A Closer Look at Concurrent Data Structures and Algorithms

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

Abstract

In this survey article, I will present three ideas, regarding the construction of concurrent data structures and algorithms, recently published in [27, 28, 29].

The first idea is that of *contention-sensitivity*. 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. This notion is formally defined, several contention-sensitive data structures are presented, and transformations that facilitate devising such data structures are discussed.

The second idea is the introduction of a new synchronization problem, called *fair synchronization*. Solving the new problem enables to automatically add strong fairness guarantees to existing implementations of concurrent data structures, without using locks, and to transform any solution to the mutual exclusion problem into a fair solution.

The third idea is a generalization of the traditional notion of *fault tolerance*. Important classical problems have no asynchronous solutions which can tolerate even a single fault. It is shown that while some of these problems have solutions which guarantee that in the presence of *any* number of faults *most* of the correct processes will terminate, other problems do not even have solutions which guarantee that in the presence of just *one* fault at least one correct process terminates.

1 Introduction

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.

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. This motivates attempts to design data structures that avoid locking.

The advantages of concurrent data structures and algorithms which completely avoid locking 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 consider an intermediate approach for the design of concurrent data structures. 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 [27]. In Section 2, the notion of contention-sensitivity is formally defined, several contention-sensitive data structures are presented, and transformations that facilitate devising such data structures are discussed.

Most published concurrent data structures which completely avoid locking do not provide any fairness guarantees. That is, they allow processes to access a data structure and complete their operations arbitrarily many times before some other trying process can complete a single operation. In Section 3, we show how to automatically transfer a given concurrent data structure which avoids locking and waiting into a similar data structure which satisfies a strong fairness requirement, without using locks and with limited waiting. To achieve this goal, we introduce and solve a *new* synchronization problem, called *fair synchronization* [29]. Solving the new problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any lock (i.e., a solution to the mutual exclusion problem) into a fair lock.

In Section 4, we generalize the traditional notion of fault tolerance in a way which enables to capture more sensitive information about the resiliency of a con-

current algorithm. Intuitively, an algorithm that, in the presence of any number of faults, always guarantees that all the correct processes, except maybe one, successfully terminate (their operations), is more resilient to faults than an algorithm that in the presence of a single fault does not even guarantee that a single correct process ever terminates. However, according to the standard notion of fault tolerance both algorithms are classified as algorithms that can not tolerate a single fault. Our general definition distinguishes between these two cases.

It is well known that, in an asynchronous system where processes communicate by reading and writing atomic read/write registers important classical problems such as, consensus, set-consensus, election, perfect renaming, implementations of a test-and-set bit, a shared stack, a swap object and a fetch-and-add object, have no deterministic solutions which can tolerate even a single fault. In Section 4, we show that while, some of these classical problems have solutions which guarantee that in the presence of *any* number of faults most of the correct processes will successfully terminate; other problems do not even have solutions which guarantee that in the presence of just *one* fault at least one correct process terminates.

Our model of computation consists of an asynchronous collection of n processes which 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. In most cases, we assume that processes communicate by reading and writing atomic registers. By atomic registers we always mean atomic read/write registers. In few cases, we will also consider stronger synchronization primitives. 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.

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

2 Contention-sensitive Data Structures

2.1 Motivation

As already mentioned in Section 1, using locks may, in various scenarios, degrade the performance of concurrent data structures. On the other hand, concurrent data structures which completely avoid locking 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. 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.

2.2 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 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 (which does not involve locking), by presenting a contention-sensitive solution to the binary consensus problem using atomic read/write registers and a single lock.

2.3 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 when 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. We assume that processes communicate via atomic registers. 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 lock **if** $out = \perp$ **then** $out := y$ **fi** 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.

2.4 Progress conditions

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.

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 [8]. *Non-blocking* (which is sometimes called lock-freedom) guarantees that *some* active process will always be able to complete its pending operations in a finite number of steps [13]. *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 [9]. 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.

2.5 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. 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 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.

In [27], several additional desirable properties are defined.

2.6 An example: Contention-sensitive election

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. 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 only atomic read/write registers [18, 31].

