Synchronizing without Locks is Inherently Expensive

Hagit Attiya
Department of Computer Science, Technion
hagit@cs.technion.ac.il

Rachid Guerraoui
School of Computer and Communication Sciences, EPFL, and Computer Science and Artificial Intelligence Laboratory, MIT
rachid.guerraoui@epfl.ch

Danny Hendler*
Faculty of Industrial Engineering and Management, Technion
hendler@techunix.technion.ac.il

Petr Kouznetsov
Max Planck Institute for Software Systems
pkouznet@mpi-sws.mpg.de

ABSTRACT

It has been considered *bon ton* to blame locks for their fragility, especially since researchers identified *obstruction-freedom*: a progress condition that precludes locking while being weak enough to raise the hope for good performance. This paper attenuates this hope by establishing lower bounds on the complexity of obstruction-free implementations in *contention-free* executions: those where obstruction-freedom was precisely claimed to be effective. Through our lower bounds, we argue for an inherent cost of concurrent computing without locks.

We first prove that obstruction-free implementations of a large class of objects, using only *overwriting* or *trivial* primitives in contention-free executions, have $\Omega(n)$ space complexity and $\Omega(\log_2 n)$ (obstruction-free) step complexity. These bounds apply to implementations of many popular objects, including variants of fetch&add, counter, compare&swap, and LL/SC. When arbitrary primitives can be applied in contention-free executions, we show that, in any implementation of binary consensus, or any perturbable object, the number of distinct base objects accessed and memory stalls incurred by some process in a contention free execution is $\Omega(\sqrt{n})$. All these results hold regardless of the behavior of processes after they become aware of contention. We also prove that, in any obstruction-free implementation of a perturbable object in which processes are not allowed to fail their operations, the number of memory stalls incurred by some process that is unaware of contention is $\Omega(n)$.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*

General Terms

Algorithms, Performance, Theory

Keywords

lock-free implementations, obstruction-freedom, perturbable objects, step contention, memory contention, lower bound

1. INTRODUCTION

At the heart of many distributed systems are *shared objects*—data structures that may be concurrently accessed by many processes. These objects are often *implemented* in software, out of more elementary *base objects*. *Lock-free* implementations of shared objects require processes to coordinate without relying on mutual exclusion, thus avoiding the inherent problems of *locking*, e.g., deadlock, convoying, and priority-inversion.

Traditional lock-free algorithms are nonblocking, i.e., they guarantee progress (for at least one process) regardless of system conditions. This imposes significant computability and complexity charges: many objects do not have traditional lock-free implementations using only read/write base objects [4, 8, 15]. Even when the implementations are possible, they are typically complex and expensive [2, 3, 14].

Obstruction-freedom has been proposed as a progress property that conciliates the benefits of lock-freedom with the feasibility and performance requirements of modern concurrent computing [11, 12, 18]. Intuitively, obstruction-freedom guarantees progress only in situations in which the steps taken by concurrent processes are not interleaved, i.e., in the absence of *step contention* [1]. The idea is formalized by the *solo termination* property [6]: a process that takes sufficiently many steps on its own returns a value. In practice, step contention is considered rare [12], or at least can be made so through operating system support. That is, only one process is typically performing an operation on an implemented object while the rest of the processes are busy with other objects, swapped-out, failed, or simply idle.

*Supported in part by Sun Microsystems and by the Technion's Aly Kaufman Fellowship.

While it was easily established that any object has an obstruction-free implementation using read/write registers [1, 12], the performance benefits of obstruction-freedom as compared to traditional lock-free conditions (wait-freedom and nonblocking [10]) were never precisely determined.

This paper studies the complexity of obstruction-free implementations. Since obstruction-freedom is considered the weakest progress property that enforces lock-free implementations, our bounds capture the seemingly inherent cost of implementing concurrent objects without using locks.

While obstruction-freedom specifies that operations must terminate in the absence of step contention, it does not dictate how processes should behave when they do identify step contention. Thus, different obstruction-free implementations may cope with step contention in different ways: a process may wait for a while and then retry its operation, or it may retry the operation immediately; a process may retry the operation by using the same set of synchronization primitives, or it may do so by using a different set of (typically stronger) primitives; alternatively, a process may elect to simply fail the operation when it encounters step contention.

To cope with this diversity and capture the fundamental complexity of obstruction-free implementations, we focus on the performance in the *uncontended* cases, for which obstruction-freedom was precisely claimed to be effective [12]. More specifically, we consider *step-contention-free* executions, in which no process encounters step contention. Complexity lower bounds on such executions apply for all obstruction-free implementations, regardless of their behavior when step contention is identified.

We first prove that n-process obstruction-free implementations of a large class of objects using only *overwriting* or *trivial* primitives (e.g., *write*, *swap* and *read*) in step-contention-free executions have $\Omega(n)$ space complexity (Theorem 1) and $\Omega(\log_2 n)$ obstruction-free step complexity (Theorem 2). These bounds apply to well known *perturbable* objects [13], including, for example, fetch&add, modulo-b counter (for $b \geq 2n$), b-valued compare&swap (for $b \geq n$), and LL/SC bits.

We then prove that for any obstruction-free implementation of binary consensus, the number of distinct base objects accessed and memory *stalls*[1] incurred by some process in an execution in which *no process is aware of step contention* is $\Omega(\sqrt{n})$ (Theorem 3); in fact, the lower bound also holds for implementations of any perturbable object (Theorem 4).

Our aforementioned lower bounds are achieved in executions where none of the processes is aware of step contention. Thus they hold for any obstruction-free implementation of these objects, regardless of how processes behave if they become aware of step contention.

