Contention-sensitive Data Structures and Algorithms*

Gadi Taubenfeld†

March 28, 2017

Abstract

A contention-sensitive data structure is a concurrent data structure in which the overhead introduced by locking is eliminated in common cases, when there is no contention, or when processes with non-interfering operations access it concurrently. When a process invokes an operation on a contention-sensitive data structure, in the absence of contention or interference, the process must be able to complete its operation in a small number of steps and without using locks. Using locks is permitted only when there is interference. We formally define the notion of contention-sensitive data structures, propose four general transformations that facilitate devising such data structures, and illustrate the benefits of the approach by implementing a contention-sensitive consensus algorithm, a contention-sensitive double-ended queue data structure, and a contention-sensitive election algorithm.

Keywords: Contention-sensitive, interference, synchronization, locks, shortcut code, disable-free, prevention-free, livelock, starvation, k-obstruction-free, wait-free.

*A preliminary version of the results presented in this paper, has appeared in *Proceedings of the 23rd International Symposium on Distributed Computing* (DISC 2009), Elche, Spain, September 2009.

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

1 Introduction

1.1 Motivation

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section code, within which the process is guaranteed exclusive access. Any sequential data structure can be easily made concurrent using such a locking approach. The popularity of this approach is largely due to the apparently simple programming model of such locks, and the availability of lock implementations which are reasonably efficient.

When using locks, the *granularity* of synchronization is important. Using a single lock to protect the whole data structure, allowing only one process at a time to access it, is an example of *coarse-grained* synchronization. In contrast, *fine-grained* synchronization enables to lock "small pieces" of a data structure, allowing several processes with non-interfering operations to access it concurrently. Coarse-grained synchronization is easier to program but is less efficient compared to fine-grained synchronization.

Using locks may, in various scenarios, degrade the performance of concurrent applications, as it enforces processes to wait for a lock to be released. Moreover, slow or stopped processes may prevent other processes from ever accessing the data structure. Locks can introduce false conflicts, as different processes with non-interfering operations contend for the same lock, only to end up accessing disjoint data.

A promising approach is the design of concurrent data structures and algorithms which avoid locking. The advantages of such algorithms are that they are not subject to priority inversion, they are resilient to failures, and they do not suffer significant performance degradation from scheduling preemption, page faults or cache misses. On the other hand, such algorithms may impose too much overhead upon the implementation and are often complex.

We propose an intermediate approach for the design of concurrent data structures, which incorporates ideas from the work on data structures which avoid locking. While the approach guarantees the correctness and fairness of a concurrent data structure under all possible scenarios, it is especially efficient in common cases when there is no (or low) contention, or when processes with non-interfering operations access a data structure concurrently.

1.2 Contention-sensitive data structures: The basic idea

Contention for accessing a shared object is usually rare in well designed systems. Contention occurs when multiple processes try to acquire a lock at the same time. Hence, a desired property in a lock implementation is that, in the absence of contention, a process can acquire the lock extremely fast, without unnecessary delays. Furthermore, such fast implementations decrease the possibility that processes which invoke operations on the same data structure in about the same time but not simultaneously, will interfere with each other. However, locks were introduced in the first place to resolve conflicts when there is contention, and acquiring a lock *always* introduces some overhead, even in the cases where there is no contention or interference.

We propose an approach which, in common cases, eliminates the overhead involved in acquiring a lock. The idea is simple: assume that, for a given data structure, it is known that in the absence of contention or interference it takes some fixed number of steps, say at most 10 steps, to complete an operation, not counting the steps involved in acquiring and releasing the lock. According to our approach, when a process invokes an operation on a given data structure, it first tries to complete its operation, by executing a short code, called the *shortcut code*, which does not involve locking. Only if it does not manage to complete the operation fast enough, i.e., within 10 steps, it tries to access the data structure via locking. The shortcut code is required to be *wait-free*. That is, its execution by a process takes only a finite number of steps and always terminates, regardless of the behavior of the other processes.

Using an efficient shortcut code, although eliminates the overhead introduced by locking in common cases, introduces a major problem: we can no longer use a sequential data structure as the basic building block, as done when using the traditional locking approach. The reason is simple, many processes may access the same data structure simultaneously by executing the shortcut code. Furthermore, even when a process acquires the lock, it is no longer guaranteed to have exclusive access, as another process may access the same data structure simultaneously by executing the shortcut code.

Thus, a central question which we are facing is: if a sequential data structure cannot be used as the basic building block for a general technique for constructing a contention-sensitive data structure, then what is the best data structure to use? Before we proceed to discuss formal definitions and general techniques, which will also help us answering the above question, we demonstrate the idea of using a shortcut code to avoid locking – in the absence of synchronization conflicts – by presenting a contention-sensitive solution to the binary consensus problem using atomic read/write registers and a single lock.

1.3 A simple example: Contention-sensitive consensus

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. A consensus algorithm is an algorithm that produces such an agreement. While various decision rules can be considered such as "majority consensus", the problem is interesting even where the decision value is constrained only when all processes are unanimous in their opinions, in which case the decision value must be the common opinion. A consensus algorithm is called *binary* consensus when the number of possible initial opinions is two.

Processes are not required to participate in the algorithm, however, once a process starts participating it is guaranteed that it may fail only while executing the shortcut code. The algorithm uses an array $x[0..1]$ of two atomic bits, and two atomic registers y and *out*. After a process executes a **decide**() statement, it immediately terminates.

CONTENTION-SENSITIVE BINARY CONSENSUS: program for process p_i with input $in_i \in \{0, 1\}$.

shared $x[0..1]$: array of two atomic bits, initially both 0
y, *out* : atomic registers which range over $\{\perp, 0, 1\}$, initially both $\perp$

1 $x[in_i] := 1$ // start shortcut code

2 **if** $y = \perp$ **then** $y := in_i$ **fi**
3 **if** $x[1 - in_i] = 0$ **then** $out := in_i$; **decide**(in_i) **fi**
4 **if** $out \neq \perp$ **then decide**(out) **fi** // end shortcut code
5 $\boxed{\text{lock}}$ **if** $out = \perp$ **then** $out := y$ **fi** $\boxed{\text{unlock}}$; **decide**(out) // locking

When a process runs alone (either before or after a decision is made), it reaches a decision after accessing the shared memory at most five times. Furthermore, when all the concurrently participating processes have the same preference – i.e., when there is no interference – a decision is also reached within five steps and without locking. Two processes with conflicting preferences, which run at the same time, will not resolve the conflict in the shortcut code if both of them find $y = \perp$. In such a case, some process acquires the lock and sets the value of *out* to be the final decision value. The assignment $out := y$ requires two memory references and hence it involves two atomic steps.

1.4 Interference

In the context of physics (wave propagation) or telecommunications (signal disruption) the notion of interference is well understood and defined. In the distributed computing literature, the notion of interference is extensively used but is not formally defined. Below I explain what interference means in the context of distributed computing.

The notion of *interference* between operations, which is commonly used in distributed computing, has different interpretations at the semantic (abstract) level and at the implementation level. At the semantic level, when referring only to the meaning (i.e., definitions) of the operations, (two or more) operations interfere if the execution of one may effect the results of the others. For example, when solving consensus, the operations of participating processes may interfere if they have different preferences. However, if all preferences are the same then, the final result is determined regardless of the implementation. At the implementation level, operations interfere if the execution of one effects in any way (by disrupting or delaying) the execution of the other operations. For example, this may happen when using locks, where a process has to wait for a lock to be released.

