Efficient Transformations of Obstruction-Free Algorithms into Non-blocking 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. Three well studied progress conditions for implementing concurrent algorithms without locking are, obstruction-freedom, non-blocking and wait-freedom. Obstruction-freedom is weaker than non-blocking which, in turn, is weaker than wait-freedom. While obstruction-freedom and non-blocking have the potential to significantly improve the performance of concurrent applications, wait-freedom (although desirable) imposes too much overhead upon the implementation.

In [5], Fich, Luchangco, Moir, and Shavit have presented an interesting transformation that converts any obstruction-free algorithm into a wait-free algorithm when analyzed in the unknown-bound semi-synchronous model. The FLMS transformation uses n atomic single-writer registers, n atomic multi-writer registers and a single fetch-and-increment object, where n is the number of processes.

We define a time complexity measure for analyzing such transformations, and prove that the time complexity of the FLMS transformation is *exponential* in the number of processes n. This leads naturally to the question of whether the time and/or space complexity of the FLMS transformation can be improved by relaxing the wait-freedom progress condition. We present several efficient transformations that convert any obstruction-free algorithm into a non-blocking algorithm when analyzed in the unknown-bound semi-synchronous model. All our transformations have $O(1)$ time complexity. One transformation uses n atomic single-writer registers and a single compare-and-swap object; another transformation uses only a single compare-and-swap object which is assumed to support also a read operation.

1 Introduction

1.1 Motivation

Three well studied progress conditions for implementing concurrent algorithms without locking are, obstruction-freedom, non-blocking and wait-freedom. An algorithm is *wait-free* if it guarantees that *every* process will always be able to complete its pending operations in a finite number of its own steps. An algorithm is *non-blocking* if it guarantees that *some* process will always be able to complete its pending operation in a finite number of its own steps. An algorithm is *obstruction-free* if it 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" long enough (that is, in the absence of interference from other processes).

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 450–464, 2007.

Clearly, obstruction-freedom is weaker than non-blocking which, in turn, is weaker than wait-freedom. The term *lock-free* algorithms refers to algorithms that do not use locking in any way. Wait-free, non-blocking and obstruction-free algorithms are by definition lock-free algorithms.[1] Advantages of using lock-free algorithms are that they are not subject to deadlocks or priority inversion, they are resilient to process crash failures (no data corruption on process failure), and they do not suffer significant performance degradation from scheduling preemption, page faults or cache misses.

While non-blocking and obstruction-freedom have the potential to significantly improve the performance of concurrent applications, and can be used in place of using locks in various cases, wait-free synchronization (although desirable) imposes too much overhead upon the implementation. Wait-free algorithms are often very complex and memory consuming, and hence considered less practical than non-blocking algorithms. Furthermore, starvation can be sometimes efficiently handled by collision avoidance techniques such as exponential backoff.

Requiring implementations to satisfy only obstruction-freedom can significantly simplify the design of concurrent algorithms, as it eliminates the need to ensure progress under contention. However, since obstruction-free algorithms do not guarantee progress under contention, they may suffer from livelocks. Various contention management techniques have been proposed to efficiently improve progress of obstruction-free algorithms under contention. Existing lock-free contention managers, which allow processes to run without interference long enough until they can complete their operations, do not provide full guarantee to ensure progress in all cases.

While obstruction-free algorithms are easier to design and are efficient in various cases, it is most desirable that a lock-free implementation do satisfy the stronger non-blocking progress condition. Hence the importance of designing efficient transformations that automatically convert any obstruction-free algorithm into a non-blocking algorithm. Such transformations should not affect the behaviour of the original (obstruction-free) algorithm in uncontended cases, or in executions where the contention management technique used is effective.

The focus of this paper is on the design of such transformations in the unknown-bound semi-synchronous model, where it is assumed that there is an *unknown* upper bound on memory access time. All practical systems satisfy the unknown-bound assumption.

1.2 Results

In [5], Fich, Luchangco, Moir, and Shavit have presented an interesting transformation that converts any obstruction-free algorithm into a wait-free algorithm when analyzed in the unknown-bound semi-synchronous model. The FLMS transforma-

[1] In the literature, the terms lock-free and non-blocking are sometimes used as synonymous, or even with opposite meaning to the way they are defined here. As suggested in [16], it is useful to distinguish between algorithms that do not require locking (i.e., lock-free algorithms) and those that actually satisfy the non-blocking progress condition.

tion uses n atomic single-writer registers, n atomic multi-writer registers and a single fetch-and-increment object.

