Computing with Reads and Writes in the Absence of Step Contention (Extended Abstract)

Hagit Attiya[1], Rachid Guerraoui[2], and Petr Kouznetsov[2]

[1] Department of Computer Science, Technion
[2] School of Computer and Communication Sciences, EPFL

Abstract. This paper studies implementations of concurrent objects that exploit the absence of *step contention*. These implementations use only reads and writes when a process is running solo. The other processes might be busy with other objects, swapped-out, failed, or simply delayed by a contention manager. We study in this paper two classes of such implementations, according to how they handle the case of step contention. The first kind, called *obstruction-free* implementations, are not required to terminate in that case. The second kind, called *solo-fast* implementations, terminate using powerful operations (e.g., C&S).

We present a generic obstruction-free object implementation that has a linear contention-free step complexity (number of reads and writes taken by a process running solo) and uses a linear number of read/write objects. We show that these complexities are asymptotically optimal, and hence generic obstruction-free implementations are inherently slow. We also prove that obstruction-free implementations cannot be *gracefully degrading*, namely, be nonblocking when the contention manager operates correctly, and remain (at least) obstruction-free when the contention manager misbehaves.

Finally, we show that any object has a *solo-fast* implementation, based on a solo-fast implementation of consensus. The implementation has linear contention-free step complexity, and we conjecture solo-fast implementations must have non-constant step complexity, i.e., they are also inherently slow.

1 Introduction

At the heart of many distributed systems are *shared objects*—data structures that are concurrently accessed by many processes. Often, these objects are *implemented* in software, out of more elementary *base objects*. *Lock-free* implementations of such objects do not rely on mutual exclusion or locking, and thereby allow processes to overcome adverse operating systems affects. This includes both *wait-free* algorithms, in which *every* process completes its operations in a finite number of steps, and *nonblocking* algorithms, where *some* process completes an operation in every sufficiently long execution [15]. The safety property typically required from both nonblocking and wait-free implementations is *linearizability* [15,18]; roughly, every operation on the object should appear instantaneous.

Although they provide very attractive guarantees, lock-free implementations were claimed to have limited usability. This is because nonblocking implementations of many objects are often impossible, e.g., when only read/write objects are available [10,12,23]. Even when the implementations are possible, which can be achieved under specific timing assumptions (e.g., encapsulated within failure detector abstractions), or using strong synchronization operations (like C&S), these implementations are typically complex and expensive [7,9,20]. The complexity and computability price paid by lock-free algorithms often originates in situations in which there is *step contention*, i.e., steps of concurrent processes are interleaved.

In this paper, we study implementations that exploit the fact that in practice, step contention is rare, or at least can be made so through operating system support. That is, only one process is typically performing visible (non local) steps within any object operation, whereas the rest of the processes are busy with other objects, swapped-out or failed. The absence of step contention does not preclude common scenarios where other processes have pending operations on the same implemented object but are not accessing the base objects. This is fundamentally different from alternative contention metrics: *point* contention [5] and *interval* contention [2]; both count also failed or swapped-out processes. (See the scenario presented in Figure 1.)

We first study *obstruction-free* implementations that guarantee termination only in the absence of step contention. This is formalized by the *solo termination* property [11]: a process that takes sufficiently many steps on its own returns a value. Clearly, obstruction-free implementations cannot rely on mutual exclusion or locks, and hence, they are lock-free. On the other hand, implementations that would guarantee termination only in the absence of interval (or point) contention can be obtained using locks. Whereas all nonblocking implementations are obstruction-free, the converse is not necessarily true, however, since obstruction-free implementations may incur scenarios (when there is step contention) in which no process is able to complete its operation in a finite number of steps.

An obstruction-free implementation has to provide a *legal* response if it returns at all, but termination is required only under very restricted conditions.

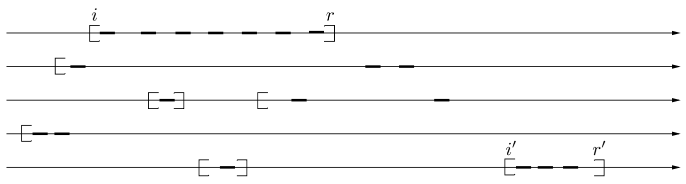

Fig. 1. An example illustrating types of contention: Operation $[i, r]$ has interval contention 5, point contention 4, and step contention 3; operation $[i', r']$ has interval and point contention 4, and step contention 1 ($[i', r']$ is step contention-free). (Square brackets denote invocations and responses, while solid intervals denote steps on base objects.)

One contribution of this paper is to disambiguate the behavior of an obstruction-free implementation when an operation cannot return a legal response. In the presence of step contention, an operation may return control to a higher-level entity, which we call the *client*. Ideally, the obstruction-free implementation should only be allowed to return a *fail* indication to the client, enabling it to choose whether to re-invoke the same operation, or to invoke another operation. We show however that there is inherent uncertainty as to whether the operation could have had an effect on the object or not, by reduction to wait-free consensus. This implies that the implementation must sometimes return a special *pause* value, indicating that the client should re-invoke the same operation. We extend the notion of linearizability so as to accommodate failed operations and re-invocations of paused operations.