Using a lock, any two operations may interfere (at the implementation level) with each other. That is, locks do not distinguish between operations that are semantically non-interfering. Consider, for example, the contention-sensitive consensus from Section 1.3. Assume that there are million processes. When all the concurrently participating processes have the same preference a decision is reached within five read/write steps of each process. On the other hand, when implementing consensus using coarse-grained locking, these million processes may have to go through the lock one at a time.

1.5 Progress conditions

A process executes a sequence of steps as defined by its algorithm. A process executes correctly its algorithm until it (possibly) crashes. After it has crashed a process executes no more steps. Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*. In an asynchronous system there is no way to distinguish between a faulty process and a process that is very slow. We will consider both the case where it is assumed that processes never fail and the case where processes may fail by crashing.

Several progress conditions have been proposed for data structures which avoid locking, and in which processes may fail by crashing. *Wait-freedom* guarantees that *every* active process will always be able to complete its pending operations in a finite number of steps [14]. *Non-blocking* (which is also called lock-freedom) guarantees that *some* active process will always be able to complete its pending operations in a finite number of steps [18]. *Obstruction-freedom* guarantees that an active process will be able to complete its pending operations in a finite number of steps, if all the other processes "hold still" long enough [15]. Obstruction-freedom does not guarantee progress under contention.

Several progress conditions have been proposed for data structures which may involve waiting. *Livelock-freedom* guarantees that processes not execute forever without making forward progress. More formally, livelock-freedom guarantees that, in the absence of process failures, if a process is active, then *some* process, must eventually complete its operation. A stronger property is *starvation-freedom* which guarantees that each process will eventually make progress. More formally, starvation-freedom guarantees that, in the absence of process failures, every active process must eventually complete its operation.

1.6 Contributions

We assume that processes communicate via shared objects. One such object is an *atomic* register. Reading or writing an atomic register is an indivisible action.

Contention-sensitive data structures. We define the new notion of contention-sensitive data structures by identifying four properties any such data structure must satisfy; and discuss three additional "nice to have" properties. This involves introducing a new notion called a *disable-free* code segment (Section 3).

A contention-sensitive election algorithm. We present a contention-sensitive election algorithm, using only atomic read/write registers (Section 4). A contention-sensitive election algorithm, by definition, is *not* required to tolerate failures. Hence, this result does not contradict the known impossibility result that, in the presence of one crash failure, it is not possible to solve election using only atomic read/write registers [31, 42].

A contention-sensitive double-ended queue. We implement a contention-sensitive double-ended queue. To increase the level of concurrency, *two* locks are used: one for the left-side operations and the other for the right-side operations (Section 5).

Three transformations. We present three transformations that facilitate devising contention-sensitive data structures.

- **Transformation 1** converts any contention-sensitive data structure which satisfies livelock-freedom into a corresponding contention-sensitive data structure which satisfies starvation-freedom. Transformation 1 adds only *one* memory reference to the shortcut code (Section 6.1).
- **Transformation 2** converts any obstruction-free data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom (Section 6.2).
- The new notions of a *prevention-freedom* progress condition (of which obstruction-freedom is a special case) and an *exit-safe* data structure are defined. **Transforma-**

tion 3 converts any prevention-free exit-safe data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom (Section 6.3).

Generalizations. We define the notion of a *k-contention-sensitive* data structure in which locks are used only when contention goes above k, and illustrate this notion by implementing a 2-contention-sensitive consensus algorithm. Then, for each $k \geq 1$, we define a progress condition called *k-obstruction-freedom*, and present a (forth) transformation that converts any k-obstruction-free data structure into the corresponding k-contention-sensitive data structure which satisfies livelock-freedom (Section 7).

The motivation for using k-contention-sensitive data structures over course-grained or fine-grained locking is simple. Let's ignore for a moment the benefits gained when the operations are non-interfering, as explained in Subsection 1.4 and focus only on contention. When using locks, a process must always acquire a lock. With a *k-contention-sensitive* data structure, as long as contention is at most k, all the participating processes will execute the shortcut code only which is potentially faster (depending on the implementation) than acquiring the lock followed by executing the critical section and releasing the lock.

Speculative lock elision [34] is a hardware technique which allows multiple processes to concurrently execute critical sections protected by the same lock; when misspeculation, due to data conflicts, is detected rollback is used for recovery, and the execution fall back to acquiring the lock and executing non-speculatively. My motivation for introducing contention-sensitivity (in software) is similar. Furthermore, unlike speculative lock elision, k-contention-sensitivity enables to avoid locking in some cases, even when there is interference.

2 Preliminaries

We focus on an architecture in which n processes communicate asynchronously via shared objects. Asynchrony means that there is no assumption on the relative speeds of the processes. Processes may fail by crashing, which means that a failed process stops taking steps forever. Numerous implementations of locks have been proposed over the years to help coordinating the activities of the various processes.

We are not interested in implementing new locks, but rather assume that we can use existing locks. We are not at all interested whether the locks are implemented using atomic registers, semaphores, etc. We do assume that a lock implementation guarantees that: (1) no two processes can acquire the same lock at the same time, (2) if a process is trying to acquire the lock, then in the absence of failures some process, not necessarily the same one, eventually acquires that lock, and (3) the operation of releasing a lock is wait-free.

We mention below several types of shared objects used in this paper. A *register* is a shared object which supports two operations: reading and writing. A *single-writer* register can be written by one predefined process and can be read by all the processes. A *multi-writer* register can both be written and read by all the processes. With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action. We will consider only atomic registers. In the sequel, by *registers* we mean *atomic* registers.

The implementation of the contention-sensitive double-ended queue, presented in Section 5, is based on shared objects which support load-link/store-conditional/validate

(LL/SC/VL) operations. For a given object o, the operations LL/SC/VL are defined as follows: (1) LL(o) returns o's value. (2) SC(o, v) by process p succeeds if and only if no process has successfully written to o since p's last LL on o. If SC succeeds, it changes o's value to v (or to the value of v, if v is a variable) and returns *true*. Otherwise, o's value remains unchanged and SC returns *false*. (3) VL(o) by p returns *true* if and only if no process performed a successful SC on o since p's last LL on o. Otherwise, VL returns *false*.

In Section 7, a *swap object* is being used. A swap object, supports the swap operation which takes a shared register and a local register, and atomically exchange their values.

An event corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but does not return a value. A (global) state of an algorithm is completely described by the values of the registers and the values of the location counters of all the processes. A run is a sequence of alternating states and events (also referred to as steps). Sometimes, it would be more convenient to define a run as a sequence of events omitting all the states except the initial state.

In a model where participation is required, every process must eventually become active and execute its code. A more interesting and practical situation is one in which participation is *not* required, as is usually assumed when solving resource allocation problems or when designing concurrent data structures. We *always* assume that participation is *not* required.

3 Defining contention-sensitive data structures

An implementation of a contention-sensitive data structure is divided into *two* continuous sections of code: the *shortcut code* and the *body code*. When a process invokes an operation it first executes the shortcut code, and if it succeeds to complete the operation, it returns. Otherwise, the process tries to complete its operation by executing the body code, where it usually first tries to acquire a lock. Once it succeeds to acquire the lock and complete the operation by executing the body code, it releases the acquired lock and returns. Below we formally define the notion of a contention-sensitive data structure.

