Obstruction-Free Algorithms can be Practically Wait-Free

Faith Ellen Fich[2], Victor Luchangco[1], Mark Moir[1], and Nir Shavit[1]

[1] Sun Microsystems Laboratories
[2] University of Toronto

Abstract. The obstruction-free progress condition is weaker than previous nonblocking progress conditions such as lock-freedom and wait-freedom, and admits simpler implementations that are faster in the uncontended case. Pragmatic contention management techniques appear to be effective at facilitating progress in practice, but, as far as we know, none *guarantees* progress.
We present a transformation that converts any obstruction-free algorithm into one that is wait-free when analyzed in the *unknown-bound* semisynchronous model. Because all practical systems satisfy the assumptions of the unknown-bound model, our result implies that, for all practical purposes, obstruction-free implementations can provide progress guarantees equivalent to wait-freedom. Our transformation preserves the advantages of any pragmatic contention manager, while guaranteeing progress.

1 Introduction

Substantial effort has been made over the last decade in designing nonblocking shared data structure implementations, which aim to overcome the numerous problems associated with lock-based implementations. Despite this effort, designs satisfying traditional nonblocking progress conditions, such as wait-freedom and—to a lesser extent—lock-freedom, are usually complicated and expensive.

Significant progress in overcoming these problems has been achieved recently by designing implementations that satisfy the weaker *obstruction-free* nonblocking progress condition, which requires progress guarantees only in the (eventual) absence of interference from other operations [16]. This weaker requirement allows simpler implementations that perform better in the common uncontended case. Recently Herlihy, Luchangco, Moir and Scherer [18] introduced a dynamic software transactional memory (DSTM) package, which allows programmers to develop obstruction-free data structures without reasoning about concurrency.

That obstruction-free data structures do not guarantee progress under contention is not just a theoretical concern: they are observed to suffer from livelock in practice. To combat this problem, obstruction-free implementations are combined with *contention managers* [18], whose role is to facilitate progress when necessary by allowing operations to run without interference long enough to complete. While a number of contention managers have proved effective in practice [18, 26, 27], as far as we know, none *guarantees* progress.

In this paper we show that the advantages of this pragmatic approach can be exploited *without* giving up strong progress guarantees. We do so by showing how to transform any obstruction-free algorithm so that it guarantees that every operation eventually completes, given some very weak timing assumptions about the target system. These assumptions are embodied by the *unknown-bound* semisynchronous model of computation [9, 2]. Roughly speaking, this model assumes that some bound exists on the relative execution rates of any two processes in the system, but does not assume that the bound is known. All practical systems satisfy this assumption.

Our transformation does not affect the behavior of the original algorithm (except for a very small overhead) until some operation decides that it has run for too long without completing. Furthermore, our transformation can be applied to an obstruction-free algorithm combined with any valid contention manager (i.e., one that preserves the obstruction-freedom of the algorithm; see [18] for restrictions on contention managers), allowing us to take advantage of the practical benefits of a heuristic contention manager that does not guarantee progress, without sacrificing progress guarantees.

Considering obstruction-free implementations significantly reduces the burden on designers of data structures (and software transactional memory implementations) by eliminating the need to ensure progress under contention. Furthermore, designers of contention managers for obstruction-free implementations have a great deal of latitude because contention managers can change the timing behavior of an execution arbitrarily without causing safety requirements to be violated. This is because the obstruction-free implementations are proved safe in an *asynchronous* model of computation, in which processes may execute arbitarily fast or slow or even stop executing all together. Therefore, contention manager designers are free to experiment with a wide range of heuristics for controlling contention, and in particular can exploit timing information available in the target system, for example to delay an operation to prevent it from interfering with another.

The idea of combining an algorithm that ensures the required safety properties in an asynchronous model, but does not guarantee progress, with a mechanism that exploits timing information about the execution environment to ensure progress is not new. For example, failure detectors [7, 19] can be used with asynchronous consensus algorithms to guarantee progress in the face of failures. Similarly, the Disk Paxos algorithm [12] employs a leader election algorithm to ensure progress.

