Computing gets bigger all the time. As we collect more and more data, computational problems are growing extremely quickly. To enable the kind of bigger computing required for these problems, we distribute tasks to computers, which may be specialized, in a variety of different physical locations. This opens up the ability to use vast amounts of processing power on a single problem, while balancing the load of maintenance and upkeep of the computers. However, distributed computation is in many ways more complex than local, and has its own set of unique challenges and foibles.

I am interested in exploring the tools that make distributed programming practical and the bounds of what collaborating computers can do. Particularly, I am interested in the properties of distributed, shared data structures, which are fundamental tools in distributed programming. By abstracting basic tasks of storing, sharing, and interacting with data needed by multiple processes, we can remove the complications of coordinating concurrent operations from a programmer’s burden. I wish to explore the possibilities of distributed data structures, from implementation to specification, speed, and computational power. To do this, I work on developing efficient algorithms for data structures, specifying and designing properties of data structures that are either useful to a user or allow more efficient implementation. Conversely, I explore lower bounds and impossibility results for distributed data. By knowing the limits of what we can do, we know where to look for improvements.

1 Implementing Relaxed Data Types

Relaxed data types add limited non-determinism to the sequential specification of a data type’s behavior. That is, a relaxed data type may have several possible correct return values to a given operation in a particular state. In practice, this allows processes to perform more computation locally and synchronize or coordinate less often. Communication between distinct physical locations is relatively expensive, since it is bounded by the speed of light. By requiring less coordination between remote processes, \[2, 9\] provided some evidence that relaxed data types could improve the performance of shared data structures, in terms of the time between invocation and response of operations on the structure.

In \[17\], we gave distributed, message-passing algorithms implementing two relaxations of FIFO queues, as well as several lower bounds on distributed implementations of a variety of queues. We considered the Out-of-Order \(k\)-relaxation and Restricted Out-of-Order \(k\)-relaxation. \[9\] specified the general forms of these relaxations, and we instantiated their definitions on queues. As our previous work \[21\] shows, Dequeue is the most expensive operation, so we focused on improving its performance. We proved that the worst-case time cost of Dequeue in several relaxations, as well as unrelaxed queues, cannot be reduced below the maximum delay a message between processes may experience. Since these relaxations cannot improve the worst-case time, we go on to show that in unrelaxed queue implementations, all Dequeues are expensive (on the order of a message delay). This yields a high amortized cost. In contrast, our implementations of relaxed queues, though they have similar, or even worse, worst-case cost for Dequeue, have much lower amortized cost than is possible in an unrelaxed queue. Our results complement the shared-memory results of \[9\] with message-passing results showing that there is a clear performance benefit available from relaxing data types. We further show that as relaxation increases, performance continues to increase, with the upper bound for large relaxations below the lower bound for small amounts of relaxation. We thus have a controllable performance gain, trading off precise behavior for increased efficiency.

1.1 Algorithms

We designed algorithms for the out-of-order and restricted out-of-order queues in a partially-synchronous message passing system. This system has known upper and lower bounds on the delay of any message
between two processes. We assume local clocks that run at the same rate as real time, which means that
they can be approximately synchronized, as per [5]. These system assumptions are slightly stronger than
real-world behavior, but algorithms for this model are close to what might be feasible in a real system, and
the extra strength means that our lower bounds apply even to real-world systems with fewer guarantees.

To implement these relaxed queues more efficiently, in an amortized sense, than is possible with an
unrelaxed queue, we wanted to allow some Dequeuees to return without waiting for any communication with
other processes. These relaxations allow us to do this, since several values are legal to return at any given
time (assuming the queue is sufficiently full). The complexity of the algorithm comes from coordinating
to ensure that no two processes remove the same value from the queue concurrently. To do this, we use a
labeling system, where values are labeled for particular processes. While a value is labeled for process p,
it may only be removed by a Dequeue invoked at p. We apply labels at times when we must synchronize
anyway, which allows all processes to agree on labels.

1.2 Lower Bounds
We also give lower bounds on the amortized complexity of Dequeue for unrelaxed and both types of relaxed
queues. First, these show that our algorithms are near optimal, with the out-of-order algorithm’s complex-
ity only a small additive term above the lower bound and the restricted out-of-order algorithm only a factor
of two above its lower bound. Second, we see that our algorithms both have lower complexity for Dequeue
than is possible in an unrelaxed queue, showing that there is inherent benefit from relaxation. Third,
since both the complexity of our algorithms and the lower bounds decrease as the relaxation parameter k
increases, we can show that sufficiently increasing the relaxation parameter allows better performance than
is possible with a less-relaxed version. Thus, we have shown that relaxation has an inherent performance
benefit, and that the more we relax a queue, the better performance we can achieve.