We also prove that, in any obstruction-free implementation of a perturbable object in which processes are not allowed to fail their operations, the number of memory stalls incurred by some process that is unaware of contention is $\Omega(n)$ (Theorem 5).

Viewed collectively, our results show that even the weakest known to date type of implementations that avoid using locks suffers from high time-complexity in uncontended executions. The conclusion is that the cost of avoiding deadlocking, convoying and priority-inversion is inherently high, at least for implementing perturbable objects.

[1]If an application of a nontrivial primitive by process p to base object r is preceded by k applications of nontrivial primitives to r performed by k distinct processes other than p, then p incurs a delay of length proportional to k. In other words, it incurs k stalls [5].

2. THE SHARED MEMORY SYSTEM MODEL

We consider a standard model of an asynchronous shared memory system, in which processes communicate by applying operations to shared objects.

An *object* is an instance of an abstract data type. It is characterized by a set of possible values and by a set of *operations* that provide the only means to manipulate it. No bound is assumed on the size of an object (i.e. the number of distinct values the object can take). An implementation of an object shared by a set **P** of n processes provides a specific data-representation for the object from a set **B** of shared *base objects*, each of which is assigned an initial value, and algorithms for each process in **P** to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects *primitive operations* or simply *primitives*. We reserve the term operations for the objects being implemented. We also say that an operation of an implemented object is *performed* and that a primitive is *applied to* a base object. The set of primitives supported by the objects we consider includes atomic read, write, and *read-modify-write* primitives [17]. We say that a primitive is *nontrivial* if it may change the value of the base object to which it is applied and *trivial* otherwise.

Let o be an object that supports two primitives f and f'. Following [6], we say that f *overwrites* f' on o, if starting from any value v of o, applying f' and then f results in the same value as applying just f, if f is applied with the same input parameters (if any) in both cases. A set of primitives is called *historyless* if all the nontrivial primitives in the set overwrite each other. Note that we require also that each such primitive overwrite itself. A set that includes the write and swap primitives is an example of a historyless set of primitives.

A *configuration* specifies the value of each base object and the state of each process. An *initial configuration* is a configuration in which all base objects have their initial values and all processes are in their initial states. When processes apply their operations to an implemented object, they perform a sequence of *step*s. Each step consists of some local computation and one shared memory *event*, which is a primitive applied to a base object. An event is *nontrivial* if it is an application of a nontrivial primitive.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* is an execution fragment that starts from an initial configuration, in which processes apply events and change states (based on the responses they receive from these events) according to their algorithm. For any finite execution fragment ω and any execution fragment ω', the execution fragment $\omega\omega'$ denotes the concatenation of ω and ω'. If $\omega\omega'$ is an execution, then the execution fragment ω' is called an *extension* of ω. We let $\omega|p$ denote the subsequence of events of execution ω that are applied by process p.

An *operation instance*, $\Phi = (\mathcal{O}, Op, p, args)$, is an application by process p of operation *Op* with arguments *args* to object $\mathcal{O}$. In an execution, each process performs a sequence of operation instances on the implemented object. To perform an operation instance, a process applies a sequence of one or more events, each of which *accesses* some base object.

If the last event of an operation instance Φ has been applied in an execution ω, we say that Φ *completes in* ω and that Φ *returns a response in* ω. The events applied by a process as it performs an operation instance can be interleaved with events applied by other processes as they apply their operation instances.

We say that a process p is *active* after ω if p is in the middle of performing some operation instance Φ, i.e. p has applied at least one event while performing Φ in ω, but Φ does not complete in ω.

If p is not active after ω, we say that p is *idle* after ω. We say that an execution ω is *Q-free*, for a non-empty set of processes $Q \subset \mathbf{P}$, if none of the events of ω is applied by any of the processes in Q. If $Q = \{q\}$, we say that ω is *q-free* instead of Q-free. Two executions are *indistinguishable* to a process p, if p applies exactly the same sequence of events and gets the same responses from these events in both executions.

If a process is active in the configuration resulting from a finite execution ω, the process has exactly one *enabled* event, i.e., the event the process is about to apply in the configuration. If a process is idle but has begun a new operation-instance, then the first event of that operation-instance is enabled; otherwise, it has no enabled event.

Let ω be an execution fragment. We say that ω is *step-contention-free for p* if the events of $\omega|p$ are contiguous in ω. We say that ω is *step-contention-free* if ω is step-contention-free for all processes.

An implementation is *obstruction-free* [1,11,12], if it guarantees that each process completes an operation instance within a finite number of its own steps if it runs in isolation long enough.

The *obstruction-free step complexity* of an implementation is the maximum number of events applied by any process p as it performs a single high-level operation, the maximum taken over all the implementation's executions that are step-contention-free for p.

3. TIME AND SPACE BOUNDS FOR SOLO-FAST IMPLEMENTATIONS

In this section we prove time and space lower bounds for obstruction-free implementations in which processes only apply nontrivial primitives from a historyless set (such as, e.g., a set that includes read, write and swap) when there is no step contention, but may fall back on more powerful primitives when step contention is identified. Such implementations are referred to in the literature as *solo-fast* [1, 16]. Our lower bounds hold for implementations of perturbable objects, defined next. (Our definition is equivalent to [13, Definition 3.1], when restricted to consider only deterministic implementations.)

DEFINITION 1. *An object $\mathcal{O}$ is* perturbable *if there is an operation instance op_n by process p_n, such that for any p_n-free execution $\alpha\lambda$ where no process applies more than a single event in λ, and for some process $p_l \neq p_n$ that applies no events in λ (if any), there is an extension of α, γ, consisting of events by p_l, such that p_n returns different responses when performing op_n by itself after $\alpha\lambda$ and after $\alpha\gamma\lambda$. We say that op_n* witnesses the perturbation *of $\mathcal{O}$.*

The following technical definition is required for our proofs.