Although these approaches are similar in spirit to the obstruction-free approach, there are differences both in motivation and in acceptable solution approaches. First, research on failure detectors focuses on fault tolerance. It is known to be impossible to tolerate the failure of even a single process in some asynchronous environments, including message passing environments and shared memory systems in which memory can be accessed only using read and write operations [11, 20]. In such environments, it is *necessary* to exploit synchrony in

the system to solve fundamental problems such as consensus. Failure detector research aims at characterizing and separating out this synchrony.

In contrast, research on obstruction-free algorithms has focused on modern shared-memory multiprocessors, which support strong synchronization primitives, such as compare-and-swap (CAS). It has long been understood that we can implement any shared data structure so that it can tolerate process crashes in such environments, even in an asynchronous model [15]. Thus, recent work on the obstruction-free approach to implementing nonblocking data structures is not motivated by fault tolerance, but by performance, simplicity of design, and separation of concerns: we achieve simpler implementations that perform better in the common uncontended case, while separating the design of mechanisms for achieving progress (e.g., contention managers) from the design of the underlying obstruction-free algorithm that guarantees the required safety properties.

The Disk Paxos algorithm [12] uses a consensus algorithm to agree on transitions of a replicated state machine, but the consensus algorithm does not guarantee progress under contention. Therefore, the algorithm uses a leader election algorithm, and processes take steps of the consensus algorithm only when they believe themselves to be the leader. The leader election algorithm eventually ensures that exactly one process is the leader (provided some reasonable assumptions about system "stability" are eventually satisfied), and thereby ensures progress. This can be viewed as a form of contention management. However, such use of a leader election algorithm as a contention manager is not acceptable in the design of shared data structures, because it eliminates concurrency in the common case. This is natural in the case of Disk Paxos, because concurrent operations trying to reach consensus necessarily synchronize with each other, but operations on shared data structures should be able to proceed in parallel when they do not conflict. Therefore, we have taken care to design our transformation so that it uses its original contention manager as long as it is effective, and only attempts to serialize operations if this contention manager proves ineffective. This way, we guarantee that every operation eventually completes, while continuing to exploit the natural concurrency between nonconflicting operations.

To our knowledge, the only other work aimed at providing strong progress guarantees for obstruction-free algorithms is due to Guerraoui, Herlihy, and Pochon [14]. They present a simple contention manager and prove that it ensures that every transaction completes after a bounded delay. However, their contention manager is blocking, which means a single thread failure can prevent further progress by any other transaction, and common events, such as thread preemptions, can prevent progress for long periods of time.

Scherer and Scott [26] developed the *timestamp* contention manager based on the ideas used by our transformation, but have not made any claims about whether or under what circumstances it ensures progress. Furthermore, their experiments show that this contention manager does not perform as well as others they have invented. Our transformation shows that we can choose a contention manager based on its performance in common cases, rather than in the worst case, without giving up guaranteed progress.

2 Background

Before presenting our transformation, we introduce background on nonblocking shared data structures, nonblocking progress conditions, and asynchronous and semisynchronous models of computation, and briefly describe some previous results that use semisynchronous models to analyze implementations.

2.1 Nonblocking shared data structures

Today, almost all concurrent programs rely on blocking constructs such as mutual exclusion locks for synchronizing access to shared data structures. The use of locks introduces numerous problems, including deadlock, performance bottlenecks, and priority inversion [15]. Researchers have investigated nonblocking implementations in the hope of eliminating these problems.

An implementation of a shared data structure in a shared memory system provides a representation of the data structure using base objects in the system and provides algorithms for the processes of the system to perform *operations* on the data structure.

Most nonblocking algorithms are based on an *optimistic* approach to synchronization, in which an operation is attempted but may fail to take effect if another concurrent operation interferes. In this case, the operation is retried. A significant source of difficulty is guaranteeing that an operation is not retried repeatedly without ever completing. Generally, stronger nonblocking progress guarantees are more difficult to achieve, and require algorithms that are more complicated and more expensive.

2.2 Nonblocking progress conditions

A *wait-free* implementation [15] guarantees that when a process performs an operation, it completes the operation in a finite number of its own steps, regardless of how fast or slowly other processes execute, and even if they stop executing permanently. Such strong progress guarantees are attractive, but often very difficult to achieve. Most wait-free algorithms in the literature are too complicated and too expensive to be useful in practice.