Definition: The problem of implementing a contention-sensitive data structure is to write the *shortcut code* and the *body code* in such a way that the following *four* requirements are satisfied,

- **Fast path:** In the absence of contention or interference, each operation must be completed while executing the shortcut code only.
- **Wait-free shortcut:** The shortcut code must be wait-free – its execution should require only a finite number of steps and must always terminate. (Completing the shortcut code does not imply completing the operation.)
- **Livelock-freedom:** In the absence of process failures, if a process is executing the shortcut code or the body code, then some process, not necessarily the same one, must eventually complete its operation.

- **Linearizability:** Although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in their "real-time" order.

It is possible to consider replacing linearizability with a weaker consistency requirement, such as sequential consistency [21]. Sequential consistency is defined as follows: The result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Livelock-freedom may still allow that individual processes may never complete their operations. We will examine also solutions which do not allow such a behavior.

- **Starvation-freedom:** In the absence of process failures, if a process is executing the shortcut code or the body code, then this process, must eventually complete its operation.

Next, we define two additional desirable properties. They are "nice to have", but it is not required that each correct implementation satisfies them. Both these properties add some limited ability to tolerate process failures. The first property has to do with a process failure that happens while executing the shortcut code, the second property has to do with a process failure that happens while executing the body code. In both cases, under various assumptions, such failures should not prevent other processes from completing their operations. First, we introduce a new notion called *disable-freedom*.

A code segment is *disable-free*, if a process that fails while executing that code segment may not prevent other processes from completing their operations. A disable-free code segment is not necessarily wait-free and vice versa. To illustrate this point, consider the following program for two processes in which a single atomic register, called x, is used. Each process executes the following three lines and terminates: (1) $x := 0$; (2) $x := 1$; (3) **while** $x \neq 1$ **do** skip **od**. Consider the code segment which consists of lines 1 and 2. It is clearly wait-free, but it is not disable-free since a process that fails just before executing line 2 may cause the other process to spin forever (in line 3). On the other hand, the code segment which consists of only line 3 is disable-free but is not wait-free.

- **Disable-free shortcut:** A process that fails (or that is very slow) while executing the shortcut code, may not prevent other processes from accessing the data structure and completing their operations.

We point out that the shortcut code of the consensus algorithm presented in the introduction is disable-free. The second "nice to have" property is,

- **Weak-blocking body:** Let p be a process that has failed while executing the body code, and let q be a process that has started executing the shortcut code after p has failed. Furthermore, assume that the operations of p and q are non-interfering, and that no other process is concurrently participating. Then, the fact that p has failed should not prevent q from completing its operation while executing the shortcut code.

The implementation of the body code can be either coarse-grained, or fine-grained.

4 A contention-sensitive election algorithm

The *election problem* is to design an algorithm in which all participating processes choose one process as their leader. More formally, each process that starts participating eventually decides on a value from the set $\{0, 1\}$ and terminates. It is required that exactly one of the participating processes decides 1. The process that decides 1 is the elected leader. Notice that it is *not* required that a process which is not elected needs to know the identity of the leader. This variant of the problem is sometimes called "implementing a test-and-set object". Processes are not required to participate in the algorithm.

4.1 The algorithm

The following algorithm solves the election problem for any number of processes, and is related to the splitter constructs from [20, 26, 29]. A single lock is used. It is assumed that after a process executes a **decide**() statement, it immediately terminates.

We start with an intuitive description of the algorithm. While executing the shortcut, each participating process is assigned into one of three groups. We name these groups: *east*, *north* and *west* (the reader may think as if the processes are arriving from the *south*). This splitting of the processes is implemented using two shared registers called x and y. The initial value of x is immaterial and the initial value of y is 0.

- *The east group.* A process first sets x to its id, and then checks the value of y. If it finds that $y = 1$, it knows that some other process has already set y to 1. At that point we say that the process belongs to the east group. The process gives up on being elected and decides 0. Just before deciding, the process notifies everybody that the east group is not empty, by setting a shared bit called b (which is initially 0) to 1.

- *The north group.* A process first sets x to its id, and then checks the value of y. If it finds that $y = 0$, it knows that no other process has yet written into y. The process sets y to 1, and checks the value of x. If the value of x *equals* to its id, it knows that no other process has set x since it has written to x. At that point we say that the process belongs to the north group. At most one process may belong to north group. The process notifies everybody that the north group is not empty, by setting a shared register called z (which is initially 0) to its id. Then, the process checks b to decide whether the east group is empty or not. If $b = 0$ (the east group is empty) it decides 1 and becomes the elected leader, otherwise the process is moved to the *west* group.

- *The west group.* A process first sets x to its id, and then checks the value of y. If it finds that $y = 0$, the process sets y to 1, and checks the value of x. If the value of x is *different* from its id, it knows that there is contention. At that point we say that the process belongs to the west group. Next, the process tries to acquire the lock. Once it acquires the lock it does the following:

  - If it was in the north group before (i.e., z equals its id) and no leader is elected yet, then it sets a shared register called *done* (which is initially 0) to 1, releases the lock, and decides 1. That is, it is the elected leader.
  - Otherwise, the process concludes that at least one other process has moved to the east or north groups. It waits until this process " reveals itself" by setting

either the b bit or the z register. After waiting, if $z = 0$ and no leader is yet elected (i.e., $done = 0$), it knows that no process can be elected from the north group. It sets *done* to 1, releases the lock, and decides 1. That is, it is the elected leader.

– Otherwise, if after waiting it finds out that either the north group is not empty (i.e., $z = 1$) or a leader has been elected (i.e., $done = 1$), it releases the lock, and decides 0.

The algorithm with a detailed correctness proof appears below.

CONTENTION-SENSITIVE ELECTION: Process i's program

shared x, z: atomic registers, initially $z = 0$ and the initial value of x is immaterial
b, y, *done*: atomic bits, initially all 0
local *leader*: local register, the initial value is immaterial

```
1   x := i                                                        // begin shortcut
2   if y = 1 then b := 1; decide(0) fi                            // I am not the leader – east
3   y := 1
4   if x = i then z := i; if b = 0 then decide(1) fi fi           // I am the leader! – north
                                                                  // end shortcut
5   lock                                                          // locking – west
6   if z = i ∧ done = 0 then leader = 1                           // I am the leader!
7       else await b ≠ 0 ∨ z ≠ 0
8            if z = 0 ∧ done = 0 then leader = 1; done := 1       // I am the leader!
9                 else leader = 0                                // I am not the leader
10           fi
11  fi
12  unlock ; decide(leader)                                       // unlocking
```

When a process runs alone before a leader is elected, it is elected and terminates after accessing the shared memory *six* times. Furthermore, all the processes that start running *after* a leader is elected terminate after three steps. The algorithm does not satisfy the disable-free shortcut property: a process that fails just before the assignment to b in line 2 or fails just before the assignment to z in line 4, may prevent other processes spinning in the *await* statement (line 7) from terminating.

4.2 A correctness proof

In any execution, define the *latecomers* to be the processes that start participating in the election algorithm after the first process decides (and terminates). Let m be the number of *early* processes that start participating in the election algorithm before some process decides.

Lemma 4.1 *In any execution of the election algorithm the following properties hold: (1) At most $m - 1$ early processes decide in line 2. (2) The latecomers all decide in line 2. (3) At least one early process decides while executing the shortcut code or at least one latecomer decides in line 2. (4) At most one process writes into z in line 4, at most one process decides*

in line 4, and if $m = 1$ then exactly one process decides in line 4. (5) The shortcut code is wait-free.