We start by defining a time complexity measure for analyzing such transformations, and prove that the time complexity of the FLMS transformation is *exponential* in the number of processes n. Then, we present several efficient transformations that convert any obstruction-free algorithm into a non-blocking algorithm when analyzed in the unknown-bound semi-synchronous model.

All our transformations have $O(1)$ time complexity. One of transformation uses n atomic single-writer registers and a single compare-and-swap object; another transformation uses only a single compare-and-swap object which is assumed to support also a read operation.

1.3 Related Work

Our work is based on the transformation presented in [5]. A comprehensive discussion of wait-free synchronization is given in [8]. In [11], the concept of a non-blocking data structure is introduced. The notion of obstruction-freedom is introduced in [9]. Contention management is discussed in [6,10,13].

The importance of the unknown-bound semi-synchronous model in the context of shared memory systems was first investigated in [1]. In [12,14], indulgent algorithm are investigated in semi-synchronous shared memory systems. The interested reader will find in [15] a pedagogical description of several families of semi-synchronous and timing-based algorithms. Message-passing algorithms for partially synchronous systems were presented in various papers [3,4].

In [7], the weakest failure detectors that allow boosting an obstruction-free implementation into a wait-free or a non-blocking implementation have recently been identified (eventual prefect failures detector [2] is the weakest to implement a wait-free contention manager, and Ω^* is the weakest to implement a non-blocking contention manager).

2 The Computational Model

The system is made up of n processes, denoted $p_1, \ldots, p_n$, which communicate via shared objects. It is assumed that any number of processes may crash. A process that crashes stops its execution in a definitive manner.

The possibility and complexity of synchronization in a distributed environment depends heavily on timing assumptions. We focus on a semi-synchronous shared-memory model of computation which provides a practical abstraction of the timing details of concurrent systems. In this model, it is assumed that there is an *unknown* upper bound on the time it takes a process to execute one step and, in particular, on the time it takes to execute a step which involves access the shared memory. This assumption is inherently different from the asynchronous model where no such bound exists.

In the semi-synchronous model a process can delay itself explicitly by executing a statement *delay*(d), for some constant d. Executing the statement *delay*(d) by a process p delays p for at least d time units before it can continue, and

there is some (unknown) upper bound (as a function of d) on the time a correct process can be delayed (for example, this bound can be $2d$).

A key idea in designing algorithms for the semi-synchronous model is that a process can delay itself for increasingly longer periods, and by doing so it can ensure that eventually other processes will take "enough" steps during one of these waiting periods. The appeal of the semi-synchronous model lies in the fact that while it abstracts from implementation details, it is a better approximation of real concurrent systems compared to the asynchronous model, as all practical (shared memory) systems satisfy the unknown-bound assumption. Furthermore, it enables to obtain more efficient solutions.

We point out that the semi-synchronous model, as defined here, is similar (but not identical) to a model where it is assumed that there is an unknown bound on the ratio of the maximum time and minimum time between the steps of the processes. That is, some unknown bound exists on the relative execution rates of any two processes in the system. In such a model, the delay statement is simply implemented by counting steps. All our results and algorithms apply also to this variant of the unknown bound semi-synchronous model.

Lock-free algorithms usually require the use of powerful atomic operations such as compare-and-swap (CAS). A CAS operation takes three parameters: a shared register r, and two values: *old* and *new*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise r is left unchanged and the value *false* is returned.

We consider three types of shared objects: (1) Atomic register – a shared register that supports atomic read and write operations; (2) Compare-and-swap object – a shared object that supports an atomic CAS operation; (3) Compare-and-swap/read object – a shared object that supports both atomic CAS and atomic read operations.

The FLMS transformation, and our transformations are all *black box* transformations: A transformation does not change anything in the original obstruction-free algorithm, it only adds code to ensure that a stronger progress condition is satisfied. Thus, a transformed algorithm performs the original algorithm on the original shared objects and does not apply any other steps to these objects. Below we define a time complexity measure for analyzing such transformations. Let T be a transformation that converts an arbitrary obstruction-free algorithm, denoted ALG, into a non-blocking or a wait-free algorithm.

Enabled Process: *A process is enabled in a given finite run of transformation T if its next step is a step of the original obstruction-free algorithm ALG, or if its last step was a step of ALG.*

Being enabled corresponds to holding a lock (i.e., being in the critical section) in lock-based algorithms. We notice that being enabled is not exactly like being in a critical section since exclusiveness is not guaranteed.

Time Complexity: *The time complexity of transformation T, is the maximum number of steps which involve access to the shared memory that a process may*

need to take until it becomes enabled since the last time some process has been enabled. Or, if no process has been enabled yet, since the beginning of the execution.