A *lock-free* implementation guarantees that, starting from any state in which one or more processes are executing operations, *some* process will complete its operation within a finite number of steps. This weaker progress condition usually makes lock-free implementations easier to design than wait-free ones. Simple and practical lock-free implementations have been achieved for a small number of important data structures, such as stacks [28], queues [25], and workstealing deques [3, 8]. Lock-freedom has generally been regarded as acceptable because well known contention management techniques such as *backoff* [1] are effective at reducing contention when it arises, thereby achieving progress in practice, despite the lack of the strong theoretical guarantee of wait-freedom.

Herlihy, Luchangco, and Moir [16] recently proposed the obstruction-free approach to implementing nonblocking operations for shared data structures. An

obstruction-free implementation simply guarantees that a process will complete its operation if it eventually executes enough steps without interference from other processes. Thus, if two or more processes repeatedly interfere with each other, it is possible that *none* of them completes its operation. The view is that, because contention management techniques are required to achieve acceptable performance when contention arises anyway, it is unnecessary to make *any* progress guarantees in the case of contention between concurrent operations.

Several examples in the literature suggest that by providing only obstruction-free progress guarantees, significantly simpler implementations can be achieved that are faster in the uncontended case [16, 18, 21]. Furthermore, although an implementation that is obstruction-free but not lock-free will exhibit livelock if contention is ignored, experience shows that livelock can be effectively avoided by using simple contention management strategies [18, 26, 27].

2.3 Asynchronous and semisynchronous models of computation

Concurrent algorithms are usually required to ensure safety properties regardless of how the steps of concurrent processes are interleaved, and (therefore) regardless of how fast or slowly any process executes. In other words, these algorithms should be proved safe in an *asynchronous* model of computation, in which the steps of processes are scheduled by an adversarial scheduler that can perform many steps of a process consecutively or perform them arbitrarily far apart. In such a model, it is impossible for a process to determine whether another process has crashed (i.e., stopped executing) or is just running very slowly.

Of course, in reality, there are limits to how fast or slowly processes can run. Some algorithms exploit assumptions about these limits to improve in various ways on algorithms designed for an asynchronous model. Such algorithms are analyzed in synchronous or semisynchronous models of computation that embody timing assumptions made about the target execution environment.

In a *synchronous* model, all processes execute steps at the same rate (until they crash). This means that if a process does not perform a step when it should, other processes can detect that it has crashed. However, if the correctness of a particular algorithm depends on all (noncrashed) processes performing steps precisely at a given rate, then tiny variations in execution rate, for example due to one processor becoming warmer than another, can cause incorrect behavior. Consequently, such algorithms are not generally practical.

Semisynchronous models relax these timing requirements, allowing processes to execute steps at different rates, and even allowing the rate at which a particular process executes to vary over time. However, it is assumed that there is an upper bound on the relative execution rates of any pair of processes. To be more precise, let us define the *maximum step time* of an execution as the longest time between the completion times of consecutive steps of any process. We define *minimum step time* analogously. Semisynchronous models assume that there exists a finite R such that in all executions, the ratio of the maximum and minimum step times is at most R. The evaluation of algorithms in semisynchronous models has value for the practitioner because real-world systems satisfy the assumptions

of such models, and for the theoretician in understanding the limitations of assumptions on timing.

In the *known-bound* model [2, 23], R is known by all processes. This implies that a process can wait long enough to guarantee that every other process has taken another step, or has crashed. Some algorithms that depend on knowledge of R can violate safety requirements in systems that do not satisfy the assumed bound. Conservative estimates of the bound for a particular system generally translate into worse performance, so designers are faced with a dangerous tradeoff in using such algorithms. Thus, such algorithms are not easily portable and indeed may violate safety properties in a given system if the system stops satisfying the timing assumptions, for example due to increased temperature.

In the *unknown-bound* model [9, 2], R is *not* known to processes. Thus, in contrast to the synchronous and known-bound models, a process does not know how long to wait to ensure that every other process that has not crashed takes a step. Therefore, it is not possible for a process to detect that another process has crashed. Nonetheless, it is possible for algorithms to wait for increasingly longer periods, and to exploit the knowledge that *eventually* all noncrashed processes have taken a step during one of these periods. It has been shown that an algorithm that is correct in this model does not violate any of its safety properties even in an asynchronous model, although progress properties proved in the unknown-bound model may not hold in an asynchronous model [2].