DEFINITION 2. *A base object o is* covered after *an execution ω if the set of all the primitives applied to o in ω is historyless, and there is a process p_n that has, after ω, an enabled event e about to apply a nontrivial primitive from this set to o. We also say that e* covers o after E. *An execution ω is k*-covering *if*

- *ω is step-contention-free,*
- *there exists a set of processes $\{p_{j_1}, \ldots, p_{j_k}\}$ that does not contain process p_n, such that all the events of ω are applied by processes in this set and each of the processes in the set has an enabled nontrivial event that covers a distinct base object after ω.*

We call the set $\{p_{j_1}, \ldots, p_{j_k}\}$ a covering set of ω.

The second condition in Definition 2 implies that if an implementation has a k-covering execution, then its space complexity is at least k. We now prove a linear lower bound on the space complexity of any obstruction-free solo-fast implementation.

THEOREM 1. *Let A be an n-process obstruction-free implementation of a perturbable object $\mathcal{O}$ for which there exists a historyless set of primitives S such that any process p can apply only primitives from S in executions that are step-contention-free for p. Then the space complexity of A is at least $n - 1$.*

PROOF. Let op_n be the operation instance that witnesses the perturbation of $\mathcal{O}$. We prove the theorem by showing that A has an $(n-1)$-covering execution.

The proof goes by induction. The empty execution is vacuously a 0-covering execution. Assume that α_i, for $i < n-1$, is an i-covering execution with covering set $\{p_{j_1}, \ldots, p_{j_i}\}$. Let λ_i be the execution fragment that consists of the nontrivial events by processes $p_{j_1} \ldots p_{j_i}$ that are enabled after α_i, arranged in some arbitrary order.

From Definition 1, there is an execution fragment γ by some process $p_{j_{i+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_i}\}$ such that op_n returns different responses after executions $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$. We claim that γ contains a nontrivial event that accesses a base object not covered after α_i. Assume otherwise to obtain a contradiction. Since all events in executions $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$ apply primitives from a historyless set, every nontrivial primitive applied to a base object in γ is overwritten by some event in λ_i. Thus, the values of all base objects are the same after $\alpha_i\lambda_i$ and after $\alpha_i\gamma\lambda_i$. This implies that op_n must return the same response after both $\alpha_i\lambda_i$ and $\alpha_i\gamma\lambda_i$, which is a contradiction.

We extend α_i by letting $p_{j_{i+1}}$ execute the shortest prefix of γ at the end of which it has an enabled nontrivial event about to access an object o not covered after α_i. We denote this prefix of γ by γ'. We define α_{i+1} to be $\alpha_i\gamma'$. Thus, at the end of α_{i+1}, $p_{j_{i+1}}$ has an enabled nontrivial event that accesses o. As none of the processes $p_{j_1}, \ldots p_{j_i}$ apply events in γ', we have that α_{i+1} is a step-contention-free execution, after which processes $p_{j_1}, \ldots p_{j_{i+1}}$ have enabled events that cover distinct objects. Hence α_{i+1} is an $(i+1)$-covering execution. It follows that A has an $(n-1)$-covering execution. ☐

Next we prove a logarithmic lower bound on the obstruction-free complexity of solo-fast implementations of perturbable objects. As the proof is quite involved, we first provide an informal description of its technique and structure.

Our goal is to construct a scenario in which some process p_n has to access a large number of base objects as it runs solo while performing an operation. To that end, our proof constructs longer and longer r-covering executions. The construction proceeds in phases. After each phase r of the construction, we consider the path that p_n will take *if* it runs solo after we 'unfreeze' the pending covering events (but we don't actually unfreeze these events). We denote this path by π_r. Note that some of the objects along this path may already be covered after phase r.

To construct phase $r+1$, we deploy a 'free' process, $p_{j_{r+1}}$, and let it run solo. As processes can only apply primitives from a historyless set, and as the implemented object is perturbable, we know that $p_{j_{r+1}}$ will eventually be about to write to an uncovered object, O, along π_r. This, however, may have the undesirable effect (from the perspective of an adversary) of making π_{r+1} shorter than π_r: p_n may read the information written by $p_{j_{r+1}}$ to O (if we unfreeze its pending covering event) and not access some other objects farther along π_r!

Note, however, that objects that are part of π_r will be absent from π_{r+1} only if O *precedes them in* π_r. Thus the set of objects along

π_{r+1} that are covered (after phase $r+1$) is 'closer', in a sense, to the beginning of the path. It follows that if there are many phases r such that $|\pi_r|$ decreases, then one of the paths π_r must be 'long'.

To capture this intuition, we define Ψ, a monotonically-increasing potential function of the phase numbers. Ψ_r is a $(\log n)$-digit binary number defined as follows. Bit 0 (the most significant bit) of Ψ_r is 1 if and only if the first object in π_r is covered; bit 1 of Ψ_r is 1 if and only if the second object in π_r exists and is covered, and so on. Note that we do not need to consider paths that are longer than $\log_2 n$. If such a path exists, the lower bound clearly holds.

As mentioned before, to construct phase $r+1$, we deploy a free process, $p_{j_{r+1}}$, and let it run solo until it is about to write to an uncovered object, O, along π_r. In terms of Ψ, this implies that the covering event of $p_{j_{r+1}}$ might flip some of the digits of Ψ_r from 1 to 0. But O corresponds to a more significant digit, and this digit is flipped from 0 to 1, hence $\Psi_{r+1} > \Psi_r$ must hold. As we have $n-1$ processes to deploy, Ψ_r must increase $n-1$ times and eventually it equals $n-1$. When it does, the length of π_r is *exactly* $\log_2 n$. The formal proof follows.