Proof: As for Property 1, we observe that a process that decides in line 2, must first read that $y = 1$ at the first statement in line 2. Before that happens some other process, say i, must have set y to 1 in line 3. Thus, i is an early process that does decide in line 2. To prove Property 2, observe that when a latecomer tests the value of y in line 2, it always finds that $y = 1$ and hence can not continue to line 3 and must decide in line 2.

To prove Property 3, observe that it follows from Property 2, that only the m early processes may not decide while executing the shortcut code. Let us assume that all the m early processes do not decide while executing the shortcut code. Let i be the last early process to set x to i in line 1. When i tests the value of x at the statement in line 4, either $x \neq i$ or $x = i$. If $x \neq i$, then some latecomer process, say j, must have modified x and, by property 2, j decides in line 2. If $x = i$ then (since i does not decide in line 4) i must find that $b \neq 0$ which implies that some latecomer has set b to 1, and later decides in line 2.

To prove Property 4, assume to the contrary that two processes i and j either both write into z in line 4 or both decide in line 4. Assume without loss of generality that process i tests the value of x at the statement in line 4 after j does so. This implies that x is not written by any process between i's assignment $x := i$ in line 1 and i's read of x in line 4. Thus, j reads of x in the statement in 4 preceded i's assignment in the statement in line 1, which in turn implies that j assigned 1 to y in the statement in line 3 before i's read of y in line 2. Thus, i must have read $y = 1$ at the statement in line 2 and then decide in line 2, a contradiction. When $m = 1$, the first process to decide runs alone without any interference. In such a case, going through the code, that process must decide in line 4. The correctness of Property 5 is obvious. ∎

Theorem 4.2 *Every participating process eventually decides and terminates; and, assuming that at least one process participates, exactly one process decides 1 and is elected as the leader.*

Proof: The third property of Lemma 4.1 assures that some process must decide in line 2 or in line 4, and perform the assignment to b or to z enabling all the *await* statements (line 7) to terminate. Thus, eventually, every participating process decides and terminates. There are exactly three locations where a process can be elected: in line 4 deciding on 1; in line 6 setting *leader* to 1; and in line 8 setting *leader* to 1.

We first prove that if a process decides in line 4 and becomes a leader, then nobody else becomes a leader by setting *leader* to 1 in line 6 or in line 8. To see why this holds, observe that: (1) a process that reaches the await statement in line 7, can set *leader* to 1 in line 8 and later become the leader only if b is set to 1 *before* z is set to a value other than 0, and (2) a process i can decide in line 4 and become a leader, only if b is set to 1 *after* z is set to i. Thus, if process i decides in line 4, by Lemma 4.1(4) only process i has written into z (afterwhich $z = i$), and this write must have happened before b was set to 1. This implies that every other process j that tests z in line 6, finds that $z \neq j$ and does not set *leader* to 1 in line 6, and every other process that reaches the await statement in line 7, finds later when it tests z line 8 that $z \neq 0$ and also is not elected.

If a process, after acquiring the lock, becomes a leader by setting *leader* to 1 in line 6, it means that this process has found that z equals its id. Since, by Lemma 4.1(4) an

assignment to z may happen at most once, any other process that will acquire the lock later will not be able to become a leader by setting *leader* to 1 in either line 6 or in line 8.

If a process becomes a leader by setting *leader* to 1 in line 8, before it releases the lock it sets *done* to 1, preventing any other process that acquires the lock later from becomes a leader by setting *leader* to 1 in either line 6 or in line 8. Thus, at most one leader is elected.

Finally, we prove that at least one leader is elected. If some process sets *leader* to 1 in line 8, then we are done. So assume that no process sets *leader* to 1 and *done* to 1 in line 8. Since *done* is only modified in line 8, its value must always stay 0. Furthermore, assume that some process executes the test in line 8, otherwise it is easy to see that a leader must have been elected. Thus, the test at line 8 fails, since $z = i$ for some process i. By Lemma 4.1(4), an assignment to z may happen at most once, and it only happens in line 4. So, after process i sets z, it tests b. If $b = 0$ then i is elected and we are done, so assume that $b \neq 0$. Thus, when i executes later the test in line 6, it finds that $z = i$ and that *done* $= 0$, and is elected. ∎

Theorem 4.3 *The election algorithm satisfies the following four requirements: fast path, wait-free shortcut, livelock-freedom, and linearizability.*

Proof: The second and forth properties of Lemma 4.1 imply that the fast path requirement is satisfied. As the shortcut code includes no await statements, wait-free shortcut is satisfied. To prove that the livelock-freedom property is satisfied, we observe that the third property of Lemma 4.1 assures that some process must decide in line 2 or in line 4, and perform the assignment to b or to z enabling all the *await* statements (line 7) to terminate. Thus, eventually, every participating process decides and terminates. As for linearizability, the operation of the elected leader is linearized at the beginning of its operation, while the operation of each other process is linearized at the end of its operation. ∎

5 A contention-sensitive double-ended queue data structure

In [15], two obstruction-free CAS-based implementations of a double-ended queue are presented; the first is implemented on a linear array, the second on a circular array. In the following, a contention-sensitive double-ended queue data structure implementation, which is based on the implementations from [15], is presented. The double-ended queue is implemented on an infinite array (denoted Q) and is based on shared objects which support load-link/store-conditional/validate (LL/SC/VL) operations.

Two locks are used: *llock* (left lock) is used by the left-side operations and *rlock* (right lock) is used by the right-side operations. Two values *lnil* (left null) and *rnil* (right null) that are different from the data values are used, and the following invariant is maintained:

> For every two integer values $i < j$, $Q[j] = lnil$ implies $Q[i] = lnil$, and $Q[i] = rnil$ implies $Q[j] = rnil$.

Two pointers are used: *Lptr* (left pointer) which holds the index of the rightmost *lnil* value, and *Rptr* (right pointer) which holds the index of the leftmost *rnil* value. A *rightpush*(*value*) (resp. *leftpush*(*value*)) operation changes the leftmost *rnil* (resp. rightmost *lnil*) value to

value. A *rightpop* (resp. *leftpop*) operation changes the rightmost (resp. leftmost) data value to *rnil* (resp. *lnil*) and returns that value.

The right-side operations, rightpush and rightpop, are shown in Figure 1. The left-side operations, leftpush and leftpop, are symmetric to the right-side operations, and hence are not presented.

When a process p invokes a right-side operation, p first reads the *Rptr* pointer to find the index of the exact location, say k, of the rightmost *lnil* value. Then, it LL($Q[k]$) and also LL $Q[k]$'s adjacent location $Q[k-1]$. In order to prevent interference by another right-side operation, process p first SC to the adjacent location $Q[k-1]$ in case of a push operation or to $Q[k]$ in case of a pop operation (without changing that location's value). If this SC succeeds, the process SC to $Q[k]$ in case of a push operation or to $Q[k-1]$ in case of a pop operation. As a result of this approach, two concurrent right-side push and pop operations can each cause the other to retry. In such a case, p tried to acquire the *rightlock* and, in its critical section, p continually repeats the above sequence of steps trying to complete its operation.

A concurrent left-side and right-side operations can interfere if they try to apply a SC to the same memory location. We observe that in such a case if as a result one of the two types of operations has to retry, then it must be the case that an operation of the other type must be completed. Since *Rptr* is updated using an atomic write operation, the implementation in Figure 1 does not satisfy the disable-free shortcut and the weak-blocking body properties.