Algorithms that are correct in an asynchronous model are nonetheless sometimes analyzed in a synchronous or semisynchronous model, thus allowing the analysis to depend on various timing assumptions. Because contention management techniques such as backoff fundamentally rely on operations waiting for some time before retrying, they cannot be meaningfully analyzed in an asynchronous model of computation, which has no notion of time whatsoever.

In this paper, we show how to transform any obstruction-free implementation into one that guarantees that every process performing an operation eventually completes the operation, when analyzed in the unknown-bound model. Thus, the resulting algorithm is safe to use in any non-real-time application, and guarantees that every operation eventually completes in any practical system.

2.4 Some previous work using semisynchronous models

The study of algorithms in semisynchronous models has a long tradition in the distributed-computing community [9, 10, 5]. Semisynchronous algorithms have received considerable attention in the context of shared-memory synchronization. For lack of space, we mention only a few of the results.

Fischer [10] was the first to propose a timing-based mutual exclusion algorithm. He showed that in a known-bound model, there is a simple and efficient algorithm that uses a single shared variable. This overcame the linear space lower bound of Burns and Lynch [6] for asynchronous systems. Unfortunately, that algorithm violates safety properties if the timing assumptions of the model are violated. Lynch and Shavit [23] improved on Fischer's algorithm: in the same model, they presented an algorithm that uses two variables and ensures safety

even if the timing assumptions are violated, although progress is not guaranteed in this case. Gafni and Mitzenmacher [13] analyzed this algorithm under various stochastic timing models. Alur, Attiya, and Taubenfeld [2] showed that timing-based mutual exclusion and consensus algorithms with constant time complexity in the uncontended case exist in the unknown-bound model.

Counting networks [4] have also been evaluated in semisynchronous models. Lynch, Shavit, Shvartsman, and Touitou [22] showed that some nonlinearizable uniform counting networks are linearizable when analyzed in the known-bound model. This result was generalized by Mavronicolas, Papatriantafilou, and Tsigas [24] to nonuniform networks under a variety of timing assumptions. In these results, the linearizability of the implementations depends on timing.

3 Our transformation

We begin by explaining some simple ways of ensuring progress for each operation under various different assumptions and models. These ideas motivate the techniques used in our algorithm, and explain why they are needed under the weak assumptions of the unknown-bound model.

First, if we assume that processes never crash, then it is easy to ensure progress, even in an asynchronous model. This is achieved by ordering operations using timestamps, and having each process wait until all earlier operations in this order have completed before performing the steps of its own operation. This ensures that operations do not encounter contention with concurrent operations while executing the original obstruction-free algorithm, so every operation eventually completes. However, if a process does crash while it has a pending operation, no operations with later timestamps can be executed.

In a synchronous model, if all processes know an upper bound B on the number of consecutive steps that must be taken by a process to ensure that its operation completes, then it is easy to guarantee that each operation completes, even if processes can crash. The idea is, again, to order operations using timestamps and to have processes refrain from executing their operations while operations with earlier timestamps are pending. However, unlike in the asynchronous model, a process can detect if another process crashed while executing its operation: If the operation is not completed within B steps, then the process executing it must have crashed. In this case, a process can execute its operation when every operation with an earlier timestamp has either completed, or will not interfere further because the process executing it has crashed.

A similar approach works in the known-bound model. In this case, a process that is waiting for an earlier operation than its own to complete must conservatively assume that it is executing its steps at the maximum speed allowed by the model relative to the speed of the process executing the earlier operation. Thus, in this model, a process must wait for RB steps in order to be sure that another process has had time to execute B steps, where R is the ratio of the maximum and minimum step times.

However, this technique does *not* work in the unknown-bound model because the bound R is not known to processes. In fact, in this model, it is impossible for one process to determine that another process has crashed. Nonetheless, ideas similar to those described above can be used to guarantee that each operation excuted by a process that does not crash will complete even in the unknown-bound model. The key idea is that, rather than delaying for an amount of time that is *known* to be long enough to allow another process to take B steps, a process can delay for increasingly long periods of time while an earlier operation has not completed.

Each time a process performs b steps of its operation, for some constant b, it increments a counter. This serves the dual purposes of demonstrating that it has not crashed, and therefore must be deferred to by later operations, as well as increasing the number of steps for which later operations must defer. After a process has waited the required number of steps for an earlier operation that has not been completed and whose counter has not been incremented, it *assmues* that the process performing the earlier operation has crashed. Consequently, it removes the timestamp of that operation from the order of operations under consideration and proceeds.