The following algorithm solves the election problem for any number of processes, and is related to the splitter constructs from [14, 16, 17]. 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   lock                                                          // locking
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.

2.7 Additional results

The following additional results are presented in [27].

- 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

- Transformations that facilitate devising contention-sensitive data structures. The first transformation converts any contention-sensitive data structure which satisfies livelock-freedom into a corresponding contention-sensitive data structure which satisfies starvation-freedom. The second transformation, converts any obstruction-free data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom.

- Finally, the notion of a *k-contention-sensitive* data structure in which locks are used only when contention goes above k is presented. This notion is illustrated by implementing a 2-contention-sensitive consensus algorithm. Then, for each $k \geq 1$, a progress condition, called *k-obstruction-freedom*, is defined and a transformation is presented that converts any k-obstruction-free data structure into the corresponding k-contention-sensitive data structure which satisfies livelock-freedom.

3 Fair Synchronization

3.1 Motivation

As already discussed in Section 1, using locks may degrade the performance of synchronized concurrent applications, and hence much work has been done on the design of wait-free and non-blocking data structures. However, the wait-freedom and non-blocking progress conditions do not provide fairness guarantees. That is, such data structures may allow processes to complete their operations arbitrarily many times before some other trying process can complete a single operation. Such a behavior may be prevented when fairness is required. However, fairness requires waiting or helping.

Using helping techniques (without waiting) may impose too much overhead upon the implementation, and are often complex and memory consuming. Does it mean that for enforcing fairness it is best to use locks? The answer is negative. We show how any wait-free and any non-blocking implementation can be automatically transformed into an implementation which satisfies a very strong fairness requirement without using locks and with limited waiting.

We require that no beginning process can complete two operations on a given resource while some other process is kept waiting on the same resource. Our approach, allows as many processes as possible to access a shared resource at the same time as long as fairness is preserved. To achieve this goal, we introduce and solve a new synchronization problem, called *fair synchronization*. Solving the fair synchronization problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any solution to the mutual exclusion problem into a fair solution.

3.2 The fair synchronization problem

The fair synchronization problem is to design an algorithm that guarantees fair access to a shared resource among a number of participating processes. Fair access means that no process can access a resource twice while some other trying process cannot complete a single operation on that resource. There is no a priori limit (smaller than the number of processes n) on the number of processes that can access a resource simultaneously. In fact, a desired property is that as many processes as possible will be able to access a resource at the same time as long as fairness is preserved.

It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, fair and exit. Furthermore, it is assumed that the entry section consists of two parts. The first part, which is called the *doorway*, is *fast wait-free*:

its execution requires only a (very small) *constant* number of steps and hence always terminates; the second part is the *waiting* part which includes (at least one) loop with one or more statements. Like in the case of the doorway, the exit section is also required to be fast wait-free. A *waiting* process is a process that has finished its doorway code and reached the waiting part of its entry section. A *beginning* process is a process that is about to start executing its entry section.

A process is *enabled* to enter its fair section at some point in time, if sufficiently many steps of that process will carry it into the fair section, independently of the actions of the other processes. That is, an enabled process does not need to wait for an action by any other process in order to complete its entry section and to enter its fair section, nor can an action by any other process prevent it from doing so.

The **fair synchronization problem** is to write the code for the entry and the exit sections in such a way that the following three basic requirements are satisfied.

- **Progress:** *In the absence of process failures and assuming that a process always leaves its fair section, if a process is trying to enter its fair section, then some process, not necessarily the same one, eventually enters its fair section.*

  The terms deadlock-freedom and livelock-freedom are used in the literature for the above progress condition, in the context of the mutual exclusion problem.

- **Fairness:** *A beginning process cannot execute its fair section twice before a waiting process completes executing its fair and exit sections once. Furthermore, no beginning process can become enabled before an already waiting process becomes enabled.*

  It is possible that a beginning process and a waiting process will become enabled at the same time. However, no beginning process can execute its fair section twice while some other process is kept waiting. The second part of the fairness requirement is called *first-in-first-enabled*. The term *first-in-first-out* (FIFO) fairness is used in the literature for a slightly stronger condition which guarantees that: no beginning process can pass an already waiting process. That is, no beginning process can enter its fair section before an already waiting process does so.

