Contention-sensitive Data Structures and Algorithms

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel,
`tgadi@idc.ac.il`,
`http://www.faculty.idc.ac.il/gadi/`

Abstract. A contention-sensitive data structure is a concurrent data structure in which the overhead introduced by locking is eliminated in the 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. Finally, we generalize the result to enable to avoid locking also when contention is low.

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

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.

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 and memory consuming.

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 the 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 most desired property in a lock implementation is that, in the absence of contention, a process can acquire the lock extremely fast. 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 can not 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. 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 $\perp$

$x[in_i] := 1$                                                    // start shortcut code
if $y = \perp$ then $y := in_i$ fi
if $x[1 - in_i] = 0$ then $out := in_i$; decide($in_i$) fi
if $out \neq \perp$ then decide($out$) fi                         // end shortcut code
$\boxed{\text{lock}}$ 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. Memory barriers may be used to prevent reordering [26].

1.4 Summery of contributions

The full list of our contributions is as follows,

1. We define 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 2).
2. 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 3).

3. Three known progress conditions are: (1) livelock-freedom, which guarantees that in the absence of process failures, *some* participating process makes progress; (2) starvation-freedom, which guarantees that in the absence of process failures, *every* participating progress makes progress; (3) obstruction-freedom, which guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" (i.e., do not take any steps) long enough. That is, obstruction-freedom guarantees progress for any process that eventually executes in isolation long enough. Under contention, obstruction-free data structures may suffer from livelocks. We presents three transformations:
   - Transformation 1, converts any contention-sensitive data structure which satisfies livelock-freedom into a corresponding contention-sensitive data structure which satisfies starvation-freedom. It adds only *one* memory reference to the shortcut code (Section 4.1).
   - Transformation 2, converts any obstruction-free data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom (Section 4.2).
   - A new progress condition called *prevention-freedom* is presented. Transformation 3, converts any prevention-free data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom (Section 4.3).
4. 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 transformation that converts any k-obstruction-free data structure into the corresponding k-contention-sensitive data structure which satisfies livelock-freedom (Section 5).
5. We present a contention-sensitive election algorithm, using atomic registers only (Section 6).

1.5 Related work

Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [6]. Since than, numerous implementations of locks have been proposed [34, 40]. Algorithms for several concurrent data structures based on locking have been proposed since at least the 1970's [5, 8, 20, 25]. Speculative lock elision [35], 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.

Implementations of data structures which avoid locking have appeared in many papers [7, 11, 14, 30, 38, 42]. 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 [15], non-blocking [19], and obstruction-freedom [16]. Wait-freedom guarantees that every process will always be able to complete its pending operations in a finite number of its own steps. Non-blocking guarantees that some process will always be able to complete its pending operations in a finite number of its own steps. All strategies that avoid locks are called lockless [18] or lock-free [29]. (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 and memory consuming. 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 [12, 36]. Other works investigated boosting obstruction-freedom by making timing assumption [4, 9, 39] and using failure detectors [13].

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 [10, 23]. 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, 24]. 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 [32].

Consistency conditions for concurrent objects are linearizability [19] and sequential consistency [22]. 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 [17, 37, 43]. It has implementations that use locks and others that avoid locking, but in both cases the complexity is hidden from the programmer. In [27], a constructive critique of locking and transactional memory: their strengths, weaknesses, and challenges, is presented.

2 Defining contention-sensitive data structures

We focus on an architecture in which n processes communicate asynchronously via a shared memory. 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 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. (It is possible to consider also using read-write locks, k-exclusion locks, etc.)

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. If it succeeds to complete the operation, it releases the acquired lock(s) and returns. 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 bounded 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 [22]. 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. 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.

3 A contention-sensitive double-ended queue data structure

In [16], 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 [16], is presented.

The double-ended queue is implemented on an infinite array (denoted Q) and is based on 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 process 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*.

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, it needs to modify in the array Q. 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 $Q[k-1]$ (without changing that location's value). If this SC succeeds, the process SC to $Q[k]$. As a result of this approach, two concurrent right-side 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 type 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. These properties can be easily satisfied by letting each process updating $Rptr$ (and $Lptr$) using (the more expensive) LL/SC/VL operations, whenever a process finds out that $Rptr$ is not updated. For lack of space, all the proofs were omitted.

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($value$)                                                              // $value \notin \{lnil, rnil\}$
1   $k := Rptr$; $prev :=$ LL($Q[k-1]$); $cur :=$ LL($Q[k]$);                   // $k$ index of leftmost $rnil$
2   if $cur = rnil \wedge prev \neq rnil$ 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   $\boxed{\text{lock}(rlock)}$
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 $cur = rnil \wedge prev \neq rnil$ 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  $\boxed{\text{unlock}(rlock)}$ ; return("ok")                               // unlocking section

rightpop()
1       $k := Rptr$; $prev :=$ LL($Q[k-1]$); $cur :=$ LL($Q[k]$)                // $k$ index of leftmost $rnil$
2       if $cur = rnil \wedge prev \neq rnil$ then                               // $Rptr$ is updated
3           if $prev = lnil \wedge$ VL($Q[k-1]$) then return("empty")            // adjacent $lnil$ and $rnil$
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   $\boxed{\text{lock}(rlock)}$
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 $cur = rnil \wedge prev \neq rnil$ then                               // $Rptr$ is updated
12          if $prev = lnil \wedge$ 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 ($done \vee empty$)
17  $\boxed{\text{unlock}(rlock)}$ ; if $done$ then return($prev$) else return("empty") fi   // unlocking section
```

Fig. 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.

4 Three transformations

Recall the question raised in the introduction: If a sequential data structure can not be used as the basic building block for constructing a contention-sensitive data structure, what is the best data structure to use? The following transformations that facilitate devising such data structures provide an answer.

4.1 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 [40] (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. The statement "**await** *condition*" is used as an abbreviation for "**while** ¬*condition* **do** *skip*".

Transformation 1: process i's program.
Initially: $flag[i] = false$, $contention = false$, the initial value of *turn* is immaterial.