This definition corresponds to the way time complexity is usually defined in lock-based (mutual exclusion) algorithms. In lock-based algorithms, time complexity is usually measured by counting the amount of "work" (time units) a winning process, a process that gets to enter its critical section, may need to do (wait) since the last time some process has released its critical section.

Since all our algorithms require very few accesses to shared memory locations, the definition does not distinguish between different types of shared memory accesses. In a different context, it would make sense to distinguish between relatively cheap operations like reads and writes to more expensive operations like compare-and-swap.

Remark: The new complexity measure is a special case of the following more general new complexity measure for synchronization algorithms, which might be interesting in its own right. Given an algorithm, denoted SYNC, let us divide its steps into three disjoint groups,

1. group A – the group of *synchronization* steps;
2. group B – the group of *real work* steps;
3. group C – the group of *inexpensive* steps.

In mutual exclusion algorithms, A may include the steps in the entry section, B the steps in the critical section, and C all other steps. For a transformation that converts an arbitrary obstruction-free algorithm, ALG, into a non-blocking or a wait-free algorithm, A may include all the step which involve access to the shared memory, B all the step of ALG, and C all other steps. A process is *enabled* in a given finite run of SYNC if its next step is a step from B or if its last step was a step from B. Next we define two possible complexity measures,

1. The maximum number of steps from group A that a process may need to take until it becomes enabled since the last time some process has been enabled. Or, if no process has been enabled yet, since the beginning of the execution.
2. The longest time interval where no process is enabled, assuming there is an upper bound of one time unit for step time and no lower bound.

The first measure generalizes the one used in this paper, the second measure is called *system response time* in the context of mutual exclusion algorithms. Other variants of these measures can be obtained by generalizing corresponding measures for lock-based algorithms (see [15], Section 1.4).

3 The Time Complexity of the FLMS Transformation

We prove that the time complexity of the FLMS transformation is at least *exponential* in the number of processes n. This exponential bound holds even when

the transformation is executed in a fault-free environment. As already mentioned, the transformation converts any obstruction-free algorithm into a wait-free algorithm when analyzed in the unknown-bound semi-synchronous model, and uses n atomic single-writer registers, n atomic multi-writer registers and a single atomic fetch-and-increment object.

In the following, we describe only the part of the FLMS transformation that is needed for proving the time complexity bound. When a process p_i notices that there is contention, it begins to participate in a strategy to ensure progress, called the *panic* mode.

> This strategy is as follows: using an atomic fetch-and-increment object, process p_i first acquires a timestamp, and initializes an atomic multi-writer register, denoted $T[i]$, with the value of its timestamp. Then p_i searches for the minimum timestamp by scanning the array $T[1..n]$. (Initially all entries of the T array are set to ∞.) During the search, all timestamps that are not ∞, but are larger than the minimum timestamp p_i has observed so far, are replaced by ∞. If process p_i determines that it has the minimum timestamp then p_i becomes enabled.
> If process p_i determines that some other process, say p_k, has the minimum timestamp, p_i waits for some time (the amount of time waited is not relevant here), and then checks the status of p_k. If p_i does not notice (after checking some shared register) that p_k has taken steps while p_i was waiting, p_i overwrites p_k's timestamp by setting $T[k]$ to ∞. Then p_i restarts executing the strategy to ensure progress (i.e., go to the beginning of the *panic* mode) using its original timestamp (i.e., p_i uses the same timestamp from the previous round). Similarly, if after it waits, p_i notices that $T[k] = \infty$, then p_i also restarts executing the strategy to ensure progress using its original timestamp.

The above partial description of the FLMS transformation is sufficient for proving its exponential time complexity.

Theorem 1. *The time complexity of the FLMS transformation is exponential, in the number of processes n.*

Proof. Consider a finite run σ where: (1) all the n processes have just started to execute the strategy to ensure progress (i.e, the panic mode); (2) each process has chosen a timestamp such that process p_i has chosen timestamp i, for all $i \in \{1, ..., n\}$; and (3) all the entries of the T array are still set to their initial value ∞.