In case the process executing the earlier operation has, in fact, not crashed, it reinstates its operation into the order (using its original timestamp). With this arrangement, if a process does crash while executing an operation, then it is removed from consideration and does not prevent progress by other operations. On the other hand, if an operation fails to complete because others did not wait long enough, then they will wait longer next time, so the bound provided by the model ensures that eventually they will wait long enough and the operation will complete.

It is important to note that the worst-case bound R for a particular system might be very high, because a process might occasionally take a very long time between two steps. However, the algorithm has no knowledge of the bound, so the bound does not affect the performance of the algorithm; only the particular execution behaviour does. Furthermore, even if an unlikely sequence of events causes progress to take a long time, this has no bearing on how the algorithm behaves in the future. In practice, processes run at approximately the same speed most of the time. Therefore, the *effective* bound will generally be small, even if, in theory, the actual bound is very large.

This description captures the key idea about how we transform implementations to provide progress guarantees in the unknown-bound model. However, because this strategy essentially amounts to eliminating concurrency, it would not be practical if simply used as described. Therefore, our transformation does not employ this strategy until some process determines that it has executed the original operation too long without making progress.

The algorithm produced by applying our transformation to an obstruction-free algorithm *OFAlg* (which may include a contention manager) is shown in Figure 1. We now describe the transformed algorithm in more detail.

```
invoke(op)
N1:  if ¬PANIC
N2:      execute up to B steps of OFAlg
N3:      if op is complete
N4:          return response
N5:      PANIC ← true
         // panic mode
P1:  t ← fetch-and-increment(C)
P2:  A[i] ← 1
         repeat
P3:      T[i] ← t
         // find minimum time stamp; reset all others
P4:      m ← t
         k ← i
P5:      for each j ≠ i
P6:          s ← T[j]
P7:          if s < m
P8:              T[k] ← ∞
P9:              m ← s
                 k ← j
             else
P10:             if (s < ∞) T[j] ← ∞
P11:     if k = i
             repeat
P12:             execute up to b steps of OFAlg
P13:             if (op is complete)
P14:                 T[i] ← ∞
P15:                 PANIC ← false
P16:                 return response
P17:             A[i] ← A[i] + 1
P18:             PANIC ← true
P19:         until (T[i] = ∞)
         else
             repeat
P20:             a ← A[k]
P21:             wait a steps
P22:             s ← T[k]
P23:         until a = A[k] or s ≠ m
P24:         if (s = m) T[k] ← ∞
P25: until (op is complete)
```

Fig. 1. The transformation

The *PANIC* flag is used to regulate when the strategy to ensure progress should be used. When a process invokes an operation, it first checks this flag (N1) and, if it is *false*, executes up to B steps of its original algorithm (N2), where B is a parameter of the transformation. If these steps are sufficient to complete its operation, the process simply returns (N3–N4). Observe that, if every operation completes within B steps, then the *PANIC* flag remains *false*, so the transformed algorithm behaves exactly like the original one, except that it must read one variable, which is likely to be cached. Thus, by choosing B appropriately, we ensure that our transformation introduces very little overhead, if the original contention manager is effective.

If its operation fails to complete within B steps, a process sets the *PANIC* flag to *true* (N5). Thereafter, until the flag is reset, all new operations see that the *PANIC* flag is *true* and begin to participate in the strategy to ensure progress (P1–P25).

A process p_i participating in this strategy first acquires a timestamp (P1), initializes its activity counter $A[i]$ (P2), and then repeats loop P3–P25 until its operation is complete. In each iteration of this loop, p_i announces its timestamp in $T[i]$ (P3) and then searches for the minimum (i.e., oldest) timestamp announced by any process. All timestamps that are not ∞, but are larger than the minimum timestamp it observes, are replaced by ∞ (P4–P10).