2 Consensus Numbers of Relaxed Queues
Having shown that relaxing shared data types can improve their performance, it is natural to ask what
we have given up to achieve this extra performance. For example, the relaxed queues we have discussed
no longer provide full guarantees on the order in which elements are removed by Dequeue. This is clearly
unacceptable for some applications which demand exact ordering, but others, such as parallel job queues,
may not be adversely affected by slight deviations in order. In [19], we sought to exactly quantify the loss
in computational power due to relaxation. [15] and [16] began exploring this question, but considered only
one relaxation, where we compared several.

2.1 Consensus Numbers
The classic measure of computational strength for distributed data types is the type’s consensus number,
introduced in [10]. The consensus problem is one of the fundamental problems in distributed computing,
requiring several processes which each have a private input to all, in a finite time, agree on one of the
processes’ inputs. In an asynchronous, shared-memory system where processes may fail by crashing, the
strength of the shared types allowed determines how large a set of processes can solve consensus. [10]
shows that if all shared objects are Read/Write registers, then not even two processes can solve consensus.
However, if shared objects can be queues or stacks, then two processes can solve consensus, and if there are
stronger types, such as queues augmented with a Peek operation or Read-Modify-Write registers, then
any number of processes can solve consensus. Once a set of processes can solve consensus, they can use
it to implement any other data type, so this defines a hierarchy, where the consensus number of a type is
defined as the maximum number of processes that can solve consensus using Read/Write registers and a
given type. We used the notion of consensus numbers to compare the strength of several queue relaxations
with different parameters.
To prove consensus numbers of non-deterministic types, we extend the classic techniques, deriving from [8], for proving upper bounds on consensus numbers. These techniques, called bivalency proofs, consider all the possible steps a consensus algorithm can take at certain points in time. With non-deterministic shared data types, we must account for more possible steps, since a given operation may have several possible behaviors. These extra steps are more things to account for in a proof, but also give extra leverage, since we can choose an execution with a particular non-deterministic behavior that shows our result.

2.2 Bounds for the Space of Relaxations

We consider the same two relaxed queues as in [17], and a third relaxation called lateness [9]. For completeness, we consider augmented queues, which have a Peek operation, as well as Enqueue and Dequeue. We also consider relaxing all three operations, instead of just Dequeue. In [17], we only relaxed Dequeue as we were trying to increase performance and Enqueue can have high performance without relaxation. Here, we are interested in relaxed queues’ behavior, so it is interesting to relax all operations. Since we are relaxing all three operations on an augmented queue, we need three distinct relaxation parameters, one for each operation. Each parameter may be a positive integer, infinite, or disabled, indicating that the corresponding operation is not allowed. These three parameters define an infinite three-dimensional space of possible relaxations.

To find the consensus number of each discrete point in this infinite three-dimensional space, we first prove a few lemmas that allow us to extend a known consensus algorithm for a particular point in the parameter space to many other points. One lemma shows that increasing relaxation either weakens a data type or leaves its strength unchanged. This is intuitive, since adding non-determinism should not increase a type’s strength. Also, for a given triple of relaxation parameters, the consensus number of a restricted out-of-order relaxed queue is at least that of either an out-of-order relaxed queue or a lateness relaxed queue, since restricted out-of-order queues satisfy all the conditions of the other two relaxed queues.

With these lemmas, we need only show the consensus numbers of a handful of specific points. We give algorithms and bivalency proofs, using our extensions for non-deterministic types. With these points, we can use the weakening and strengthening lemmas, as well as the relation between relaxations to bound the infinite number of points in the three relaxation spaces. We find that different choices of relaxation and parameters drop consensus number from infinite to at most two very suddenly. This shows that it behooves a developer interested in using one of these relaxed queues to choose their relaxation type and parameters very carefully to maintain the required strength.

3 Relaxed Data Types as Weak Consistency Conditions

In the work discussed so far, we have explored data type relaxation as a means of trading off some behavioral guarantees for increased performance. Researchers have been exploring various methods of optimizing shared data structure implementations for nearly as long as we have had networked computers. One of the main techniques has been to consider different conditions on how the concurrent behavior of a distributed system may relate to the classical, sequential specification of the data type. In this work, we compare this method with relaxation and show that these are two different ways to express the same idea, with relaxation giving an alternate language to describe certain behaviors. Since we now have two different ways to express the same idea, we can use whichever formalism is easiest to use or understand, simplifying implementations and proofs of correctness and efficiency. We can also use any tools developed for one formalism to show results for the other.