Theorem 5.1 *The implementation in Figure 1, correctly implements a contention-sensitive double-ended queue and satisfies the following four requirements: fast path, wait-free shortcut, livelock-freedom, and linearizability.*

Proof: In the absence of interference, a process completes its (push or pop) operation after executing at most one atomic read, one atomic write, two LL, two SC and one VL operations, and without locking. Thus, the fast path requirement is satisfied. As the shortcut code includes no await statements, the wait-free shortcut property is also clearly satisfied.

To see that the livelock-freedom property is satisfied, we observe that a process first tries to complete its right-side operation by executing the shortcut code. If there is no contention the process will complete its operation without locking. Otherwise if the process does not succeed in completing its operation by executing the shortcut code, it tries to acquire *rlock*. A process that is holding *rlock* may experience interference from some other right-side operations. However, either *some* process will manage to complete its right-side operation or, because the number of processes is finite, this interference will vanish after some finite time. At that point, the process that holds *rlock* will be able to complete its operation, assuming there is no interference by a left-side operation.

Next we show that there can not be alivelock as a result of interference between concurrent left-side and right-side operations. Assume to the contrary that such a livelock can occur. This means that eventually there will be two processes which hold the two locks and each one of the other processes that are involved in the livelock will be waiting for a lock to be released. Let's focus on the two active operations invoked by the two processes that are holding the two locks. Two concurrent left-side and right-side operations interfere, when they both try to apply SC to the same memory location. In such a case, focusing only on the above two active operations, one of the SC will fail and the other will succeed.

CONTENTION-SENSITIVE DOUBLE-ENDED QUEUE: program for each one of the n processes

shared $Q[-\infty..\infty]$: infinite array; initially, $Q[i] = lnil$ for all $i < 0$ and $Q[i] = rnil$ for all $i \geq 0$
$Lptr, Rptr$: integers; initially, $Lptr = -1$ and $Rptr = 0$
local $done, empty$: boolean; $cur, prev$: both range over {all data values, $lnil, rnil$}
k: integer

```
rightpush(value)                                                    // value ∉ {lnil, rnil}
1   k := Rptr; prev := LL(Q[k − 1]); cur := LL(Q[k]);               // k index of leftmost rnil
2   if cur = rnil ∧ prev ≠ rnil then                                // Rptr is updated
3       if SC(Q[k − 1], prev) then                                  // prevent interfering operations
4           if SC(Q[k], value) then                                 // push new value
5               Rptr := Rptr + 1; return("ok") fi fi fi             // update Rptr
6   [lock(rlock)]
7   done := false                                                   // set local variable
8   repeat
9       k := Rptr; prev := LL(Q[k − 1]); cur := LL(Q[k])           // k index of leftmost rnil
10      if cur = rnil ∧ prev ≠ rnil then                            // Rptr is updated
11          if SC(Q[k − 1], prev) then                              // prevent interfering operations
12              if SC(Q[k], value) then                             // push new value
13                  Rptr := Rptr + 1; done := true fi fi fi         // update Rptr
14  until (done)
15  [unlock(rlock)] ; return("ok")                                  // unlocking section

rightpop()
1       k := Rptr; prev := LL(Q[k − 1]); cur := LL(Q[k])           // k index of leftmost rnil
2       if cur = rnil ∧ prev ≠ rnil then                            // Rptr is updated
3           if prev = lnil ∧ VL(Q[k − 1]) then return("empty")      // adjacent lnil & rnil
4           else if SC(Q[k], rnil) then                             // prevent interfering operations
5               if SC(Q[k − 1], rnil) then                          // pop value
6                   Rptr := Rptr − 1; return(prev) fi fi fi fi       // update Rptr
7   [lock(rlock)]
8   done := false; empty := false                                   // set local variables
9   repeat
10      k := Rptr; prev := LL(Q[k − 1]); cur := LL(Q[k])           // k index of leftmost rnil
11      if cur = rnil ∧ prev ≠ rnil then                            // Rptr is updated
12          if prev = lnil ∧ VL(Q[k − 1]) then empty := true        // adjacent lnil and rnil
13          else if SC(Q[k], rnil) then                             // prevent interfering operations
14              if SC(Q[k − 1], rnil) then                          // pop value
15                  Rptr := Rptr − 1; done := true fi fi fi fi       // update Rptr
16  until (done ∨ empty)
17  [unlock(rlock)] ; if done then return(prev) else return("empty") fi   // unlocking
```

Figure 1: A contention-sensitive double-ended queue data structure. The left-side operations, leftpush and leftpop, are symmetric to the right-side operations. The first 5 lines (6 lines, resp.) of the rightpush (rightpop, resp.) operation is the shortcut code. Two locks are used: *llock* (left lock) is used by the left-side operations and *rlock* (right lock) is used by the right-side operations.

As a result, one of the two operations will have to retry, and the other operation will be successfully completed. A contradiction. This explains why livelock-freedom is satisfied.

Next we consider linearizability. In the following, RHj refers to line #j in the code of the rightpush method, and RPj refers to line #j in the code of the rightpop method.

Each *rightpush*(*value*) operation is linearized to the point at which it changes an *rnil* to *value* (RH4, RH12). We show that when a *rightpush*(*value*) operation changes the value of some $Q[k]$ from *rnil* to *value* (RH4, RH12), $Q[k-1] \neq rnil$. Before the assignment to $Q[k]$ takes place and the rightpush operation succeeds, it is verified that $Q[k-1] \neq rnil$ and then there is a successful SC to $Q[k-1]$, without changing its value (RH3, RH11). We show that after the successful SC to $Q[k-1]$, and before the successful SC to $Q[k]$ (RH4, RH12), no concurrent operation could not have changed $Q[k-1]$ to *rnil*.

Only rightpop operations may assign *rnil*. Assume that some concurrent rightpop operation tries to SC to $Q[k-1]$, between the two successful SC to $Q[k-1]$ and $Q[k]$ by the rightpush operation. This implies that the preceding SC to $Q[k]$ by rightpop operation was successful, which in turn implies that the LL of $Q[k-1]$ by the rightpop operation, must have happened before the successful SC to $Q[k-1]$ by the rightpush operation. Thus the SC to $Q[k-1]$ by the rightpop operation must fail.

Each *rightpop* operation that returns *value* is linearized to the point at which it changes an array location from *value* to *rnil* (RP5, RP14). We show that when a *rightpop* operation changes the value of some $Q[k-1]$ from *value* to *rnil* (RP5, RP14), $Q[k] = rnil$. (The proof is similar to the one proved in the previous paragraph.) Before the assignment to $Q[k-1]$ takes place and the rightpop operation succeeds, it is verified that $Q[k] = rnil$ and then there is a successful SC to $Q[k]$, without changing its value (RP4, RP13). We show that after the successful SC to $Q[k]$, and before the successful SC to $Q[k-1]$ (RP5, RP14), no concurrent operation could not have changed $Q[k]$ from *rnil* to something else.

Only rightpush operations may change $Q[k]$ from *rnil* to something else. Assume that some concurrent rightpush operation tries to SC to $Q[k]$, between the two successful SC to $Q[k]$ and $Q[k-1]$ by the rightpop operation. This implies that the preceding SC to $Q[k-1]$ by rightpush operation was successful, which in turn implies that the LL of $Q[k]$ by the rightpush operation, must have happened before the successful SC to $Q[k]$ by the rightpop operation. Thus the SC to $Q[k]$ by the rightpush operation must fail.

