 9363, pp. 371–387.
28. Lamport, L. On Interprocess Communication. Part II: Algorithms. *Distrib. Comput.* **1986**, *1*, 86–101. [CrossRef]
29. Shao, C.; Welch, J.L.; Pierce, E.; Lee, H. Multiwriter Consistency Conditions for Shared Memory Registers. *SIAM J. Comput.* **2011**, *40*, 28–62. [CrossRef]
30. Kosa, M.J. Time Bounds for Strong and Hybrid Consistency for Arbitrary Abstract Data Types. *Chicago J. Theor. Comput. Sci.* **1999**, doi:10.4086/cjtcs.1999.009 [CrossRef]
31. Attiya, H.; Friedman, R. A Correctness Condition for High-Performance Multiprocessors. *SIAM J. Comput.* **1998**, *27*, 1637–1670. [CrossRef]