- **Concurrency:** *All the waiting processes which are not enabled become enabled at the same time.*

It follows from the *progress* and *fairness* requirements that *all* the waiting processes which are not enabled will eventually become enabled. The concurrency requirement guarantees that becoming enabled happens simultaneously, for all the waiting processes, and thus it guarantees that many processes will be able to access their fair sections at the same time as long as fairness is preserved. We notice that no lock implementation may satisfy the concurrency requirement.

The processes that have already passed through their doorway can be divided into two groups. The enabled processes and those that are not enabled. It is not possible to always have all the processes enabled due to the fairness requirement. All the enabled processes can immediately proceed to execute their fair sections. The waiting processes which are not enabled will eventually simultaneously become enabled, before or once the currently enabled processes exit their fair and exit sections. We observe that the stronger FIFO fairness requirement, the progress requirement and concurrency requirement cannot be mutually satisfied.

3.3 The fair synchronization algorithm

We use one atomic bit, called *group*. The first thing that process i does in its entry section is to read the value of the *group* bit, and to determine to which of the two groups (0 or 1) it should belong. This is done by setting i's single-writer register $state_i$ to the value read.

Once i chooses a group, it waits until its group has priority over the other group and then it enters its fair section. The order in which processes can enter their fair sections is defined as follows: If two processes belong to different groups, the process whose group, as recorded in its *state* register, is *different* from the value of the bit *group* is enabled and can enter its fair section, and the other process has to wait. If all the active processes belong to the same group then they can all enter their fair sections.

Next, we explain when the shared *group* bit is updated. The first thing that process i does when it leaves its fair section (i.e., its first step in its exit section) is to set the *group* bit to a value which is *different* from the value of its $state_i$ register. This way, i gives priority to waiting processes which belong to the same group that it belongs to.

Until the value of the *group* bit is first changed, all the active processes belong to the same group, say group 0. The first process to finish its fair section flips the value of the *group* bit and sets it to 1. Thereafter, the value read by all the new beginning processes is 1, until the group bit is modified again. Next, *all* the processes which belong to group 0 enter and then exit their fair sections possibly at the same time until there are no active processes which belong to group 0. Then

all the processes from group 1 become enabled and are allowed to enter their fair sections, and when each one of them exits it sets to 0 the value of the *group* bit, which gives priority to the processes in group 1, and so on.

The following registers are used: (1) a single multi-writer atomic bit named *group*, (2) an array of single-writer atomic registers *state*[1..*n*] which range over $\{0, 1, 2, 3\}$. To improve readability, we use below subscripts to index entries in an array. At any given time, process i can be in one of four possible states, as recorded in it single-writer register $state_i$. When $state_i = 3$, process i is not active, that is, it is in its remainder section. When $state_i = 2$, process i is active and (by reading *group*) tries to decide to which of the two groups, 0 or 1, it should belong. When $state_i = 1$, process i is active and belongs to group 1. When $state_i = 0$, process i is active and belongs to group 0.

The statement **await** *condition* is used as an abbreviation for **while** ¬*condition* **do** *skip*. The *break* statement, like in C, breaks out of the smallest enclosing *for* or *while* loop. Finally, whenever two atomic registers appear in the same statement, two separate steps are required to execute this statement. The algorithm is given below.[1]

A Fair Synchronization Algorithm: `process i's code` ($1 \le i \le n$)

Shared variables:

group: atomic bit; the initial value of the group bit is immaterial.
state[1..*n*]: array of atomic registers, which range over $\{0, 1, 2, 3\}$
Initially $\forall i : 1 \le i \le n : state_i = 3$ `/* processes are inactive */`

```
1   state_i := 2                                          /* begin doorway */
2   state_i := group                          /* choose group and end doorway */
3   for j = 1 to n do                                     /* begin waiting */
4       if (state_i ≠ group) then break fi          /* process is enabled */
5       await state_j ≠ 2
6       if state_j = 1 − state_i                      /* different groups */
7       then await   (state_j ≠ 1 − state_i) ∨ (state_i ≠ group) fi
8   od                                                      /* end waiting */
9   fair section
10  group := 1 − state_i                                    /* begin exit */
11  state_i := 3                                              /* end exit */
```

In line 1, process i indicates that it has started executing its doorway code. Then,

[1] To simplify the presentation, when the code for a fair synchronization algorithm is presented, only the entry and exit codes are described, and the remainder code and the infinite loop within which these codes reside are omitted.

in *two* atomic steps, it reads the value of *group* and assigns the value read to $state_i$ (line 2).