Each *rightpop* operation that returns "empty" is linearized to the point at which it was last executed the statement "$cur := \mathrm{LL}(Q[k])$" (RP1, RP10). We show that if a *rightpop* operation returns "empty", then at the moment when it was last executed the statement "$cur := \mathrm{LL}(Q[k])$", $Q[k-1] = lnil \wedge Q[k] = rnil$, which means that the queue was empty at that moment. The string "empty" is returned only if $\mathrm{VL}(Q[k-1])$ returns *true* (RP3, RP12). Thus, from the time the statement "$prev := \mathrm{LL}(Q[k-1])$" was last executed (RP1, RP10) until the validation (RP3, RP12), the value of $Q[k-1]$ has not changed. In particular, $Q[k-1]$ equals *lnil* at the moment the value *rnil* was assigned to *cur*, when the statement "$cur := \mathrm{LL}(Q[k])$" was last executed (RP1, RP10). (The last value of *cur* is known from the test in RP2, RP11.)

The left-side operations, which are symmetric to the to right-side operations, are linearized similarly. Furthermore, a location in Q changes only when an operation is linearized as above. ∎

6 Three transformations

Recall the question raised in the introduction: If a sequential data structure cannot be used as the basic building block for constructing a contention-sensitive data structure, what is the best data structure to use? The following three transformations that facilitate devising such data structures provide an answer. The correctness proofs of the transformations are straightforward and are left as an exercise for the reader.

6.1 From livelock-freedom to starvation-freedom

The transformation converts any contention-sensitive data structure, denoted A, which satisfies livelock-freedom into a corresponding contention-sensitive data structure, denoted B, which satisfies starvation-freedom. It adds only *one* memory reference to the shortcut code. It is an extension of a known transformation, for the mutual exclusion problem, that has appeared in [39] (page 83).

It is assumed that A is implemented using a single lock, and that the *body* of A is divided into three continuous sections of code: *locking*, *main-body*, and *unlocking*. When a process invokes an operation on A it first executes the shortcut code of A, and if it succeeds to complete the operation, it returns. Otherwise, it executes the body code, where it first tries to acquire the single lock by executing the locking code. If it succeeds to acquire the lock, it executes the *main-body*. If it succeeds to complete the operation, it releases the lock.

Using A, we construct B as follows: In addition to the objects used in A, we use an atomic register called *turn* which is big enough to store a process identifier, a boolean array called *flag*, and a boolean bit called *contention*. All the processes can read and write *turn* and the *contention* bit, the processes can read the bit $flag[i]$, but only process i can write $flag[i]$. The processes are numbered 1 through n.

When process i invokes an operation on B it first executes the shortcut code of B. In this shortcut i first checks if there is contention or interference by testing the value the *contention* bit. If there is no contention or interference (i.e., *contention* = *false*) then i executes the shortcut of A. Otherwise, i starts executing the body code of B.

In the body code, it first sets $flag[i]$ to *true* letting everybody know that it is around. Then, i waits until one of two events happens: either $turn = i$ which means that it is process' i turn to proceed or until $flag[turn] = false$ which means that the process that it is its turn to proceed is not active. Then, process i executes the locking part of A trying to acquire the lock. Once it succeeds, it sets the *contention* bit to *true* letting all the newcomers know that there is contention; it executes the main-body of A; and sets the *contention* bit back to *false*. Before it release the lock, process i checks the $flag[turn]$ bit and if the value is *false* it increments *turn* by 1, giving priority to the next process in line. Finally it executes the unlocking code of A.

The exact code is given below. The statement "**await** *condition*" is used as an abbreviation for "**while** $\neg$*condition* **do** *skip*".

Transformation 1: process i's program.
Initially: $flag[i] = false$, $contention = false$, the initial value of *turn* is an arbitrary number between 1 to n.

```
1   if contention = true then goto lock fi          // begin shortcut of B
```

```
2    shortcut of A                                          // end shortcut of B

3  lock: flag[i] := true                                    // begin body of B
4    await (turn = i or flag[turn] = false)
5    locking of A

6    contention := true
7    main-body of A
8    contention := false

9    flag[i] := false
10   if flag[turn] = false then turn := (turn mod n) + 1 fi
11   unlocking of A                                         // end body of B
```

Setting the contention bit to true, happens after acquiring the lock which implies that there has been contention and interference. Evaluating the condition $flag[turn] = false$ requires *two* memory references, one to read *turn* and the other to read the *flag* bit.

It is easy to see that the transformation satisfies starvation-freedom. Assume some arbitrary process, say i, starts executing B. Then, either i terminates after executing the shortcut, or i reaches the await statement in line 4. Each process that acquires the lock eventually executes line 10 in which *turn* is incremented if $flag[turn] = false$. This guarantees that eventually process i will move past the await statement. Furthermore, either process i will acquire the lock or the value of *turn* will be set to i. Once $turn = i$, no newcomers will be able to move past the await statement in line 4, enabling i to eventually acquire the lock and complete its operation.

6.2 From obstruction-freedom to livelock-freedom

Next we present a transformation that converts a obstruction-free data structure, denoted *DS*, into a corresponding contention-sensitive data structure. The idea is to use a lock to choke down parallelism and eventually eliminate interference on an obstruction-free data structure.

Let us denote by *first*(*DS*) the number of steps that a process needs to take in order to complete its operation of *DS* when there is no contention. In simple data structures like queues and stacks, *first*(*DS*) would be a known constant. In a data structure like a linked list, *first*(*DS*) may depend on the current length of the list. Thus, a linked list may have the property that *first*(*DS*) increases as the execution progresses. In the specific case of a lined list, *first*(*DS*) can be calculated by having each process records in a shared location the number of insert and delete operations it has preformed so far. Now, each process can read these numbers and calculate *first*(*DS*).

The following transformation is restricted to obstruction-free data structure implementations, in which a process, after taking an a priori known number of steps, can figure out what *first*(*DS*) should be.[1] The transformation uses a single lock.

[1] A simpler way to deal with the case that *first*(*DS*) can change over time is to restrict attention only to *bounded* obstruction-free data structures. For a given bounded obstruction-free data structure, there is a finite bound such that every operation completes within that number of steps after it encounters no more interference. Restricting attention only to *bounded* obstruction-freedom seems too restrictive.

Transformation 2: program for a process which *invokes* operation *op*.

```
1  execute up to first(DS) steps of DS                          // shortcut
2  if op is completed then return response fi
3  lock                                                         // body
4  continue to execute steps of DS until op is completed
5  unlock
```

First a process tries to complete its operation *op* of *DS* without holding the lock. If there is no contention the process will complete its operation without locking. Otherwise, if after taking *first*(*DS*) steps, it does not succeed in completing its operation, it tries to acquire the lock. As a result of such an approach, a process that is already holding the lock may experience interference. However, either *some* process will manage to complete its operation without holding the lock, or (since the number of processes is finite) this interference will eventually vanish.

6.3 From prevention-freedom to livelock-freedom

We introduce a new progress condition, called prevention-freedom, which is an interesting *generalization* of the obstruction-freedom progress condition. Then, we present a transformation that converts a prevention-free data structure, into a corresponding contention-sensitive livelock-free data structure. Since obstruction-freedom is a special case of prevention-freedom, Transformation 3 below generalizes Transformation 2.