THEOREM 2. *Let A be an n-process obstruction-free implementation of a perturbable object $\mathcal{O}$ for which there exists a historyless set of primitives S such that any process p can apply only primitives from S in executions that are step-contention-free for p. Then A has a step-contention-free execution in which a process accesses at least $\log_2 n$ distinct base objects in the course of performing a single operation instance.*

PROOF. If there is an execution in which a process accesses more than $\log_2 n$ distinct base objects in the course of performing a single operation instance in a step-contention-free manner then we are done. Assume otherwise. We construct a step-contention-free execution in which a process accesses *exactly* $\log_2 n$ distinct base objects in the course of performing a single operation instance.

The construction proceeds in at most n phases. In phase $r \geq 0$, we construct an execution $\alpha_r\delta_r\phi_r$ with the following structure:

- α_r is an r-covering execution with a covering set $p_{j_1}, \ldots, p_{j_r}$,
- in δ_r, each of the processes $p_{j_1}, \ldots, p_{j_r}$ applies a nontrivial event to an object that is covered after α_r, and
- in ϕ_r, process p_n runs solo after $\alpha_r\delta_r$ until it completes the operation instance op_n.

Let $C(\alpha_r)$ denote the set of base objects that are covered after α_r. Let $\pi_r = O_r^1 \ldots O_r^{i_r}$ denote the sequence of all distinct base objects accessed by p_n in ϕ_r (after $\alpha_r\delta_r$) indexed according to the order in which they are first accessed by p_n. Also let S_{π_r} denote the set of these base objects.

In execution $\alpha_r\delta_r\phi_r$, p_n accesses i_r distinct base objects. Thus, it suffices for the proof to construct such an execution with $i_r = \log_2 n$. For $j \in \{1, \ldots, i_r\}$, we let b_r^j be the indicator variable whose value is 1 if $O_r^j \in C(\alpha_r)$ and 0 otherwise. We associate an integral progress parameter, Ψ_r, with each phase $r \geq 0$, defined as follows:

$$\Psi_r = \sum_{j=1}^{i_r} b_r^j \cdot 2^{\log_2 n - j} \; . \tag{1}$$

As we assume that $i_r \leq \log_2 n$ for all r, Ψ_r can be viewed as a $\log_2 n$-digit number in base 2 whose j'th most significant bit is 1 if the j'th object in π_r exists and is in $C(\alpha_r)$, or 0 otherwise. This implies that the number of base objects in π_r that are covered after α_r equals the number of 1-bits in Ψ_r.

We now describe our construction. Let α_0 and δ_0 denote the empty execution; let ϕ_0 denote the solo execution that results when p_n performs the operation instance op_n starting from an initial configuration, and let i_0 denote the number of distinct objects accessed in ϕ_0. Since $C(\alpha_0) = \emptyset$, we have $\Psi_0 = 0$. Suppose that, for some r, $0 \leq r < n-1$, we have constructed $\alpha_r\delta_r\phi_r$ and $\Psi_r < n-1$.

As $\mathcal{O}$ is perturbable with operation instance op_n witnessing that, there is an execution fragment γ_{r+1} by some process $p_{j_{r+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_r}\}$ such that op_n returns different responses to p_n after executions $\alpha_r\delta_r$ and $\alpha_r\gamma_{r+1}\delta_r$. We claim that, in γ_{r+1}, $p_{j_{r+1}}$ applies a nontrivial event to an object in $S_{\pi_r} \setminus C(\alpha_r)$. Assume that γ_{r+1} contains no nontrivial events to objects in $S_{\pi_r} \setminus C(\alpha_i)$ to obtain a contradiction. As $\alpha_r\gamma_{r+1}$ is step-contention-free, all the events of γ_{r+1} either access base objects not in S_{π_r} or are overwritten by the events of δ_r. It follows that $\alpha_r\gamma_{r+1}\delta_r\phi_r$ is also an execution of A and that $\alpha_r\delta_r\phi_r$ and $\alpha_r\gamma_{r+1}\delta_r\phi_r$ are indistinguishable to p_n. This implies that op_n must return the same responses after both executions, which is a contradiction.

Let γ'_{r+1} be the shortest prefix of γ_{r+1} after which $p_{j_{r+1}}$ has an enabled event, e, about to apply a nontrivial event to a base object $O_r^k \in S_{\pi_r} \setminus C(\alpha_r)$. Define $\alpha_{r+1} = \alpha_r\gamma'_{r+1}$, $\delta_{r+1} = \delta_r e$ and let ϕ_{r+1} denote the execution fragment in which p_n applies events by itself after $\alpha_{r+1}\delta_{r+1}$ as it performs the operation instance op_n to completion. It is easily verified that α_{r+1} is an $(r+1)$-covering execution and that $C(\alpha_{r+1}) = C(\alpha_r) \cup O_r^k$.