If p_i determines that it has the minimum timestamp (P11), then it repeatedly takes up to b steps of the original algorithm (P12) (where the constant b is a parameter of the algorithm), increments its activity counter (P17), and resets the *PANIC* flag to *true* (P18). Note that the *PANIC* flag may have been set to *false* because some other process completed its operation (P15). Resetting the *PANIC* flag to *true* ensures that new operations continue to participate in the strategy to ensure progress. Process p_i repeats these steps until either its operation finishes (P13–P16) or some other process overwrites its timestamp with ∞ (P19). The latter case indicates that this other process has read an older timestamp (P8, P10) or thinks that p_i may have crashed (P24).

If process p_i determines that some other process p_k has the minimum timestamp (P11), then p_i enters loop P20–P23. During each iteration of this loop, p_i reads p_k's activity counter $A[k]$ (P20) and waits for the indicated number of steps (P21). If p_k's timestamp is overwritten during this period of time, then either p_k has completed its operation, another process thought that p_k had crashed, or another process saw an operation with a smaller timestamp. In this case, p_i exits the loop (P23). If p_k's timestamp is not overwritten by another value and p_k does not increment its activity counter during this period of time, then p_k may have crashed, so p_i exits the loop (P23) and overwrites p_k's timestamp with ∞ (P24).

In the next section, we present a careful proof of correctness for the resulting algorithm. Specifically, we show that, if process p_i has the smallest timestamp among all active processes with uncompleted operations, then p_i eventually completes its operation. Before doing so, we informally explain why our strategy ensures this property.

Eventually, in every iteration of loop P3–P25, process p_i enters loop P12–P19. Meanwhile, other processes determine that p_i's timestamp is the minimum and wait for a number of steps indicated by p_i's activity counter $A[i]$. If p_i doesn't complete its operation within b steps, then it increments its activity counter $A[i]$. Eventually, no process resets $T[i]$ to ∞, and $A[i]$ becomes large enough so that each process executing loop P20–P23 waits long enough at P21 so that p_i increments $A[i]$ during this period. Thus, eventually, all other active processes remain in loop P20–P23, so no process except p_i executes steps of the original algorithm. Hence, obstruction freedom guarantees that p_i eventually completes its operation.

On the other hand, if p_i crashes, then the other processes will no longer see $A[i]$ change, will stop waiting for p_i, and will overwrite its timestamp with ∞. Then the way is clear for the next operation in timestamp order (if any) to make progress.

An important feature of the transformed implementation is that, if the original contention manager is occasionally ineffective, causing the *PANIC* flag to be set, the *PANIC* flag will be reset and normal execution will resume, provided the original contention manager does not *remain* ineffective. To see this, recall that every operation by a noncrashed process eventually completes, and note that each operation either sees that *PANIC* is *false* and does not set it (N1), or sets it to *false* before returning (P15). Furthermore, *PANIC* is set to *true* only by an operation that has executed either B or b steps of the original algorithm (including the original contention manager) without completing. Thus, with appropriate choices for B and b, we ensure that our mechanism continues to be invoked only if the original contention manager continues to be ineffective.

4 Proof of Correctness

The transformed algorithm performs the original algorithm on the original shared objects and does not apply any other steps to those objects. Thus, the algorithm produced by applying our transformation to any obstruction-free algorithm retains the semantics of the original algorithm.

It remains to prove that the resulting algorithm is wait-free (assuming that there is a bound on the ratio of the maximum time and minimum time between steps of each process), that is, when the algorithm is executed in a system that satisfies the assumptions of the unknown-bound semisynchronous model, every operation executed by a process that does not crash eventually completes.

We begin with the following lemma, which we use in the wait-freedom proof.

Lemma 1. *If two different processes p_i and p_j are both in the loop at lines P12–P19, then either $T[i] = \infty$ or $T[j] = \infty$.*

Proof. Before reaching line P12, each process must set its entry in T to the timestamp of its operation at line P3. Consider the last time each process did so, and suppose, without loss of generality, that p_j did so before p_i did. Because p_i reached line P12, p_i must have set $T[j]$ to ∞ (or read that $T[j] = \infty$) after it

set $T[i]$, which was after the last time p_j set $T[j]$ to a finite value. Since no other process sets $T[j]$ to anything other than ∞, we have $T[j] = \infty$, as required.

The proof of wait-freedom is by contradiction. Suppose there is an execution in which some process takes an infinite number of steps after invoking an operation, but does not complete the operation. All the claims that follow are made within the context of this execution. Throughout the proof, we use v_i to denote the local variable v in the code of process p_i.