For every $i \in \{1, ..., n-1\}$, let R_i denotes the maximum number of times, that process p_n has to scan the array $T[1..n]$ starting from (the end of) run σ before p_n becomes the first enabled process, assuming that only the $i + 1$ processes p_n,...,p_{n-i} may take steps in an extension of σ. We prove by induction that $R_i \geq 2^i$ for every $i \in \{1, ..., n-1\}$, which would imply that the time complexity is of order $2^{n-1} \times n$. Actually, we prove by induction the following (stronger) claim:

For every $i \in \{1, ..., n-1\}$, there is an extension σ_i of σ where:

1. p_n has performed 2^i scans of T in σ_i.
2. Only the $i+1$ processes p_n,...,p_{n-i} have taken steps in the extension σ_i of σ.
3. p_n is enabled in σ_i, and no process is enabled in any strict prefix of σ_i which extends σ.
4. For every $j \in \{1, ..., n-1\}$, process p_j is (again) at the beginning of the (code of the) panic mode in σ_i, with timestamp j and $T[j] = \infty$.
5. In the last steps in σ_i process p_n has scanned the array $T[1..n]$. We denote by σ_i^* the prefix of σ_i which result from omitting this last single scan of T by p_n.

We notice that the existence of run σ_i implies that $R_i \geq 2^i$.

When $i = 1$, σ_1 is constructed as follows: we first let process p_{n-1} set $T[n-1]$ to $n-1$. Then, we run p_n *alone*. Process p_n searches for the minimum timestamp by scanning the array $T[1..n]$ once, and determines that process p_{n-1} has the minimum timestamp. Then, process p_n delays itself for some time and then checks the status of p_{n-1}. Since p_{n-1} has taken no steps while p_n was waiting, p_n overwrites p_{n-1}'s timestamp by setting $T[n-1]$ to ∞. Then p_n restarts executing the strategy to ensure progress (i.e., go to the beginning of the panic mode) using its original timestamp. Next, p_n sets $T[n]$ to n, searches again for the minimum timestamp by scanning T, determines that it has the minimum timestamp and becomes enabled. Since p_n has scanned T twice in σ_1, we get,

$$R_1 \geq 2^1. \tag{1}$$

When $i = 2$, σ_2 is constructed as follows: we first repeat the extension from the previous case and stop just *before* the last scan on T by p_n (i.e., the extension σ_1^*). Then we let process p_{n-2} set $T[n-2]$ to $n-2$, and let both p_{n-1} and p_n scan T (notice that so far the number of scans of p_n equals 2^1 as in σ_1). Both determine that process p_{n-2} has the minimum timestamp, each one delays itself for some time and then checks the status of p_{n-2}. Since p_{n-2} has taken no steps while p_{n-1} and p_n were waiting, they overwrite p_{n-2}'s timestamp by setting $T[n-2]$ to ∞. Then they restart executing the strategy to ensure progress. At this point, from process p_{n-1} and process p_n point of view, they are back at a situation similar to the one at run σ. So, we repeat the construction from the previous case of $i = 1$ (in which the number of scans of p_n equals 2^1). Since p_n has scanned T four times in σ_2, we get,

$$R_2 \geq 2^2. \tag{2}$$

Induction hypothesis: we assume that a run σ_{i-1} exists and prove that run σ_i exists.

We consider now the general case where $i+1$ processes participate. We first repeat the extension from the case when only i processes participate and stop just *before* the last scan on T by p_n (i.e., the extension σ_{i-1}^*). Then we let process p_{n-i} set $T[n-i]$ to $n-i$, and let the i processes p_{n-i+1} through p_n

scan T (notice that so far the number of scans by p_n equals 2^{i-1} as in σ_{i-1}). All the processes determine that process p_{n-2} has the minimum timestamp, they delay themselves for some time and then check the status of p_{n-i}. Since p_{n-i} has taken no more steps, they overwrite p_{n-i}'s timestamp by setting $T[n-i]$ to ∞. Then they restart executing the strategy to ensure progress. At this point, from processes p_{n-i+1} through p_n point of views, they are back at a situation similar to run σ. So, we repeat the construction from the $i-1$ case in which only i processes participate (in which the number of scans of p_n equals 2^{i-1}). Since p_n has scanned T $2 \times 2^{i-1}$ times in σ_i, we get,

$$R_i \geq 2^i. \tag{3}$$

Thus, from the construction of run σ_{n-1} where all the n processes participate, we get,

$$R_{n-1} \geq 2^{n-1}. \tag{4}$$

We have proved that there is an extension of σ where the number times, that process p_n has to scan the array $T[1..n]$ before it becomes the first enabled process is at least 2^{n-1}. Each such scan involves n accesses to shared memory location. Thus, we conclude that the time complexity the FLMS transformation is at least of order $2^{n-1} \times n$. □

4 The Main Transformation

We now present our main transformation. It has $O(1)$ time complexity, and uses n atomic single-writer registers and a single compare-and-swap object which supports also a read operation. The other transformations, presented later, are variants of this transformation. One important strength of all the transformations is their simplicity.