We claim that $\Psi_{r+1} > \Psi_r$ holds. As $O_r^k \notin C(\alpha_r)$, we have $b_r^k = 0$. As the values of objects $O_r^1 \cdots O_r^{k-1}$ are the same after $\alpha_r\delta_r$ and $\alpha_{r+1}\delta_{r+1}$, it follows that $b_r^j = b_{r+1}^j$ for $j \in \{1, \ldots, k-1\}$. This implies, in turn, that $O_r^k = O_{r+1}^k$. As $O_{r+1}^k \in C(\alpha_{r+1})$, we have $b_{r+1}^k = 1$. We get:

$$\begin{aligned}
\Psi_{r+1} &= \textstyle\sum_{j=1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&= \textstyle\sum_{j=1}^{k-1} b_{r+1}^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} + \\
&\quad \textstyle\sum_{j=k+1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&= \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} + \\
&\quad \textstyle\sum_{j=k+1}^{i_{r+1}} b_{r+1}^j \cdot 2^{\log_2 n - j} \\
&\geq \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + 2^{\log_2 n - k} \\
&> \textstyle\sum_{j=1}^{k-1} b_r^j \cdot 2^{\log_2 n - j} + \sum_{j=k+1}^{i_r} b_r^j \cdot 2^{\log_2 n - j} \\
&= \Psi_r.
\end{aligned}$$

By definition, we have $0 \leq \Psi_r \leq n-1$ for all r. Furthermore, just a single process joins the execution in each phase. As we've shown that Ψ is monotonically increasing with with r, this implies that we eventually reach a phase r^* with $\Psi_{r^*} = n-1$. This implies in turn that $i_{r^*} = \log_2 n$. $\square$

4. TIME BOUNDS FOR IMPLEMENTATIONS USING ARBITRARY PRIMITIVES

In Section 3 we considered obstruction-free implementations that can only apply synchronization primitives from a restricted set in step-contention-free executions; the metric that we used counted the worst-case number of steps made by a process in such executions.

In this section, we investigate obstruction-free implementations that can use *arbitrary* primitives even in step-contention-free executions. The metric that we use here counts both the number of steps made by a process and the number of stalls it incurs as a result of memory contention with other processes. The following definition formalizes the notion of a stall. It captures the fact that when multiple processes apply non-trivial operations simultaneously to the same base object, these operations are being serialized.

DEFINITION 3. *Let e be an event applied by a process p as it performs an operation instance Φ in execution ω. Let r be the base object accessed by e. Also let $\omega = \omega_0 e_1 \cdots e_k e \omega_1$, where $e_1 \cdots e_k$ is a maximal sequence of $k \geq 1$ consecutive nontrivial events, by distinct processes other than p, that access r. Then we say that* Φ incurs k memory stalls in ω on account of e. *The* number of memory stalls incurred by Φ in ω *is the sum of memory stalls Φ incurs in ω over all the events of Φ in ω.*

Let p be a process and consider the set of executions $\mathcal{E}_p$ that are indistinguishable to p from an execution that is step-contention-free to p. This is a superset of all the executions that are step-contention-free to p. From obstruction freedom, p must make progress in any execution of $\mathcal{E}_p$. Let $\omega \in \mathcal{E}_p$ be an execution and let e be an event of p that is enabled after ω. We say that e is *issued while p is unaware of step contention*. It might be that p becomes aware of step contention when it receives the response of e. Nevertheless, the delay incurred by p until it becomes aware of step contention includes the delay it incurs on account of e.

In a similar manner, we let $\mathcal{E}$ denote the set of executions that are indistinguishable to *all* processes from a step-contention-free execution. Let $\omega \in \mathcal{E}$ be an execution and let e be an event that is enabled after ω. We say that e is *issued while no process is aware of step contention*.

In the proofs that follow we consider the worst-case time complexity incurred by processes on account of the events they issue while being unaware of step contention.

4.1 A $\sqrt{n}$ Lower Bound

In this section we prove an $\Omega(\sqrt{n})$ time lower bound on obstruction-free implementations of binary consensus and perturbable objects. This bound holds for all obstruction-free implementation of these objects, regardless of how processes behave when they encounter step contention. It implies that, for these implementations, a process can be made to incur a delay of length $\Omega(\sqrt{n})$ before *any process* becomes aware of step contention.

A *binary consensus* object supports a single operation called *decide* with input value from the domain $\{0, 1\}$. Every process can call the *decide* operation at most once. An implementation of consensus is correct if the following two conditions hold for every execution ω.

Consistency: The responses of all the instances of *decide* that complete in ω are equal.

Validity: If an operation instance returns response v in ω, then v is the input value of a decide operation instance by some process in ω.

THEOREM 3. *Let A be an n-process obstruction-free implementation of binary consensus. Then there is an execution ω of A and a process p such that the sum of events issued by p in ω while no process is aware of step contention and the stalls it incurs on account of these events is at least $\sqrt{n}$.*

PROOF. Consider executions of A in which processes $p_1, \ldots, p_{n-1}$ perform instances of *decide* with input 0 and process p_n performs an instance of *decide* with input 1. Let ϕ be the execution in which, starting from the initial configuration, p_n performs its *decide* instance to completion. Let B denote the set of base objects that are accessed in ϕ. If $|B| \geq \sqrt{n}$ then we are done. Assume otherwise.

We construct a p_n-free execution at the end of which there is a subset of processes $S \subset \{p_1, \cdots, p_{n-1}\}$ of size exactly $\sqrt{n}$, all the processes of which have enabled nontrivial events about to access the same object in B. The execution is constructed inductively in at most $n-1$ phases. We denote the execution constructed in phases $1, \cdots, i$ by ω_i. Our construction maintains the following invariants for all $i \leq n-1$:

- ω_i is step-contention-free,
- all the events of ω_i are applied by processes in $\{p_1, \ldots, p_i\}$,
- ω_i does not contain any nontrivial event applied to an object in B, and
- each of the processes $p_1, \cdots, p_i$ has a nontrivial event to a base object in B that is enabled at the end of ω_i.

We let ω_0 denote the empty execution. It is easily verified that the above invariants are vacuously met by ω_0. Assume we have constructed ω_i, for $i < n-1$, and that the number of enabled nontrivial events about to access any single object in B at the end of ω_i is less than $\sqrt{n}$. We now describe the construction of ω_{i+1}. We let process p_{i+1} perform its instance of *decide* by itself after ω_i until it either has an enabled nontrivial event about to access an object in B, or its *decide* instance completes.