Any process that does not complete its operation within B steps must have seen (or set) $PANIC = true$. Then, on line P1, it must have applied fetch-and-increment to C and received a unique timestamp for this operation.

Let t^* be the minimum timestamp of any operation that does not complete even though the process p_{i^*} that invoked this operation takes an infinite number of steps. Thus, after some point in the execution, any process that gets a timestamp less than t^* for an operation it invokes either completes the operation or stops taking steps. Let X be any point in the execution after p_{i^*} first sets $T[i^*]$ to t^* on line P3 and such that, after X, no process takes a step of an operation with timestamp less than t^*.

Lemma 2. *Infinitely often, p_{i^*} is not in loop P12–P19.*

Proof. Suppose that there is some point Y after X in the execution such that after Y, p_{i^*} remains in loop P12–P19 forever. Note that p_{i^*} is the only process that can set $T[i^*]$ to a value other than ∞. However, p_{i^*} does not do so after Y. Thus, if $T[i^*]$ is set to ∞ after Y, then it will remain ∞. But then p_{i^*} will exit the loop, contrary to assumption. Hence, from Y onwards, $T[i^*] = t^*$.

By Lemma 1, any other process p_j that is in loop P12–P19 after Y has $T[j] = \infty$ and so, if it continues to take steps, would eventually exit this loop. Process p_j cannot re-enter this loop after leaving it: since $t_j > t^* = T[i^*]$, when p_j next reaches line P11, $k_j \neq j$. Thus, there is some point after Y in the execution after which no process except p_{i^*} performs steps in loop P12–P19.

Since the only line that sets $PANIC$ to *false* is P15, which p_{i^*} does not perform after Y (or else p_{i^*} would complete its operation), and since p_{i^*} sets $PANIC$ to *true* every iteration, eventually $PANIC$ remains *true*. Thus, eventually, no process will perform $N2$. Thus, there is some point after Y in the execution after which no process except p_{i^*} performs steps of *OFAlg*.

Note that, in each iteration of loop P12–P19, process p_{i^*} executes b steps of *OFAlg*. Since *OFAlg* is obstruction free, this implies that p_{i^*} eventually completes its operation and exits the loop by performing line P16. This contradicts the assumption that p_{i^*} remains in loop P12–P19 from Y onwards.

When p_{i^*} performs line P4 during its last operation, it sets m_{i^*} to t^*. Thereafter, m_{i^*} is always less than or equal to t^*, since the only other way m_{i^*} gets assigned a value is when lines P7–P9 are performed. After X, every process p_k with $T[k] < t^*$ takes no steps, so $A[k]$ does not change. Thus, if p_{i^*} enters loop P20–P23 (with $k_{i^*} \neq i^*$ and $m_{i*} < t^*$), it will eventually leave it, because $A[k_{i^*}]$ does not change. Furthermore, if no other process sets $T[k_{i^*}]$ to a different value

(either to the timestamp of a later operation performed by $p_{k_{i^*}}$ or to ∞), then p_{i^*} will set $T[k_{i^*}]$ to ∞ on line P24.

Because p_{i^*} takes an infinite number of steps and remains in loop P3–P25, but does not remain in loop P12–P19 or loop P20–P23 forever, p_{i^*} performs lines P3–P10 infinitely often. Thus, eventually, $T[j] > t^*$ for all $j \neq i$. In each iteration of loop P3–P25 starting after this point, p_{i^*} performs a successful test on line P11 and executes loop P12–P19. In each iteration of this inner loop, p_{i^*} increments $A[i^*]$. Thus $A[i^*]$ increases without bound. Furthermore, the number of steps performed between successive increments of $A[i^*]$ is bounded by a constant.