3.1 Consistency Conditions

Relaxations weaken the sequential specification of a data type’s behavior. In contrast, consistency conditions control the relation between the concurrent schedule of events in a distributed system with that sequential specification. The strongest consistency condition, which we have implicitly used in everything
discussed so far, is linearizability. Linearizability requires that all process agree on a single order of all operations in a concurrent execution which respects the order of operations that do not overlap in real time and that the sequence given by the order is legal in the specification of the base type. Other, weaker consistency conditions allow more possible concurrent behaviors, such as allowing operations at different processes to appear as if they occurred in a different order than they actually did. It has been shown that weaker consistency conditions can admit more efficient implementations than linearizability (e.g. [4]).

3.2 Converting between Relaxations and Consistency Conditions

We apply both consistency conditions and data type relaxations in comparing a concurrent schedule of a distributed system to the sequential specification of a data type. We can think of them both as functions: consistency conditions map a sequence of steps at each process to one or more sequences of operations on a data type; relaxations map one sequence of data type operations to another. We then check that the sequences resulting from the composition of these functions are legal. Since the combination is just another function, we can split it anywhere we like.

If we incorporate the effect of a data type relaxation into a consistency condition, we can treat the entire package as just a consistency condition. Similarly, if we can split a complex consistency condition into linearizability and some other mapping between sequences of data type operations, then we can convert that consistency condition into a data type relaxation. To formally define this equivalence, we consider the set of concurrent schedules which is considered to be legal on the given data type under a consistency condition and, possibly, a relaxation. When speaking of a relaxation alone, we assume linearizability.

We give examples of both directions of this conversion. We first show that the out-of-order and restricted out-of-order $k$-relaxations mentioned above, as well as another relaxation from [9], called the stuttering $k$-relaxation can be alternately defined as consistency conditions. Conversely, we generalize the consistency condition $k$-Atomicity, defined in [3] for Read/Write registers, to arbitrary data types and show that it can be equivalently expressed as linearizability and a data type relaxation.

3.3 Comparing Consistency Conditions

We next used techniques from the literature on consistency conditions (e.g. [20]) to show that our newly-defined consistency conditions, corresponding to relaxations, are not comparable to similar, known consistency conditions. This means that, even though these relaxations can be expressed as consistency conditions, they are distinct from conditions considered in the consistency condition literature, and thus of independent interest. To show that they are not completely unrelated, though, we found a particular class of data types for which the consistency condition version of the stuttering relaxation is strictly stronger than $k$-Atomicity.

4 Finding Consensus Numbers of Abstract Data Types

In this work [18], we sought to address the issue of determining the computational power, as represented by its consensus number, of an arbitrary given data type. This is an important task for a developer designing a distributed system, as the choice of shared data types has a large effect on the efficiency and power of the whole system. The standard technique for determining a data type’s consensus number is to give a consensus algorithm for a certain number of processes using that type, and then to give an impossibility proof that shows that that is the maximum number of processes which can use the type to solve consensus. This is a relatively large burden on a system developer, so we would like to simplify the process. Ideally, we could provide classifications of all data types that would allow a developer to quickly check the consensus number of a new data type. This general solution is not possible, as determining consensus numbers is undecidable [11]. While we cannot solve the problem in all cases, we were able to give some conditions that may allow a developer to easily determine the consensus number of some data types. Our conditions also formalize and generalize the difference between queues with a Peek operation, which have infinite
consensus number, and stacks with a Peek, which only have consensus number 2, the same as a stack without Peek. Other works, such as [13, 14], also explore this area, identifying different categories of data types whose consensus numbers can be easily recognized.

4.1 Sensitivity

We begin by observing that in a consensus algorithm, processes must share their input values with each other. Thus, the values among which processes must decide will be the arguments to some operations on shared data objects, since this is the only way processes can communicate in a shared memory setting. To learn another processes’ inputs, a process must then receive a return value to an operation on a shared object from which it can deduce that process’ input value. To quantify this ability to learn other processes’ values, we defined the notion of an operation’s sensitivity. An operation is sensitive to a set $S$ of past operations if it can determine the arguments of the operations in $S$. To use this notion, we define several specific sets $S$ and determine the consensus numbers of operations which are $S$-sensitive for those sets.

4.2 Results

The first set $S$ we define is the singleton set containing the $k$th operation in a run which changed the state of the shared object. This is a generalization of Peek on an augmented queue, which can learn the argument of the first Enqueue (in an algorithm which doesn’t use Dequeue). If they can learn the argument of the $k$th operation, any number of processes can solve consensus, since they can all decide that $k$th argument, in an algorithm where all processes use their consensus inputs as arguments. In contrast, if we generalize Peek on a stack (recall that stacks with Peek have consensus number 2), and set $S$ equal to the singleton set containing the $k$th most recent operation which changed the state of the shared object, then we showed that any such data type has consensus number at most 3, or less for certain subclasses of data types. If an operation can learn the arguments of all of the $l$ most recent operations, then we show that it has consensus number either $l$ or $l+1$, depending on other properties of the operation. This allows us to construct a data type with arbitrary consensus number which is a puzzle that has independent interest, since most common data types have consensus numbers in the set $\{1, 2, \infty\}$.