We show that the latter cannot occur. Assume otherwise to obtain a contradiction. From the validity requirement, p_n's instance of *decide* returns response 1 in ϕ. Let σ_{i+1} be the execution in which p_{i+1} performs its *decide* instance after ω_i until it completes. As $\omega_i\sigma_{i+1}$ is p_n-free, we get from the validity requirement that p_{i+1}'s instance of *decide* returns response 0 in $\omega_i\sigma_{i+1}$.

From the induction hypothesis applied to ω_i, and as we assume that no nontrivial event was applied to an object in B in σ_{i+1}, $\omega_i\sigma_{i+1}\phi$ is an execution that is indistinguishable from ϕ to p_n. It follows that the responses of the instances of *decide* by p_{i+1} and p_n in $\omega_i\sigma_{i+1}\phi$ are 0 and 1, respectively. This contradicts the consistency requirement.

Thus, at the end of ω_{i+1}, process p_{i+1} has an enabled nontrivial event about to access a base object in B. From the induction hypothesis applied to ω_i, we get that at the end of ω_{i+1}, each of $p_1, \cdots, p_{i+1}$ has an enabled nontrivial event about to access an object in B, and that ω_{i+1} is a step-contention-free execution that contains no nontrivial event applied to an object in B.

As $|B| < \sqrt{n}$, there is a phase j, $j \leq n-1$, such that after ω_j there exist at least $\sqrt{n}$ processes, all of which have enabled nontrivial events about to access the same object $o \in B$. Let α be some ordering of these events. Also let β be the longest prefix of ϕ that does not access o, and let e be p_n's enabled event after β. Then p_n incurs at least $\sqrt{n}$ memory stalls in $\omega_j\beta\alpha e$. To conclude the proof, we note that $\omega_j\beta$ is step-contention-free and that each of the events in αe is by a different process. Thus all the events of $\omega_j\beta\alpha e$ are issued while no process is aware of step contention. □

The proof of the following theorem is build along the lines of that of Theorem 3.

THEOREM 4. *Let A be an n-process obstruction-free implementation of a perturbable object. Then there is an execution ω of A and a process p such that the sum of events issued by p in ω while no process is aware of step contention and the stalls it incurs on account of these events is at least $\sqrt{n}$.*

PROOF. Let op_n be the operation instance that witnesses the perturbation of $\mathcal{O}$. Let ϕ be the execution of A in which, starting from the initial configuration, p_n performs op_n until it completes it. Let B denote the set of base objects that are accessed in ϕ. If $|B| \geq \sqrt{n}$ then we are done. Assume otherwise.

We construct a p_n-free execution at the end of which there is a subset of processes $S \subset \{p_1, \cdots, p_{n-1}\}$ of size exactly $\sqrt{n}$, all the processes of which have enabled nontrivial events about to access the same object in B. The execution is constructed inductively in at most $n-1$ phases. We denote the execution constructed in phases $1, \ldots, i$ by ω_i. Our construction maintains the following invariants for all $i \leq n-1$:

- ω_i is step-contention-free,
- ω_i does not contain any nontrivial event applied to an object in B, and
- there exists a set of processes $\{p_{j_1}, \ldots, p_{j_i}\}$ that does not contain p_n, such that
  - each of these processes has an enabled nontrivial event about to access a base object in B after ω, and
  - none of the events of ω_i are applied by processes not in $\{p_{j_1}, \ldots, p_{j_i}\}$.

We let ω_0 denote the empty execution. It is easily verified that the above invariants are vacuously met by ω_0. Assume we have constructed ω_i, for $i < n-1$, and that the number of enabled nontrivial events about to access any single object in B at the end of ω_i is less than $\sqrt{n}$. We now describe the construction of ω_{i+1}. From the induction hypothesis applied to ω_i, no process has applied a nontrivial event in ω_i to an object in B.

Let δ denote the execution fragment that consists of the events by $\{p_{j_1}, \ldots, p_{j_i}\}$ that are enabled after ω_i. As $\mathcal{O}$ is perturbable with operation instance op_n witnessing that, there is an execution fragment γ by some process $p_{j_{i+1}} \notin \{p_n, p_{j_1}, \ldots, p_{j_i}\}$ such that op_n returns different responses to p_n after executions $\omega_i\delta$ and $\omega_i\gamma\delta$. We claim that $p_{j_{i+1}}$ applies in γ a nontrivial event to an object in B. Assume otherwise to obtain a contradiction. Then from the induction hypothesis and our assumption, $\omega_i\gamma\delta\phi$ is an execution that is indistinguishable to p_n from $\omega_i\delta\phi$. It follows that the responses of op_n are the same in $\omega_i\gamma\delta\phi$ and $\omega_i\delta\phi$. This is a contradiction.

Let γ' be the shortest prefix of γ after which $p_{j_{i+1}}$ has an enabled nontrivial event about to access a base object in B. We let ω_{i+1} be $\omega_i\gamma'$. Thus, from the induction hypothesis applied to ω_i, we get that at the end of ω_{i+1} each of $p_{j_1}, \cdots, p_{j_{i+1}}$ has an enabled nontrivial event about to access an object in B, and that ω_{i+1} is a step-contention-free execution that contains no nontrivial event applied to an object in B.

As $|B| < \sqrt{n}$, there is a phase k, $k \leq n-1$, such that after ω_k there exist at least $\sqrt{n}$ processes, all of which have enabled nontrivial events about to access the same object $o \in B$.