To understand how the transformation works, let us start by assuming a fault-free model in which no process ever crashes. In such a model, we can design a simple transformation by using a single (mutual exclusion) lock. To avoid interference between different operations, a process performs steps of the original obstruction-free algorithm, denoted ALG, only inside its critical section (after it has acquired the lock), within which the process is guaranteed exclusive access with no interference to the original algorithm shared objects.

Using a single lock to prevent interference between different operations of ALG may degrade the performance, as it enforces processes to wait for a lock to be released, and thus, does not allow several processes with non-interfering operations to proceed concurrently. Furthermore, when there is no contention, acquiring the lock introduces additional overhead.

To overcome these limitations, before a process tries to acquire the lock, it first tries to complete its operation of ALG without holding the lock. If there is no contention or if the contention manager is effective the process will complete its operation without any overhead. Otherwise if the process, after taking many steps, does not succeed in completing its operation, it tries to acquire the lock.

Of course, 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 this interference will vanish after some finite time.

Going back to our original model where processes may crash, using locks is problematic as a process may crash while holding the lock, preventing *all* other processes from ever completing their operations. Resolving this problem, is the main difficulty in designing efficient transformations, and is done as follows: The *winner* – the process that is currently holding the lock – is required to increment a (single-writer) counter, denoted $W[winner]$ every few steps, of ALG. A process p that fails to acquire the lock, reads the value of the winner's counter and delays itself for $W[winner]$ time units. Then, p checks $W[winner]$ again, and if the value was updated p delays itself again, and so on. Otherwise, if $W[winner]$ has not been changed, p assumes that the *winner* has crashed and releases the lock.

Releasing the lock by a process p, which is not the *winner*, is a very delicate issue, since the *winner* might be alive but very slow, and as a result: (1) the winner will notice that the lock has been released although it is interested in holding it further; (2) we might end up with two or more processes holding the lock at the same time; and (3) process p might be suspended just before releasing the lock, and may release the lock at some unexpected time later on.

We address these problems as follows: when a winner process, say p_i, notices that the lock has been released p_i tries to acquire the lock again. However, before doing so, p_i waits long enough so that other processes that have mistakenly concluded that p_i has crashed, will have enough time to release the lock (again) before p_i tries to acquire it again. Ensuring that eventually at most one correct process will hold the lock, has to do with the fact that the value of the counter of a winning process $W[winner]$ keeps on increasing over time. Thus, forcing processes that fail to acquires the lock to delay themselves for increasingly longer periods, and eventually – by the unknown-bound assumption, the waiting time is long enough to guarantee that only one process will hold the lock and that some process will complete its operation of ALG.

The code of our main transformation, Transformation 1, is given in Figure 1. Transformation 1 converts an arbitrary obstruction-free algorithm, denoted ALG, which may include a contention manager, into a non-blocking algorithm.

Process p_i first tries to execute X steps (for some predetermined constant X) of the original obstruction-free algorithm ALG (line 1). If p_i succeeds to complete its operation, it returns (line 2), otherwise p_i tries to acquire the lock. The lock is implemented by a compare-and-swap object, named T. $T = 0$ means that the lock is free, $T = i$ means that process p_i has acquired the lock. So process p_i tries to acquire the lock by setting the value of T to i (line 5). If p_i succeeds it tries to complete its operation by taking steps of the original algorithm ALG (lines 6 – 12). Every X such steps p_i increments its counter $W[i]$ by 1. It continues doing so until it either completes its operation and releases the lock (line 9) or finds that it is no longer holding the lock (line 12).

```
shared
    T: CAS/read object, initially 0                                  /* "the lock" */
    W[1..n]: array of atomic single-writer registers        /* initial values immaterial */
local                                                       /* initial values immaterial */
    winner: ranges over {0, ..., n}; wait: integer; b: boolean

invoke(op)
1  execute up to X steps of ALG                          /* ALG is the original algorithm */
2  if op is completed then return response fi
3  W[i] := 1                                  /* contention possible – set initial delay */
4  repeat                                   /* tries to execute op without interference */
5      if CAS(T, 0, i) then                          /* tries to acquires the "lock" */
6          repeat                                                    /* p_i is enabled */
7              execute up to X steps of ALG                 /* original algorithm */
8              if op is completed then
9                  CAS(T, i, 0)                                   /* release "lock" */
10                 return response
11             else W[i] := W[i] + 1 fi                          /* increase delay */
12         until read(T) ≠ i                        /* equivalent to ¬CAS(T, i, i) */
13         delay(2 × W[i])          /* flash out processes waiting in lines 16–22 */
14     else                                                              /* loser */
15         winner := read(T)        /* tricky to imp. efficiently using CAS only */
16         if winner ≠ 0 then                    /* "lock" is captured by winner */
17             repeat                          /* wait for the winner to proceed */
18                 wait := W[winner]                                  /* delay time */
19                 delay(wait)                                  /* wait as requested */
20                 b := read(T) = winner       /* b := CAS(T, winner, winner) */
21             until wait = W[winner] ∨ ¬b                  /* winner crashed? */
22             if wait = W[winner] ∧ b then CAS(T, winner, 0) fi fi fi  /*release */
23 until op is completed
```