This work took the intuition behind the difference in consensus number between queues and stacks, each augmented with Peek, and used it to find larger classes of data types with specific consensus numbers. Now, if a developer can determine whether a data type is sensitive to one of the sets of past operations we defined, then the type’s consensus number is known without further work.

5 Future Work

5.1 Consistency Conditions vs. Relaxations

As described above, we have begun exploring the relation of consistency conditions and relaxed data types. So far, we have shown that relaxations can be equivalently described as consistency conditions, and some consistency conditions can be expressed as relaxations. We have also taken advantage of tools in the consistency condition literature to compare the strength of data type relaxation definitions to consistency conditions. In future work, I would like to more fully explore the implications and uses of this partial equivalence. It may be possible to generalize the definition of data type relaxations to represent more consistency conditions than the current definition allows. In large part, the difficulty here is finding a way to represent concurrency in a sequential specification. This begins to approach other work (e.g. [6, 7]) on the specification of problems which cannot be represented sequentially. Perhaps a broader definition of relaxation could apply to such problems, as well.

In another direction, an interesting possible method for describing, reasoning about, and implementing weak consistency conditions is to combine relaxations with consistency conditions weaker than linearizability. One benefit of this approach is that it moves part of the complexity of possible behaviors to the
sequential world, which is typically easier to reason about. I intend to explore whether this separation of
definition could open the path to more efficient implementations or easier impossibility and lower bound
proofs for data types. Another possible benefit is that extending relaxations to work with arbitrary con-
sistency conditions could also allow extensions to the equivalence between the two systems. I would like
to explore this more fully. Ideally, we would be able to express a larger portion of the space of consistency
conditions as data type relaxations, and vice versa. It still seems that some aspects of concurrent behavior
would be impossible to express sequentially, but that leads us to the question of where the edges of the
equivalence are. I would like to continue my work in this direction to more fully understand how to express
various behaviors of distributed data structures.

5.2 Practical Implementations of Arbitrary Data Types

In past work [21], we presented an algorithm for an arbitrary data type in a partially-synchronous envi-
ronment. This model approximates a real-world system, but relies on upper bounds on message delays
that may not be realistic. I would like to continue this work in a truly asynchronous message-passing
system, attempting to find an optimal implementation for any data type. Such an implementation should
be easy to port to a real-world system, since it assumes very little about the environment. A general
implementation which can be instantiated to implement an arbitrary data type is often very difficult to
achieve, and may not even be possible. I would thus probably start with implementations of various data
types of interest, with the end goal of generalizing implementations as much as possible.

Another direction to take these implementations is to extend in the direction of relaxation. [21] presents
a general algorithm in a partially synchronous system, but also restricts it to only deterministic operations.
If this latter constraint can be removed, then the general implementation could also be applied to relaxed
and other non-deterministic data types.

5.3 Classifying Operations

To show that a general implementation of abstract data types is optimal or near-optimal, we can start with
the lower bounds from the partially-synchronous environment, since we are weakening assumptions. Those
bounds are not complete, though, covering only operations with certain algebraic properties. An ongoing
task, vital to optimal implementations of arbitrary data types, is to expand the classification ([12, 21], etc.)
of all possible operations on data types, so that we can prove lower bounds on all operations. I would like
to work on identifying more classes of operations and proving lower bounds on their implementation.

Since there are infinitely many possible operations and interesting classes of operations, I would also
like to explore the properties that characterize operations of particular interest in distributed data types.
An obvious example characteristic is that operations must have some effect, either storing or reporting
information. If we can find features that seem necessary for an operation to be of value, then we can
prioritize the search for new operation classifications.

5.4 Applications of Relaxed Data Types

While I have been involved in theoretical work on relaxed data types, there is still a wide field of application
research to explore. Finding applications that benefit from the increased efficiency of relaxed types but
do not suffer too much from the relaxed guarantees is an area that promises many interesting questions.
For example, task allocation for robot swarms has the potential for increased throughput if some tasks
may be multiply assigned, while others are assigned out of order. With proper tuning, it may be possible
to increase total performance, without sacrificing too much from guarantees on priority ordering. Such
problems would be ideal for collaboration with domain experts from a variety fields, who could identify
tasks which do not rely too heavily on deterministic guarantees such as ordering, but which would benefit
from faster distributed execution. This is the ultimate goal of research on relaxed data types, and I am
excited to work with other researchers to solve new and interesting problems.
References