Let α be some ordering of these events. Also let β be the longest prefix of ϕ that does not access o, and let e be p_n's enabled event after β. Then p_n incurs at least $\sqrt{n}$ memory stalls in $\omega_k\beta\alpha e$ on account of e. To conclude the proof, we note that $\omega_k\beta\alpha e$ and $\omega_k\alpha\beta e$ are indistinguishable to p_n and that $\omega_k\alpha\beta e$ is step-contention-free for p_n. $\square$

4.2 A Linear Lower Bound For Non-Failing Implementations

We say that an obstruction-free implementation is *non-failing* if processes are not allowed to fail their operations when they become aware of step contention. For such implementations we obtain a stronger bound than that obtained in Section 4.1.

Fich, Hendler, and Shavit [7] prove a lower bound of $n-1$ on the worst-case number of stalls incurred by a process as it performs a single operation instance. This bound holds for non-failing obstruction-free implementations of objects in a class $\mathcal{G}$, that includes counter and single-writer snapshot objects. It can be shown that the same lower bound holds for any perturbable object. In the following, we prove that this bound holds in an execution in which all the events of the process whose operation instance incurs the linear complexity are issued while it is not aware of step contention.

The following definition of *k-stall-execution* is taken from [7] with minor terminology adaptation.

DEFINITION 4. *An execution $\omega\sigma_1 \cdots \sigma_i$ is a k-*stall execution *for process p if*

- *ω is p-free,*
- *there are distinct base objects $O_1, \ldots, O_i$ and disjoint sets of processes $S_1, \ldots, S_i$ whose union does not include p and has size k such that, for $j = 1, \ldots, i$,*
  - *each process in S_j has an enabled nontrivial event about to access O_j after ω, and*
  - *in σ_j, process p applies events by itself until it is first about to apply an event to O_j, then each of the processes in S_j applies an event that accesses O_j, and, finally, p applies an event that accesses O_j,*
- *all processes not in $S_1 \cup \cdots \cup S_i$ are idle after ω,*
- *p starts at most one operation instance in $\sigma_1 \cdots \sigma_i$, and*
- *in every $(\{p\} \cup S_1 \cup \cdots \cup S_i)$-free extension of ω, no process applies a nontrivial event to any base object accessed in $\sigma_1 \cdots \sigma_i$.*

In a k-stall execution for process p, p incurs k stalls, since it incurs $|S_j|$ stalls when it applies its first event to O_j, for $j = 1, \ldots i$. The results of [7] are obtained by proving that non-failing obstruction-free implementations of objects such as those mentioned above have $n-1$ stall executions for any process. Our contribution lies in the following technical lemma. It shows that a process p is not aware of step-contention in a k-stall execution for p.

LEMMA 1. *Let ω be a k-stall execution for process p. Then all of p's events in ω are issued while p is unaware of step contention.*

PROOF. Let $\omega\sigma_1 \ldots \sigma_i$ be a k-stall execution for process p for some $k > 0$. For $j = 1, \ldots, i$, let S_j and O_j be as in Definition 4. For an execution σ, let $\sigma|\overline{p}$ be the subsequence of events in σ that are applied by processes other than p.

We prove that the sequence of events $\theta = \omega(\sigma_1|\overline{p}) \cdots (\sigma_i|\overline{p}) (\sigma_1|p) \cdots (\sigma_i|p)$ is an execution and that it is indistinguishable from $\omega\sigma_1 \ldots \sigma_i$ to all processes. Since θ is step-contention-free for p, this will establish that all of p's events in $\omega\sigma_1 \ldots \sigma_i$ are issued while p is unaware of step contention.

The proof goes by double induction. For $l = 0, \ldots, i$, let ω_l denote the execution $\omega\sigma_0 \ldots \sigma_l$. The outer induction is on the executions ω_l. We prove that, for $l = 0, \ldots, i$, $\theta_l = \omega(\sigma_1|\overline{p}) \cdots (\sigma_l|\overline{p}) (\sigma_1|p) \cdots (\sigma_l|p)$ is an execution that is indistinguishable to all processes from ω_l. The claim holds vacuously for $l = 0$. For $l < i$, assume that θ_l is an excution that is indistinguishable from ω_l to all processes.

Consider the sequence of events $\sigma_{l+1}|\overline{p}$. From Definition 4, these events are enabled at the end of ω_l. Consequently, from outer induction hypothesis, they are also enabled at the end of θ_l. As all the events of $(\sigma_1|p) \cdots (\sigma_l|p)$ are applied by p, the events of $\sigma_{l+1}|\overline{p}$ are enabled at the end of $\omega(\sigma_1|\overline{p}) \cdots (\sigma_l|\overline{p})$. Additionally, as each of the events of $\sigma_{l+1}|\overline{p}$ is applied by a distinct process in S_{l+1}, $\omega(\sigma_1|\overline{p})\cdots(\sigma_{l+1}|\overline{p})$ is an execution.

From outer induction hypothesis, all processes in $S_1\cup\cdots\cup S_l$ apply the same events and get the same responses in $\omega(\sigma_1|\overline{p})\cdots(\sigma_l|\overline{p})$ and ω_l. As all the events of $\sigma_{l+1}|\overline{p}$ access O_{l+1} and none of the events of $\sigma_1\cdots\sigma_l$ accesses O_{l+1}, it follows that all processes in $S_1\cup\cdots\cup S_{l+1}$ apply the same events and get the same responses in ω_{l+1} and in $\omega(\sigma_1|\overline{p})\cdots(\sigma_{l+1}|\overline{p})$, and hence also in θ_{l+1}.

We next show that θ_{l+1} is an execution and that p gets the same responses from the events it applies in it as in ω_{l+1}. We show this by inner induction on the number of events, m, applied by p in $(\sigma_1|p)\cdots(\sigma_{l+1}|p)$.