Fig. 1. Transformation 1. Program for process p_i which invokes operation *op*.

In line 13, process p_i delays itself, so that other processes that may have concluded that p_i has crashed (lines 16–21), will have enough time to release the lock (line 22) before p_i tries to acquire the lock again. After executing the delay (line 13), process p_i tries to acquire the lock again. We notice that p_i may decrease the value of $W[i]$ only after it completes its operation of ALG.

If p_i fails to acquire the lock (line 5), it executes the code at lines 15 – 22. It finds out the identity of the winner (line 15), and waits for $W[winner]$ time units. Then, p_i checks $W[winner]$ again, and if the value has been updated (meaning the winner is alive) p_i delays itself again, and so on. p_i does so until it notices that either $W[winner]$ has not been changed or that the lock has been released (line 21). If $W[winner]$ has not been changed, p_i assumes that the *winner* has crashed, releases the lock (line 22), and tries to acquire the lock (line 5).

Theorem 2. *Transformation 1 converts any obstruction-free algorithm into a non-blocking algorithm when analyzed in the unknown-bound semi-synchronous model.*

Proof. Assume to the contrary that there exists an obstruction-free algorithm, ALG, such that the transformation does not convert ALG into a non-blocking algorithm when analyzed in the unknown-bound semi-synchronous model. Thus, there exists a suffix, σ_0, of an infinite run σ in which (1) no process succeeds to complete an operation of ALG; and (2) no process executes line 1 or line 2. Let P denotes the set of all correct processes that do no succeed to complete their operations in σ_0.

Clearly, there must be at least one process $p_i \in P$, which succeeds to capture the "lock" in line 5 infinitely often (that is, its compare-and-swap operation in line 5 is successful infinitely often) and hence it executes the repeat loop in lines 6–12 infinitely often. This implies that the value of $W[i]$ grows without bound in σ_0. Thus, there exists a suffix σ_1 of σ_0, in which the value of $W[i]$ is big enough such that immediately after p_i executes the delay statement $delay(2 \times W[i])$ in line 13, no correct process is in the middle of executing any of the lines 16 – 22 while having its local variable *winner* set to i. In particular, no process can successfully execute the statement $CAS(T, i, 0)$ in line 22 (without p_i taking further steps).

Let us denote by r an upper bound on the number of time units required for p_i to go through the repeat loop at lines 6 – 12 *once* regardless of the activity of the other processes (such a bound exists by the properties of the unknown-bound semi-synchronous model). Let σ_2 be a suffix of σ_1 where (1) p_i succeeds in capturing the "lock" in line 5, and starts executing the repeat loop at lines 6 – 12, (2) $W[i] \geq r$, and (3) no correct process is in the middle of executing any of the lines 16 – 22 while having its local variable *winner* set to i.

Clearly, in σ_2, no process p_j will ever be able to successfully executes the statement $CAS(T, i, 0)$ in line 22, because each time p_j will execute the delay statement in line 19, p_i will go through the loop at least once and increment $W[i]$. Thus, (1) from that point on the value of T forever equals i, and (2) process p_i will never leave the repeat loop at lines 6 – 12. Thus, in σ_2, every other process that is in the middle of executing the repeat loop at lines 6 – 12 will eventually execute line 12 and exits the repeat loop. Thus, there exists a suffix σ_3 of σ_2 where p_i forever executes the loop at lines 6 – 12 alone. This implies that in σ_3 processes p_i will execute its operation on ALG continuously without interference and hence this operation must eventually be completed. A contradiction □

Theorem 3. *Transformation 1 has $O(1)$ time complexity, and uses n atomic single-writer registers and one CAS/read object.*