After passing its doorway, process i waits in the *for loop* (lines 3–8), until all the processes in the group to which it belongs are simultaneously enabled and then it enters its fair section. This happens when either, ($state_i \neq group$), i.e. the value the *group* bit points to the group which i does *not* belong to (line 4), or when all the waiting processes (including i) belong to the same group (line 7). Each one of the terms of the await statement (line 7) is evaluated separately. In case processes i and j belong to different groups (line 6), i waits until either (1) j is not competing any more or j has reentered its entry section, or (2) i has priority over j because $state_i$ is *different* than the value of the *group* bit.

In the exit code, i sets the *group* bit to a value which is different than the group to which it belongs (line 10), and changes its state to not active (line 11). We notice that the algorithm is also correct when we replace the order of lines 9 and 10, allowing process i to write the group bit immediately before it enters its fair section. The order of lines 10 and 11 is crucial for correctness.

We observe that a *non* beginning process, say p, may enter its fair section ahead of another waiting process, say q, twice: the first time if p is enabled on the other group, and the second time if p just happened to pass q which is waiting on the same group and enters its fair section first. We point out that omitting lines 1 and 5 will result in an incorrect solution. It is possible to replace each one of the 4-valued single-writer atomic registers, by three *separate* atomic bits.

An *adaptive* algorithm is an algorithm which its time complexity is a function of the actual number of participating processes rather than a function of the total number of processes. An adaptive fair synchronization algorithm using atomic register is presented in [29]. It is also shown in [29] that $n - 1$ read/write registers and conditional objects are necessary for solving the fair synchronization problem for n processes. A conditional operation is an operation that changes the value of an object only if the object has a particular value. A *conditional object* is an object that supports only conditional operations. Compare-and-swap and test-and-set are examples of conditional objects.

3.4 Fair data structures and fair locks

In order to impose fairness on a concurrent data structure, concurrent accesses to a data structure can be synchronized using a fair synchronization algorithm: a process accesses the data structure only inside a fair section. Any data structure can be easily made fair using such an approach, without using locks and with limited waiting. The formal definition of a fair data structure can be found in [29].

We name a solution to the fair synchronization problem a (finger) *ring*.[2] Using a single *ring* to enforce fairness on a concurrent data structure, is an example of coarse-grained *fair* synchronization. In contrast, fine-grained *fair* synchronization enables to protect “small pieces" of a data structure, allowing several processes with *different* operations to access it completely independently. For example, in the case of adding fairness to an existing wait-free queue, it makes sense to use two rings: one for the enqueue operations and the other for the dequeue operations.

We assume the reader is familiar with the definition of a deadlock-free mutual exclusion algorithm (DF-ME). By composing a fair synchronization algorithm (FS) and a DF-ME, it is possible to construct a *fair* mutual exclusion algorithm (FME), i.e., a fair lock. The entry section of the composed FME algorithm consists of the entry section of the FS algorithm followed by the entry section of the ME algorithm. The exit section of the FME algorithm consists of the exit section of the ME algorithm followed by the exit section of the FS algorithm. The doorway of the FME algorithm is the doorway of the FS algorithm.

4 Fault Tolerance

4.1 Motivation

According to the standard notion of fault tolerance, an algorithm is t-resilient if in the presence of up to t faults, *all* the correct processes can still complete their operations and terminate. Thus, an algorithm is *not* t-resilient, if as a result of t faults there is *some* correct process that can not terminate. This traditional notion of fault tolerance is not sensitive to the *number* of correct processes that may or may not complete their operations as a result of the failure of other processes.