The claim is obvious for $m=0$. Assume that $\sigma_1^p\cdots\sigma_{l+1}^p$ consists of $m>0$ events and that the claim holds for the first $m-1$ events. Let e be the m'th event. Two cases exist. If e accesses a base object $O\notin\{O_1,\ldots,O_{l+1}\}$, then, from Definition 4, O is not accessed in $\omega\sigma_1\cdots\sigma_{l+1}$ by any process other than p. Thus, from the inner induction hypothesis, O has the same value when e accesses it in both ω_l and θ_l. Otherwise, suppose that $O=O_j$ for some $j\in\{1,\ldots,l+1\}$. The subsequence of events that precede e in accessing O_j is $(\sigma_j|\overline{p})$ in both ω_{l+1} and θ_{l+1}. Consequently, from inner and outer induction hypotheses, O has the same value when accessed by e in both ω_{l+1} and θ_{l+1}. It follows that, in both cases, e returns the same response in ω_l. Hence also p applies the same events, and gets the same responses from these events, in both ω_l and θ_l.

As all processes apply the same events, and get the same responses from these events in both ω_l and θ_l, and as ω_l is an execution, it follows that θ_l is also an execution. This concludes the proof of the lemma. □

The proof of Theorem 6 in [7] can be used to establish that any non-failing obstruction-free n-process implementation of a perturbable object has an $(n-1)$-stall execution for any process that shares the implementation. Combining that with Lemma 1 gives the following.

THEOREM 5. *Let A be an n-process non-failing obstruction-free implementation of a perturbable object. Then for any process p there is an execution ω of A such that p incurs in ω at least $n-1$ stalls on account of events that it issues while it is unaware of step contention, as it performs a single operation instance.*

5. RELATED WORK

Solo-fast implementations that use only reads and writes when there is no step contention, but may fall back on more powerful primitives when step contention is encountered, are considered by Luchangco et al. [16] and by Attiya et al. [1]. There is a universal solo-fast implementation [1], with linear obstruction-free step complexity and space complexity. Our lower bounds provide a partial positive answer to the open question (posed in [1]) of whether the high complexity price of this universal implementation is inherent. Our results show that, although reads and writes are considered comparatively cheap [16], solo-fast implementations are not scalable, because a process that runs by itself may have to apply $\Omega(\log n)$ reads and writes in the course of performing a single operation.

Jayanti, Tan, and Toueg [13] obtain linear time and space lower bounds for solo-terminating implementations of perturbable objects from *historyless* base objects, i.e., objects that only support nontrivial primitives that overwrite each other. Specifically, they obtain a worst-case lower bound of $n-1$ on the number of steps taken by a process as it performs a single operation. As observed by [1], a simple reduction to [13] implies the same lower bound for obstruction-free implementations of perturbable objects. The execution constructed by [13] to obtain this bound is not necessarily step-contention-free, however. Thus their result does not imply any lower bound on solo-fast implementations, as processes may apply strong synchronization primitives in the execution they construct.

Hendler and Shavit [9] consider nonblocking implementations of a class of objects that includes all of the well-know perturbable objects mentioned above. They prove an $\Omega(\sqrt{n})$ lower bound on the number of distinct base objects accessed and memory stalls incurred by a process as it performs a single operation. Their bound holds for nonblocking implementations but is obtained in executions in which the process that incurs this complexity may be aware of step contention.

A recent paper by Fich et al. [7] considers n-process obstruction-free implementations of objects, such as a modulo-m counter (for $m\geq n$) and single-writer snapshot, that can use arbitrary primitives. They show a bound of $n-1$ on the number of stalls incurred by a process as it performed a single operation. It can be shown that this result holds for all perturbable objects. We actually prove that their bound is obtained in an execution in which the process that incurs the linear complexity is not aware of step contention. Thus, even when processes can apply arbitrary primitives, a process can incur linear complexity and still 'believe' it runs in isolation (Theorem 5). This result, as well as the result of [7], holds for obstruction-free implementations where processes do not fail their operations when they encounter step contention. This follows from the fact that, similarly to the proofs of Jayanti et al. [13], the linear bound is obtained in executions that are not necessarily step-contention-free.

The potential-function technique that we use to prove a logarithmic lower bound on the obstruction-free complexity of solo-fast implementations of perturbable objects is an extension of a proof technique originated in [7]. A major challenge by our proof is that here, unlike in [7], once a process covers a base object along a current path, that process cannot be used again in a later phase, because it may then become aware of step contention. A key novelty of our technique is in extending the potential-function argument so that it can handle this type of, in a sense, *one-shot* covering scenarios.

6. SUMMARY

We prove lower bounds on the cost of obstruction-free implementation of shared objects. We do so by focusing on the complexity of obstruction-free implementations in uncontended executions (which are argued to be the most frequent in practice), without restricting the behavior of the processes in contended situations where processes might be using locks, randomization or other expensive mechanisms.

By measuring the complexity of the weakest form of lock-free implementations known to date, our results capture the seemingly inherent cost of preventing deadlock, convoying, and priority-inversion.

7. REFERENCES

[1] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, 2005.

[2] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems (ICDCS'93)*, pages 264–273, 1993.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[4] D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[5] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[6] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.

[7] F. E. Fich, D. Hendler, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS)*, 2005.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.

[9] D. Hendler and N. Shavit. Operation-valency and the cost of coordination. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 84–91, 2003.

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

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[12] M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 522–529, 2003.

[13] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[14] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 130–140, 1994.

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

[16] V. Luchangco, M. Moir, and N. Shavit. On the uncontended complexity of consensus. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, pages 45–59, 2003.

[17] E. Ruppert. Determining consensus numbers. *SIAM Journal of Computing*, 30(4):1156–1168, 2000.

[18] M. L. Scott and W. N. Scherer III. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.