Proof. Assume that process p_i becomes enabled. Lets examine what is the maximum number of steps which involve access to the shared memory that p_i may need to take until it becomes enabled since the last time some process has been enabled. Process p_i can becomes *enabled* in one of three ways: (1) when it starts it execution (line 1); (2) immediately after it succeeds to set the value of the CAS object T to its id (line 5), and (3) starting to execute another round of the repeat loop after executing lines 11 and 12. Option 1 requires 0 steps by p_i. Option 3 requires 2 steps by p_i since p_i was last enabled. So, lets assume that p becomes enabled as a result of option 2. Process p_i succeeds in setting the value of T to its id (line 5) only when $T = 0$. As soon as the value of T is 0 it will

take p_i at most 5 steps (which involve access to the shared memory) to reach line 5. Finally, if $T = j$ and process p_j crashes or is slow, it will take p_i or some other process at most 8 steps until they set T to 0 in line 22. The result about the space complexity is obvious. □

5 Transformation 2: Using a CAS Object with n Atomic Registers

Our second transformation is a modified version of Transformation 1, in which the three *read*(T) operations from Transformation 1 (in lines 12, 15, and 20), are

```
shared
    T: CAS object, initially 0                                   /* "the lock" */
    W[1..n]: array of atomic single-writer registers   /* initial values immaterial */
local                                                  /* initial values immaterial */
    winner: ranges over {−1, 0, ..., n}; wait, t: integer; b: boolean

invoke(op)
1   execute up to X steps of ALG                    /* ALG is the original algorithm */
2   if op is completed then return response fi
3   W[i] := 1                                  /* contention possible – set initial delay */
4   repeat                                  /* tries to execute op without interference */
5       if CAS(T, 0, i) then                          /* tries to acquires the "lock" */
6           repeat                                                   /* p_i is enabled */
7               execute up to X steps of ALG                 /* original algorithm */
8               if op is completed then
9                   CAS(T, i, 0)                                   /* release "lock" */
10                  return response
11              else W[i] := W[i] + 1 fi                          /* increase delay */
12          until ¬CAS(T, i, i)
13          delay(2 × W[i])                 /* flash out processes waiting in line 22 */
14      else                                                              /* loser */
15.1        j := 0; winner := −1
15.2        repeat                                          /* find the winner's id*/
15.3            if j (mod n) + 1 ≠ i then
                    j := j (mod n) + 1 else j := j + 1 (mod n) + 1 fi
15.4            if CAS(T, j, j) then winner := j fi          /* is j the winner? */
15.5            if CAS(T, 0, 0) then winner := 0 fi      /* "lock" is released ? */
15.6        until winner ≠ −1                                  /* winner found */
16          if winner ≠ 0 then                  /* "lock" is captured by winner */
17              repeat                          /* wait for the winner to proceed */
18                  wait := W[winner]                                /* delay time */
19                  delay(wait)                          /* wait as requested */
20                  b := CAS(T, winner, winner)
21              until wait = W[winner] ∨ ¬b                  /* winner crashed? */
22              if wait=W[winner] ∧ b then CAS(T, winner, 0) fi fi fi  /*release */
23  until op is completed
```

Fig. 2. Transformation 2. Program for process p_i which invokes operation *op*.

```
type
   lock: record {id: integer ; W[1..n]: array of integers}
shared
   T: CAS/read object of type lock , initially T.id = 0                /* "the lock" */
local                                                  /* initial values immaterial */
   temp, temp_1, temp_2: of type lock; winner, wait: integer; b: boolean

invoke(op)
1  execute up to X steps of ALG                        /* ALG is the original algorithm */
2  if op is completed then return response fi
3  setW(1)                                                          /* set W[i] to 1 */
4  repeat                                  /* tries to execute op without interference */
5       if setTid(0, i) then                          /* tries to acquires the "lock" */
6            repeat                                                 /* p_i is enabled */
7                 execute up to X steps of ALG                  /* original algorithm */
8                 if op is completed then
9                      setTid(i, 0)                                  /* release "lock" */
10                     return response
11                else temp := read(T); setW(temp.W[i] + 1) fi  /* increment W[i] */
12           until temp.id ≠ i                    /* until p_i does not hold the lock */
13           delay(2 × temp.W[i])       /* flash out processes waiting in lines 16–22 */
14      else                                                          /* loser */
15           temp := read(T); winner := temp.id
16           if winner ≠ 0 then                    /* "lock" is captured by winner */
17                repeat                          /* wait for the winner to proceed */
18                     wait := temp.W[winner]                        /* delay time */
19                     delay(wait)                    /* wait as requested */
20                     temp := read(T); b := temp.id = winner
21                until wait = temp.W[winner] ∨ ¬b          /* winner crashed? */
22                if wait = temp.W[winner] ∧ b then setTid(winner, 0) fi fi fi/*rel.*/
23 until op is completed

function setW (val: integer)                                        /* W[i] := val */
1  repeat
2       temp_1 := read(T); temp_2 := temp_1; temp_2.W[i] = val
3  until CAS(T, temp_1, temp_2)
end

function setTid (old:integer, new:integer) return: boolean   /* CAS(T, old, new) */
1  temp_1 := read(T); b := false
2  while temp_1.id = old do
3       temp_2 := temp_1; temp_2.id = new
4       b := CAS(T, temp_1, temp_2)
5       temp_1 := read(T) od
6  return(b)
end
```