Consider for example the renaming problem, which allows processes, with distinct initial names from a large name space, to get distinct new names from a small name space. A renaming algorithm that, in the presence of any number of faults, always guarantees that *most* of the correct processes, but not necessarily all, get distinct new names is clearly more resilient than a renaming algorithm that in the presence of a single fault does not guarantee that even one correct process ever gets a new name. However, using the standard notion of fault tolerance, it is not possible to compare the resiliency of such algorithms – as both are simply not even 1-resilient. This motivates us to suggest and investigate a more general notion of fault tolerance.

We generalize the traditional notion of fault tolerance by allowing a limited number participating correct processes not to terminate in the presence of faults.

[2] Many processes can simultaneously pass through the ring’s hole, but the size of the ring may limit their number.

Every process that do terminate is required to return a correct result. Thus, our definition guarantees safety but may sacrifice liveness (termination), for a limited number of processes, in the presence of faults. The consequences of violating liveness are often less severe than those of violating safety. In fact, there are systems that can detect and abort processes that run for too long. Sacrificing liveness of few processes allows us to increase the resiliency of the whole system.

4.2 A general definition of fault tolerance

For the rest of the section, n denotes the number of processes, t denotes the number of faulty processes, and $N = \{0, 1, ..., n\}$.

> **Definition:** For a given function $f : N \to N$, an algorithm is (t, f)-*resilient* if in the presence of t' faults at most $f(t')$ participating correct processes may *not* terminate their operations, for every $0 \leq t' \leq t$.

It seems that (t, f)-*resiliency* is interesting only when requiring that $f(0) = 0$. That is, in the absence of faults all the participating processes must terminate their operations. The standard definition of t-resiliency is equivalent to (t, f)-resiliency where $f(t') = 0$ for every $0 \leq t' \leq t$. Thus, the familiar notion of *wait-freedom* is equivalent to $(n-1, f)$-resiliency where $f(t') = 0$ for every $0 \leq t' \leq n-1$. The new notion of (t, f)-resiliency is quite general, and in this section we focus mainly on the following three levels of resiliency.

- An algorithm is *almost-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = 1$, for every $1 \leq t' \leq t$. Thus, in the presence of any number of up to t faults, all the correct participating processes, except maybe one process, must terminate their operations.

- An algorithm is *partially-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = t'$, for every $1 \leq t' \leq t$. Thus, in the presence of any number $t' \leq t$ faults, all the correct participating processes, except maybe t' of them must terminate their operations.

- An algorithm is *weakly-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$, and in the presence of any number of up to $t \geq 1$ faults, if there are *two* or more correct participating processes then one correct participating process must terminate its operation. (Notice that for $n = 2$, if one process fails the other one is not required to terminate.)

For $n \geq 3$ and $t < n/2$, the notion of weakly-t-resiliency is strictly weaker than the notion of partially-t-resiliency. For $n \geq 3$, the notion of weakly-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For $n \geq 3$ and $t \geq 2$,

the notion of partially-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For all n, partially-1-resiliency and almost-1-resiliency are equivalent. For $n = 2$, these three notions are equivalent. We say that an algorithm is *almost-wait-free* if it is *almost*-$(n-1)$-*resilient*, i.e., in the presence of any number of faults, all the participating correct processes, except maybe one process, must terminate. We say that an algorithm is *partially-wait-free* if it is *partially*-$(n-1)$-*resilient*, i.e., in the presence of any number of $t \leq n-1$ faults, all the correct participating processes, except maybe t of them must terminate.

4.3 Example: An almost-wait-free symmetric test-and-set bit

A test-and-set bit supports two atomic operations, called *test-and-set* and *reset*. A test-and-set operation takes as argument a shared bit b, assigns the value 1 to b, and returns the previous value of b (which can be either 0 or 1). A reset operation takes as argument a shared bit b and writes the value 0 into b.

The *sequential specification* of an object specifies how the object behaves in sequential runs, that is, in runs when its operations are applied sequentially. The sequential specification of a test-and-set bit is quite simple. In sequential runs, the first test-and-set operation returns 0, a test-and-set operation that happens immediately after a reset operation also returns 0, and all other test-and-set operations return 1. The consistency requirement is linearizability.