```
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 $turn := (turn \bmod n) + 1$ 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.

4.2 From obstruction-freedom to livelock-freedom

Next we present a transformation that converts any 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.[1] The transformation uses a single lock.

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  $\boxed{\text{lock}}$                                                     // body
4  continue to execute steps of DS until op is completed
5  $\boxed{\text{unlock}}$
```

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.

A data structure which is constructed using the above transformation satisfies also the disable-free shortcut property and the weak-blocking body property.

4.3 From prevention-freedom to livelock-freedom

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.

> **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 its own 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 crash 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

[1] In simple data structures like a queue or a stack the number of *first*(*DS*) steps would be a constant. In a data structure like a search tree the number would depend on the size or depth of the tree; this value can be stored in a shared location that each process can read and update.

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.

Let *DS* be a data structure that is *prevention-free* w.r.t. some set $\{G_1, ..., G_n\}$. We say that *DS* is *exit-safe* if, regardless of contention, it is always the case that after a process invokes an operation of *DS* and takes *first*(*DS*) steps, either the process completes its operation or the process can always continue taking a small number of additional steps until it reaches a gate. Below we present a transformation which converts any prevention-free exit-safe data structure, denoted *DS*, into a corresponding contention-sensitive data structure. 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.

5 Generalizations

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 follows: 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. We demonstrate this idea, by presenting a 2-contention-sensitive consensus algorithm. The algorithm uses atomic registers and a single swap object.[2]

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 $\boxed{\text{lock}}$ if $out = \perp$ then $out := y$ fi $\boxed{\text{unlock}}$ ; decide($out$)   // locking
```

[2] A swap operation takes a shared registers and a local register, and atomically exchange their values. It is known that there is no wait-free consensus algorithm for more than two processes, using atomic registers and atomic swap objects [15].

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 $|P|$ denotes 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 its own 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.

> **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.[3] Once it acquires the lock it "enters" the *DS* code at the same point where 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 its own 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.

6 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

[3] 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.

elected leader. Processes are not required to participate, however, once a process starts participating it is guaranteed that it will not fail. It is known that in the presence of one crash failure, it is not possible to solve election using atomic registers only [33, 41].

The following algorithm solves the election problem for any number of processes, and is related to the splitter constructs from [21, 28, 31]. A single lock is used. It is assumed that after a process executes a **decide**() statement, it immediately terminates.

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
3   $y := 1$
4   if $x = i$ then $z := i$; if $b = 0$ then decide(1) fi fi       // I am the leader!
                                                                    // end shortcut
5   $\boxed{\text{lock}}$                                           // locking
6   if $z = i \wedge done = 0$ then $leader = 1$                    // I am the leader!
7       else await $b \neq 0 \vee z \neq 0$
8           if $z = 0 \wedge done = 0$ then $leader = 1$; $done := 1$   // I am the leader!
9               else $leader = 0$                                   // I am not the leader
10          fi
11  fi
12  $\boxed{\text{unlock}}$; 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.

7 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.

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. R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.
4. 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.
5. R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.
6. E. W. Dijkstra. Solution of a problem in concurrent programming control. *CACM*, 8(9):569, 1965.
7. W. B. Easton. Process synchronization without long-term interlock. In *Proc. of the 3rd ACM symp. on Operating systems principles*, pages 95–100, 1971.
8. 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.
9. 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.
10. M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
11. 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.
12. 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.
13. R. Guerraoui, M. Kapalka and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.
14. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. 15th international symp. on distributed computing*, *LNCS* 2180:300–314, 2003.
15. M. P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.
16. 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.
17. 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.
18. 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.
19. 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.
20. H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.
21. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1-11, 1987.

22. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, September 1979.
23. 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.
24. 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.
25. 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.
26. P. E. McKenney. Memory ordering in modern microprocessors, Part I & Part II. *Linux Journal*, 2005(136) 2 pages, and 2005(137) 5 pages, 2005. (Revised April 2009.)
27. 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.
28. M. Moir and J. Anderson. Wait-Free algorithms for fast, long-lived renaming, *Science of Computer Programming* 25(1):1–39, 1995.
29. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.
30. 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.
31. M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Proc. of the 14th International Symp. on Distributed Computing*, LNCS 1914, 164–178, 2000.
32. 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.
33. S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Info. Processing Letters*, 26:141–151, 1987.
34. M. Raynal. Algorithms for mutual exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.
35. 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.
36. 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.
37. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, 1995.
38. 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.
39. 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.
40. G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.
41. G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.
42. J. D. Valois. Implementing lock-free queues. In *Proc. of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 212–222, 1994.
43. Transactional memory. For a list of citations see: http://www.cs.wisc.edu/trans-memory/.