Fig. 3. Transformation 3. Program for process p_i which invokes operation *op*.

implemented without using an implicit read operations of T. Transformation 2 has $O(1)$ time complexity, and uses n atomic single-writer registers and a single compare-and-swap object (which *does not* support a read operation). The code of Transformation 2, is given in Figure 2.

The *read*(T) operations in lines 12 and 20 are easy to implement, as we are only interested in knowing whether the value of T equals some specific value. Implementing the *read*(T) operation in line 15, while preserving the $O(1)$ time complexity of the transformation is slightly more complicated. The easiest way to implement *read*(T) is to check, for each value $i \in \{0, ..., n\}$, whether the operation $CAS(T, i, i)$ returns true. However, such an implementation would increase time complexity of the transformation to $O(n)$. This can be easily fixed. First, we observe that as long as the value of T (in Transformation 1) is different from 0, some process is enabled; and that only steps that are taken while no process is enabled are counted. Thus, after each time we check whether the value of T equals i for $i \neq 0$ (by executing $CAS(T, i, i)$) we check whether the value of T equals 0 (by executing $CAS(T, 0, 0)$). The final implementation can be seen in lines 15.1 to 15.6. Correctness follows from that of Transformation 1.

6 Transformation 3: Using a Single CAS/Read Object

Our third transformation is also a modified version of Transformation 1, in which the values of the atomic registers $W[1..n]$ are encoded as part of the state of the CAS/read object T. Transformation 3 has $O(1)$ time complexity, and uses a single compare-and-swap/read object (with no atomic registers).

Using only a single shared object may degrade the performance, as it forces all processes to reference the same shared memory location. Thus, under contention, the average waiting time to access the shared object would be high. The code of Transformation 3, is given in Figure 3. The correctness of Transformation 3 follows from that of Transformation 1.

7 Discussion

We have introduced a new complexity measure and presented three transformations which are shown very efficient according to this measure. The transformations convert any obstruction-free algorithm into a non-blocking algorithm when analyzed in the unknown-bound semi-synchronous model.

As we have shown, the FLMS transformation has exponential time complexity. It is an open question whether achieving wait-freedom must require exponential time complexity when using only atomic registers and fetch-and-increment objects. It would be interesting to find tight bounds also when using other base objects. In particular, in what cases obstruction-free to non-blocking transformations have better time complexity than obstruction-free to wait-free transformations? Are transformations to wait-free implementations are inherently expensive?

References

1. Alur, R., Attiya, H., Taubenfeld, G.: Time-adaptive algorithms for synchronization. SIAM Journal on Computing 26(2), 539–556 (1997)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
3. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987)
4. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
5. Fich, E.F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 78–92. Springer, Heidelberg (2005)
6. Guerraoui, R., Herlihy, M., Pochon, B.: Towards a theory of transactional contention managers. In: Proc. 24th Symposium on Principles of Distributed Computing, pp. 258–264 (2005)
7. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 376–390. Springer, Heidelberg (2006)
8. Herlihy, M.P.: Wait-free synchronization. ACM Trans. on Programming Languages and Systems 13(1), 124–149 (1991)
9. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proc. of the 23rd International Conf. on Dist. Computing Systems, p. 522 (2003)
10. Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proc. 22nd ACM Symp. on Principles of Distributed Computing, pp. 92–101. ACM Press, New York (2003)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems 12(3), 463–492 (1990)
12. Raynal, M., Taubenfeld, G.: The notion of a timed register and its application to indulgent synchronization. In: Proc. 19th ACM Symp. on Parallelism in Algorithms and Architectures. ACM Press, New York (2007)
13. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proc. 24th Symposium on Principles of Distributed Computing, pp. 240–248 (2005)
14. Taubenfeld, G.: Computing in the presence of timing failures. In: Proc. 26th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'06) (2006)
15. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Pearson / Prentice-Hall, p. 423 (2006) ISBN 0-131-97259-6
16. Valois, J.D.: Implementing lock-free queues. In: Proc. of the 7th International Conference on Parallel and Distributed Computing Systems, pp. 212–222 (1994)