The algorithm below is for n processes each with a unique identifier taken from some (possibly infinite) set which does not include 0. It makes use of exactly n registers which are long enough to store a process identifier and one atomic bit. The algorithm is based on the symmetric mutual exclusion algorithm from [24].[3]

The algorithm uses a register, called *turn*, to indicate who has priority to return 1, $n-1$ *lock* registers to ensure that at most one process will return 1 between resets, and a bit, called *winner*, to indicate whether some process already returned 1. Initially the values of all these shared registers are 0. In addition, each process has a private boolean variable called *locked*. We denote by *b.turn*, *b.winner* and *b.lock*[*] the shared registers for the implementation of a specific test-and-set bit, named b.

[3] A symmetric algorithm is an algorithm in which the only way for distinguishing processes is by comparing identifiers, which are unique. Identifiers can be written, read and compared, but there is no way of looking inside any identifier. Thus, identifiers cannot be used to index shared registers.

AN ALMOST-WAIT-FREE SYMMETRIC TEST-AND-SET BIT: `process p's program.`

```
function test-and-set (b:bit) return:value in {0, 1};                    /* access bit b */
1    if b.turn ≠ 0 then return(0) fi;                                          /* lost */
2    b.turn := p;
3    repeat
4            for j := 1 to n − 1 do                                        /* get locks */
5                if b.lock[j] = 0 then b.lock[j] := p fi od
6            locked := 1;
7            for j := 1 to n − 1 do                                   /* have all locks? */
8                if b.lock[j] ≠ p then locked := 0 fi od;
9    until b.turn ≠ p or locked = 1 or b.winner = 1;
10   if b.turn ≠ p or b.winner = 1 then
11           for j := 1 to n − 1 do                              /* lost, release locks */
12               if b.lock[j] = p then b.lock[j] := 0 fi od
13           return(0) fi;
14   b.winner := 1; return(1).                                             /* wins */
end_function

function reset (b:bit);                                                  /* access bit b */
1    b.winner := 0; b.turn := 0;                                       /* release locks */
2    for j := 1 to n − 1 do
3            if b.lock[j] = p then b.lock[j] := 0 fi od.
end_function
```

In the test-and-set operation, a process, say p, initially checks whether $b.turn \neq 0$, and if so returns 0. Otherwise, p takes priority by setting $b.turn$ to p, and attempts to obtain all the $n-1$ locks by setting them to p. This prevents other processes that also saw $b.turn = 0$ and set $b.turn$ to their ids from entering. That is, if p obtains all the locks before the other processes set $b.turn$, they will not be able to get any of the locks since the values of the locks are not 0. Otherwise, if p sees $b.turn \neq p$ or $b.winner = 1$, it will release the locks it holds, allowing some other process to proceed, and will return 0. In the reset operation, p sets $b.turn$ to 0, so the other processes can proceed, and releases all the locks it currently holds. In [28], it is shown that even in the absence of faults, any implementation of a test-and-set bit for n processes from atomic read/write registers must use at least n such registers.

4.4 Additional results

The following additional results are presented in [28].

Election. It is known that there is no 1-resilient election algorithm using atomic registers [18, 31]. It is shown that:

> *There is an almost-wait-free symmetric election algorithm using* $\lceil \log n \rceil +$ 2 *atomic registers.*

The known space lower bound for election in the absence of faults is $\lceil \log n \rceil + 1$ atomic registers [24].

Perfect Renaming. A *perfect* renaming algorithm allows n processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ...n\}$. A *one-shot* renaming algorithm allows each process to acquire a distinct new name just once. A *long-lived* renaming algorithm allows processes to repeatedly acquire distinct names and release them. It is shown that:

> (1) *There is a partially-wait-free symmetric one-shot perfect renaming algorithm using* (*a*) $n - 1$ *almost-wait-free election objects, or* (*b*) $O(n \log n)$ *registers.* (2) *There is a partially-wait-free symmetric long-lived perfect renaming algorithm using either* $n - 1$ *almost-wait-free test-and-set bits.*

It is known that in asynchronous systems where processes communicate by atomic registers there is no 1-resilient perfect renaming algorithm [18, 31].