For a given implementation of a concurrent data structure, DS, assume that each statement is uniquely numbered by a natural number. Let S_i denote the set of all the numbers of statements in the code of process p_i (where $i \in \{1, ..., n\}$). For $s \in S_i$, we say that process p_i is at s if the next step of p_i is to execute the statement numbered s. Let G_i be a subset of S_i. In the following definition, intuitively, the G_i's include location in the codes such that a process that is suspended or even crashed in one of these locations does not block other processes.

> **Prevention-freedom:** A data structure is *prevention-free* w.r.t. $\{G_1, ..., G_n\}$ if it is guaranteed that each process p_i will be able to complete its pending operations in a finite number of steps, if all the other processes simultaneously "hold still" long enough, where each process $p_j \neq p_i$ "holds still" (i.e., waits) at some $g_j \in G_j$.

Each $g_j \in G_j$ is called a *gate*. Prevention-freedom guarantees that if $n - 1$ processes are suspended or even crashed while each one of them is at a gate, the remaining process is not effected and can complete its operation. We assume that when a process does not invoke an operation, it is at a gate. A data structure is obstruction-free if and only if, it is prevention-free w.r.t. $\{S_1, ..., S_n\}$. In an obstruction-free data structure each (number of a) statement is a gate.

> **The exit-safe property:** We say that a prevention-free data structure is *exit-safe* if, for every process p, it is always the case that after p invokes an operation of *DS* and takes *first*(*DS*) steps, either (1) p completes its operation or (2) by

taking a finite number of additional steps p always reaches a gate, regardless of the behavior of the other processes.

Below we present a transformation which converts a prevention-free exit-safe data structure, denoted *DS*, into a corresponding contention-sensitive data structure. The following transformation is restricted to prevention-free data structure implementations, in which a process, after taking an a priori known number of steps, can figure out what $first(DS)$ should be. The transformation uses a single lock.

> **Transformation 3:** First a process tries to complete its operation *op* of *DS* without holding the lock. If there is no contention the process will complete its operation without locking. Otherwise if the process, after taking $first(DS)$ steps, does not succeed in completing its operation it continues taking steps until it reaches a gate, and at that point it "exits" the *DS* code, and tries to acquire the lock. Once it acquires the lock it "enters" the *DS* code at the same point where it left it – i.e., through the gate – and continues taking steps trying to complete the operation *op*. If *op* is completed it releases the lock.

A data structure which is constructed using Transformation 3, does not necessarily satisfy the disable-free shortcut property or the weak-blocking body property.

7 Generalizations

In this section we present an interesting generalization of the notion of contention-sensitives data structures, called *k-contention-sensitive* data structures, where $k \geq 1$. A k-contention-sensitive data structure is a data structure in which contention resolution using locks is used only when contention goes above k. It is defined by modifying the *fast path* requirement.

As in the definition of contention-sensitive data structure from Section 3, an implementation of a k-contention-sensitive data structure is divided into *two* continuous sections of code: the *shortcut code* and the *body code*. When a process invokes an operation it first executes the shortcut code, and if it succeeds to complete the operation, it returns. Otherwise, the process tries to complete its operation by executing the body code, where it usually first tries to acquire a lock. Once it succeeds to acquire the lock and complete the operation by executing the body code, it releases the acquired lock and returns. Below we formally define the notion of a k-contention-sensitive data structure.

Definition: The problem of implementing a k-contention-sensitive data structure is to write the *shortcut code* and the *body code* in such a way that the following *four* requirements are satisfied,

> **k-fast path:** When there is contention of at most k processes, or when there is no interference, each operation must be completed while executing the shortcut code only.

The other three requirements, namely, *wait-free shortcut*, *livelock-freedom* and *linearizability*, are as defined in Section 3.

We demonstrate this idea, by presenting a 2-contention-sensitive consensus algorithm. The algorithm uses atomic registers and a single swap object. It is known that there is no wait-free consensus algorithm for more than two processes, using atomic registers and atomic swap objects [14].

2-CONTENTION-SENSITIVE CONSENSUS: program for process p_i with input $v_i \in \{0, 1\}$.

shared $x[0..1]$: array of two atomic bits, initially both 0
y, *out* : atomic registers which range over $\{\perp, 0, 1\}$, initially both $\perp$
z : a swap object which ranges over $\{\perp, 0, 1\}$, initially $\perp$
local in_i : a register which ranges over $\{\perp, 0, 1\}$

0 $in_i := v_i$; $swap(z, in_i)$; **if** $in_i = \perp$ **then** $in_i := v_i$ **fi** // start shortcut code
1 $x[in_i] := 1$
2 **if** $y = \perp$ **then** $y := in_i$ **fi**
3 **if** $x[1 - in_i] = 0$ **then** $out := in_i$; **decide**(in_i) **fi**
4 **if** $out \neq \perp$ **then decide**(out) **fi** // end shortcut code
5 lock **if** $out = \perp$ **then** $out := y$ **fi** unlock ; **decide**(out) // locking

Processes are not required to participate, however, once a process starts participating it is guaranteed that it may fail only while executing the shortcut code. Once a process decides, it immediately terminates. For a set of processes P, let $|P|$ denote the size of P. Consider the following generalization of the notion of obstruction-freedom:

> **k-obstruction-freedom:** For any $k \geq 1$, the progress condition *k-obstruction-freedom* guarantees that for every set of processes P where $|P| \leq k$, every process in P will be able to complete its pending operations in a finite number of steps, if all the processes not in P do not take steps for long enough.

These progress conditions cover the spectrum between obstruction-freedom and wait-freedom; 1-obstruction-freedom is the same as obstruction-freedom, and in a system of k processes, k-obstruction-freedom is the same as wait-freedom. The following transformation converts any k-obstruction-free data structure, denoted *DS*, into a corresponding k-contention-sensitive data structure which satisfies livelock-freedom. Let us denote by *k-first*(DS) the number of steps that a process needs to take in order to complete its operation of *DS* when the contention level is at most k. The following transformation is restricted to data structure implementations, in which a process, after taking an a priori known number of steps, can figure out what *first*(DS) should be.

> **Transformation 4:** First a process tries to complete its operation *op* of *DS* without holding the lock. If the contention level is at most k, the process will complete its operation without locking. Otherwise if the process, after taking *k-first*(DS) steps, does not succeed in completing its operation it "exits" the *DS* code, and tries to acquire the lock. In this case it is sufficient to use a k-exclusion lock.[2] Once it acquires the lock it "enters" the *DS* code at the same point where

[2] A k-exclusion lock guarantees that: (1) no more than k processes can acquire the lock at the same time, (2) if strictly fewer than k processes fail (are delayed forever) then if a process is trying to acquire the lock, then some process, not necessarily the same one, eventually acquires the lock, and (3) the operation of releasing a lock is wait-free.

it left it and continues taking steps trying to complete the operation *op*. If *op* is completed it releases the lock.

A similar transformation can be designed for the following weaker condition:

> **k-obstacle-freedom:** For any $k \geq 1$, the condition *k-obstacle-freedom* guarantees that for every set of processes P where $|P| \leq k$, **some** process in P with pending operations will be able to complete its operations in a finite number of steps, if all the processes not in P do not take steps for long enough.

We notice that, 1-obstacle-freedom is the same as obstruction-freedom, and in a system of k processes, k-obstacle-freedom is the same as non-blocking. The correctness proofs of the 2-contention-sensitive consensus algorithm and of Transformation 4 are rather simple and are left as an exercise for the reader.