Eventually, from some point Z on, $T[j] > t^*$ for all $j \neq i^*$ and $A[i^*]$ is greater than R times the maximum number of steps between successive increments of $A[i^*]$, where R is the ratio of the maximum and minimum step times. If, after Z, process p_j begins loop P20–P23 with $k_j = i^*$, then, while it is waiting on line P21, process p_{i^*} increments $A[i^*]$. Hence, if p_j exits this loop, $s_j \neq m_j$ and p_j does not set $T[i^*]$ to ∞. The only other places that $T[i^*]$ can be set to ∞ are lines P8, P10, and P14. By assumption, process p_{i^*} does not complete its operation, and so does not execute P14. Note that after X, $T[i^*]$ contains only t^* or ∞. If process p_j reads $T[i^*] = \infty$ on line P6, then it performs no write on line P10. Otherwise process p_j reads $T[i^*] = t^*$ on line P6. In this case, it sets $m_j = t^*$ on line P9. After Z, $T[k] > t^*$ for all $k \neq i^*$, so process p_j does not write to $T[i^*]$ on line P8. Thus, eventually, no process writes ∞ to $T[i^*]$. Since process p_{i^*} writes t^* to $T[i^*]$ infinitely often, $T[i^*]$ has value t^* from some point onwards.

But this implies that, eventually, process p_{i^*} enters and never exits loop P12–P19. This contradicts Lemma 2.

5 Concluding remarks

We have shown that any obstruction-free algorithm can be transformed into a new algorithm that is wait-free when analyzed in the unknown-bound semisynchronous model of computation. Our transformation can be applied to an obstruction-free implementation, together with any valid contention manager, and the transformed implementation behaves like the original as long as the chosen contention manager is effective. Because real-world systems satisfy the assumptions of the model we consider, our result shows that obstruction-free algorithms and ad hoc contention managers can be used in practice without sacrificing the strong progress guarantees of wait-freedom.

In an earlier version of our transformation, a process incremented its activity counter only once in each iteration of the outer loop P3–P25, rather than each iteration of loop P12–P19. Hugues Fauconnier pointed out that this transformation applies only to *bounded* obstruction-free algorithms. These are algorithms having a finite bound such that every operation completes within that number of steps after it encounters no more interference. In contrast, the algorithm we present here can be applied even if an execution contains an operation and an infinite sequence of different configurations with increasing time requirements for

completion of that operation when running alone. In other words, our algorithm ensures that every operation executed by a noncrashed process eventually completes, even if the underlying obstruction-free algorithm only guarantees *eventual* completion after encountering no more interference.

Our result can easily be made stronger from both practical and theoretical points of view. First, as presented, our transformation introduces the need to know of the maximum number of processes that use the implementation. However, this disadvantage can easily be eliminated using results of Herlihy, Luchangco and Moir [17]. From a theoretical point of view, our use of the fetch-and-increment can be eliminated by using standard timestamping techniques based on an array of single-writer-multiple-reader registers. Thus, our transformation is applicable in a wide range of shared memory systems, as it does not depend on any special support for synchronization.

Acknowledgements: We are grateful to Hugues Fauconnier for his observation. Faith Ellen Fich is grateful for financial support from the Natural Sciences and Engineering Research Council of Canada and from the Scalable Synchronization Research Group of Sun Microsystems, Inc.

References

1. A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.
2. R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. Comput.*, 26(2):539–556, 1997.
3. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.
4. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.
5. H. Attiya and M. Mavronicolas. Efficiency of semisynchronous versus asynchronous networks. *Math. Syst. Theory*, 27(6):547–571, 1994.
6. J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.
7. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
8. D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28. ACM Press, 2005.
9. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
10. M. Fischer. Personal communication with Leslie Lamport. June 1985.
11. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, pages 374–382, 1985.
12. E. Gafni and L. Lamport. Disk Paxos. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 330–344, London, UK, 2000. Springer-Verlag.

13. E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. In *PODC '99: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–21. ACM Press, 1999.
14. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 258–264. ACM Press, 2005.
15. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
16. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, 2003.
17. M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium (CATS)*, 2003.
18. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
19. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 280–295, London, UK, 1994. Springer-Verlag.
20. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, CT, 1987.
21. V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA '03: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323. ACM Press, 2003.
22. N. Lynch, N. Shavit, A. Shvartsman, and D. Touitou. Timing conditions for linearizability in uniform counting networks. *Theor. Comput. Sci.*, 220(1):67–91, 1999.
23. N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 2–11. IEEE Press, 1992.
24. M. Mavronicolas, M. Papatriantafilou, and P. Tsigas. The impact of timing on linearizability in counting networks. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 684–688. IEEE Computer Society, 1997.
25. M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
26. W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In Moir and Shavit, editors, *In Proceedings of Workshop on Concurrency and Sycnhronization in Java Programs*, July 2004.
27. W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240–248. ACM Press, 2005.
28. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.