Fetch-and-add, swap, stack. A *fetch-and-add* object supports an operation which takes as arguments a shared register r, and a value *val*. The value of r is incremented by *val*, and the old value of r is returned. A *swap* object supports an operation which takes as arguments a shared registers and a local register and atomically exchange their values. A shared stack is a linearizable object that supports push and pop operations, by several processes, with the usual stack semantics. It is shown that:

> *There are partially-wait-free implementations of a fetch-and-add object, a swap object, and a stack object using atomic registers.*

The result complements the results that in asynchronous systems where processes communicate using registers there are no 2-resilient implementations of fetch-and-add, swap, and stack objects [8].

Consensus and Set-consensus. The *k-set consensus* problem is to find a solution for n processes, where each process starts with an input value from some domain, and must choose some participating process' input as its output. All n processes together may choose no more than k distinct output values. The 1-set consensus problem, is the familiar consensus problem. It is shown that:

(1) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using either atomic registers or sending and receiving messages. In particular, for $n \geq 3$, there is no weakly-1-resilient consensus algorithm using either atomic registers or messages.* (2) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using almost-wait-free test-and-set bits and atomic registers.*

These results strengthen the know results that, in asynchronous systems where processes communicate either by atomic registers or by sending and receiving messages, there is no 1-resilient consensus algorithm [7, 15], and there is no k-resilient k-set-consensus algorithm [3, 11, 22].

5 Related Work

All the ideas and results presented in this survey are from [27, 28, 29]. Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [5]. Since than, numerous implementations of locks have been proposed [20, 25]. The fair synchronization algorithm, presented in Section 3.3, uses some ideas from the mutual exclusion algorithm presented in [26].

Algorithms for several concurrent data structures based on locking have been proposed since at least the 1970's [2]. Speculative lock elision [21], 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.

The benefits of avoiding locking has already been considered in [6]. There are many implementations of data structures which avoid locking [12, 19, 25]. 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 [8], non-blocking [13], and obstruction-freedom [9]. Progress conditions, called k-waiting, for $k \geq 0$, which capture the "amount of waiting" of processes in asynchronous concurrent algorithm, are introduced in [30].

Extensions of the notion of fault tolerance, which are different from those considered in Section 4, were proposed in[4], where a precise way is presented to characterize adversaries by introducing the notion of disagreement power: the biggest integer k for which the adversary can prevent processes from agreeing on k values when using registers only; and it is shown how to compute the disagreement power of an adversary.

Linearizability is defined in [13]. 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 [10, 12, 23]. It has implementations that use locks and others that avoid locking, but in both cases the complexity is hidden from the programmer.

6 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 together with lock-based objects; and, as suggested in Section 2, *fusing* lockless objects and locks together in order to create new interesting types of shared objects.

In Section 3, we have proposed to enforce fairness as a wrapper around a concurrent data structure, and studied the consequences. We have formalized the fair synchronization problem, presented a solution, and then showed that existing concurrent data structures and mutual exclusion algorithms can be encapsulated into a fair synchronization construct to yield algorithms that are inherently fair. Since many processes may enter their fair sections simultaneously, it is expected that using fair synchronization algorithms will not degrade the performance of concurrent applications as much as locks. However, as in the case of using locks, slow or stopped processes may prevent other processes from ever accessing their fair sections.

Finally, in Section 4, we have refined the traditional notion of *t-resiliency* by defining the finer grained notion of (t, f)-*resiliency*. Rather surprisingly, while some problems, that have no solutions which can tolerate even a single fault, do have solutions which satisfy almost-wait-freedom, other problems do not even have weakly-1-resilient solutions.

References

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

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

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

[4] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmanns. The disagreement power of an adversary. In *Proc. 28th ACM Symp. on Principles of Distributed Computing*, pages 288–289, 2009.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 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] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

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

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

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

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

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

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *toplas*, 12(3):463–492, 1990.

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

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

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

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

[18] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

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

[20] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[21] 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.

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

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

[24] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

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

[26] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.

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

[28] G. Taubenfeld. A closer look at fault tolerance. In *Proc. 31st ACM Symp. on Principles of Distributed Computing*, pages 261–270, 2012.

[29] G. Taubenfeld. Fair synchronization. In *27th international symposium on distributed computing*, October 2013. *LNCS 8205* Springer Verlag 2013, 179–193.

[30] G. Taubenfeld. Waiting without locking. Unpublished manuscript, 2014.

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