8 Related work

Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [5]. Since then, numerous implementations of locks have been proposed [32, 39]. Algorithms for several concurrent data structures based on locking have been proposed since at least the 1970s [4, 7, 19, 24]. Speculative lock elision [34] is a hardware technique which allows multiple processes to concurrently execute critical sections protected by the same lock; when misspeculation, due to data conflicts, is detected rollback is used for recovery, and the execution falls back to acquiring the lock and executing non-speculatively.

The benefit of avoiding locking has already been considered in [6]. Implementations of data structures which avoid locking have appeared in many papers, a few examples are [6, 10, 13, 28, 37, 43]. Several progress conditions have been proposed for data structures which avoid locking. The most extensively studied conditions, in order of decreasing strength, are wait-freedom [14], non-blocking [18], and obstruction-freedom [15]. All strategies that avoid locks are called lockless [17] or lock-free [27]. (In some papers, lock-free means non-blocking.)

Non-blocking and wait-freedom (although desirable) may impose too much overhead upon the implementation, and are often complex. Requiring implementations to satisfy only obstruction-freedom can simplify the design of algorithms. However, since it does not guarantee progress under contention, such algorithms may suffer from livelocks. Various contention management techniques have been proposed to improve progress of obstruction-free algorithms under contention while still avoiding locking [11, 35]. Other works investigated boosting obstruction-freedom by making timing assumption [3, 8, 38] and using failure detectors [12].

In [41], it is shown that between the two extremes, *lock-based* algorithms, which involve "a lot of waiting", and *wait-free* algorithms, which are "free of locking and waiting", there is an interesting spectrum of different levels of waiting. New progress conditions, called k-waiting, for $k \geq 0$, which are intended to capture the "amount of waiting" of processes in asynchronous concurrent algorithms, were introduced. To illustrate the utility of the new k-waiting conditions, they are used in [41] to derive new lower and upper bounds, and impossibility results for well-known basic problems such as consensus, election, renaming and mutual exclusion. Furthermore, the relation between waiting and fairness was explored.

It is known that even in the presence of only one crash failure, it is not possible to solve consensus using atomic read/write registers only [9, 22]. Wait-free consensus algorithms that use read and write operations in the absence of (process) contention, or even in the absence of step contention, and revert to using strong synchronization operations when contention occurs, are presented in [2, 23]. A wait-free consensus algorithm that in any given execution uses objects with consensus number above k, only when contention goes above k, appeared in [30].

Following the notion contention-sensitive data structures introduced in the conference version of this paper [40], hybrid implementations of concurrent objects in which lock-based code and lock-free code are merged in the same implementation of a concurrent object, are discussed in [33].

Consistency conditions for concurrent objects are linearizability [18] and sequential consistency [21]. A tutorial on memory consistency models can be found in [1]. Transactional memory is a methodology which has gained momentum in recent years as a simple way for writing concurrent programs [16, 36, 44]. It has implementations that use locks and others that avoid locking, but in both cases the complexity is hidden from the programmer. In [25], a constructive critique of locking and transactional memory: their strengths, weaknesses, and challenges, is presented.

9 Discussion

None of the known synchronization techniques is optimal in all cases. Despite the known weaknesses of locking and the many attempts to replace it, locking still predominates. There might still be hope for a "silver bullet", but until then, it would be constructive to also consider integration of different techniques in order to gain the benefit of their combined strengths. Such integration may involve using a *mixture* of objects which avoid locking (also called lockless objects) together with lock-based objects; and, as suggested in this paper, *fusing* lockless objects and locks together in order to create new interesting types of shared objects.

Acknowledgements: I wish to thank the two anonymous referees for their constructive suggestions.

References

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

[2] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. *Proceedings of the 19th International Symposium on Distributed Computing*, LNCS 3724, 122–136, 2005.

[3] M. K. Aguilera and S. Toueg. Timeliness-based wait-freedom: a gracefully degrading progress condition. In *Proc. 27rd ACM Symp. on Principles of Distributed Computing*, pages 305–314, 2008.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *CACM*, 8(9):569, 1965.

[6] W. B. Easton. Process synchronization without long-term interlock. In *Proc. of the 3rd ACM symp. on Operating systems principles*, pages 95–100, 1971.

[7] C. S. Ellis. Extendible hashing for concurrent operations and distributed data. In *Proc. of the 2nd ACM symposium on Principles of database systems*, pages 106–116, 1983.

[8] E. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. *Proc. of the 19th International Symp. on Distributed Computing*, LNCS 3724, pp. 78-92, 2005.

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

[10] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 50–59, 2004.

[11] R. Guerraoui, M. P. Herlihy and B. Pochon. Towards a theory of transactional contention managers. In *Proc. of the 24th Symp. on Principles of Dist. Computing*, pp. 258–264, 2005.

[12] R. Guerraoui, M. Kapalka and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.

[13] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th international symp. on distributed computing*, *LNCS* 2180:300–314, 2003.

[14] M. P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.

[15] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conf. on Dist. Computing Systems*, pages 522–529, 2003.

[16] M. P. Herlihy and J.E.B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the 20th annual international symp. on Computer architecture*, pages 289–300, 1993.

[17] T. E. Hart, P. E. McKenney, and A. D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Proc. of the 20th international Parallel and Distributed Processing Symp.*, 2006.

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

[19] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[20] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1-11, 1987.

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

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

[23] V. Luchangco, M. Moir and N. Shavit. On the uncontended complexity of consensus. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 45–59, 2003.

[24] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. on Database Systems*, 6(4):650–670, 1981.

[25] P. E. McKenney, M. M. Michael and J. Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. In *Proc. of the 4th workshop on Programming languages and operating systems*, pp. 1–5, 2007.

[26] M. Moir and J. Anderson. Wait-Free algorithms for fast, long-lived renaming, *Science of Computer Programming* 25(1):1–39, 1995.

[27] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[28] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[29] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Information and Computation* 233 (2013) 12–31. Also in: LNCS 1914, 164–178, DISC 2000.

[30] M. Merritt and G. Taubenfeld. Resilient consensus for infinitely many processes. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 1–15, 2003.

[31] S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Info. Processing Letters*, 26:141–151, 1987.

[32] M. Raynal. Algorithms for mutual exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.

[33] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer. ISBN 978-3-642-32027-9, 515 pages, 2013.

[34] R. Rajwar and J. R. Goodman, Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. 34th Inter. Symp. on Microarchitecture*, pp. 294–305, 2001.

[35] W. N. Scherer and M. L. Scott. Advanced Contention Management for dynamic software transactional memory. In *Proc. of the 24th Symp. on Principles of Dist. Computing*, pp. 240-248, 2005.

[36] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, 1995.

[37] H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *8th International Conference on Principles of Distributed Systems*, 2004.

[38] G. Taubenfeld. Efficient transformations of obstruction-free algorithms into non-blocking algorithms. *Proc. of the 21st International Symp. on Distributed Computing*, LNCS 4731, pp. 450–464, 2007.

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

[40] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd international symposium on distributed computing (DISC 2009)*, September 2009. *LNCS 5805* Springer Verlag 2009, 157–171.

[41] G. Taubenfeld. Waiting in Concurrent Algorithms. In *4th international conference on networked systems (NETYS 2016)*, Marrakech, Morocco, May 2016.

[42] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.

[43] J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.

[44] Transactional memory. For a list of citations see: http://www.cs.wisc.edu/trans-memory/.