An obstruction-free implementation of any object is presented (Section 3.2), which exemplifies how pause and fail values are returned when a legal response is not possible. A natural way to evaluate obstruction-free implementations is by considering the *contention-free step complexity*, namely, the number of steps taken by a process running alone, until it returns a value. Our implementations have linear contention-free step complexity and use a linear number of read/write base objects. By reduction to the lower bound of Jayanti, Tan and Toueg [19], we show that obstruction-free implementations of many long-lived objects from historyless base objects must have $\Omega(n)$ contention-free step complexity and must use $\Omega(n)$ historyless objects.

In practice, the burden of providing termination of obstruction-free implementations is shifted to a system-supported *contention manager* that relies on low-level mechanisms such as timers, identifiers and interrupts [17,25]. The contention manager instructs the clients if and when to invoke operations, trying to ensure that only a single process eventually accesses the concurrent object. To explore inherent characteristics of obstruction-free implementations, we consider a specific contention manager that can turn any obstruction-free implementation into a nonblocking one (none of those of [14,17,25,26] can do so). The contention manager indicates the client whether to continue or not (a binary indication), and should eventually indicate only to a single client to continue [8]. [3]

We show (Section 3.5) that there are no *gracefully degrading* consensus implementations, which are nonblocking when the contention manager operates correctly, but remain (at least) obstruction-free when the contention manager is unsuccessful.

We finally explore *solo-fast* implementations. These are wait-free linearizable object implementations that use only read/write base objects when there is no step contention, but may fall back on more powerful objects like compare&swap, when contention occurs. Luchangco, Moir and Shavit [24] presented a generic object implementation that uses only reads and writes when an operation runs in

[3] This specification style is inspired by the way *failure detectors* [8, 9] abstract away (partial) synchrony assumptions. It highlights the intriguing connection between obstruction-free implementations and Paxos-style algorithms for consensus and state-machine replication [21].

the absence of contention. However, in their implementation this also means lack of pending operations, namely, lack of *point* contention; moreover, a transient increase in point contention will cause a subsequent operation (that has no point contention) to invoke costly C&S operations. In light of this, it is challenging to design truly solo-fast implementations that do not invoke C&S operations in the more common case of no *step* contention.

Surprisingly, we show in this paper that any object has a solo-fast implementation by describing a solo-fast consensus implementation, and employing it within Herlihy's universal construction [15] (Section 4). The implementation has linear contention-free step complexity. We conjecture that solo-fast ones are inherently slow: they must have (at least) non-constant step complexity.

2 Model

A system contains a set Π of $n > 1$ *processes* $p_1, \ldots, p_n$ that communicate through shared *objects*.

Every object has a *type* that is defined by a triple (O, R, Δ), where O is a set of *invocations*, R is a set of *responses*, and Δ is a set of sequences of invocation-response pairs. The set Δ, known as the *sequential specification* of the type, contains all the sequences of invocations and responses allowed by the object.

For example, the *compare&swap* (C&S) object is accessed by a $CS(r_1,\ r_2,\ m)$ operation; the operation compares that value in memory location m with the content of local variable r_1, and if equal, writes the value of r_2 to m. The operation returns the *old* value of m. The sequential specification of the *compare&swap* type includes all sequences of CS operations that obey this rule.

Another important example is the *consensus* object, on which processes perform a *propose* operation with an argument in some set V. The sequential specification of consensus includes all sequences of *propose* operations that return the argument of the first operation in every sequence.

To implement a (high-level) object from a collection of *base* objects, processes follow an *algorithm* A, which is a collection of state machines $A_1, \ldots A_n$, one for each process.

When receiving an *invocation* (to the high-level object), process p_i takes steps according to A_i. In each step, p_i can either (a) invoke an operation on a base object, or (b) receive the response of its previous base operation, or (c) perform some local computation. After each step, p_i changes its local state according to A_i, and possibly returns a response on the pending high-level operation.

We investigate implementations that work when process speeds are highly-variable, and at the extreme case, a process may stop taking steps.

An *execution* e of an algorithm A is a sequence of interleaved *events*. Every execution induces a *history* that includes only the invocations and responses of the high-level operations. Each invocation or response is associated with a single process and a single object. A *local history* of process p_j in H, $H|j$, is the subsequence of H containing only events of p_j. Similarly, $H|x$ is the subsequence of H of operations on an object x.

A response *matches* an invocation if they are associated with the same process and the same object. A local history is *well-formed* if it is a sequence of matching invocation-response pairs, except perhaps for the last invocation in a finite local history. A history H is *well-formed* if every local history in H is well-formed.

A matching invocation-response pair $[i, r]$ is called a *complete operation*, and we say that i *returns* r. An invocation i without a matching response is called a *pending operation*; a *completion* of a pending operation, that is, an invocation, is the invocation together with an appropriate response. The fragment of H (or e, its corresponding execution) between the invocation i and its matching response r (if it exists) is the operation's *interval.*

In an infinite execution, a process is *correct* if it takes an infinite number of steps or it has no pending operation; otherwise, it is *faulty.*

A history H is *sequential* if every invocation is immediately followed by its matching response. A sequential history H is *legal* if for every object x, $H|x$ is in the sequential specification of x.

Two different invocations i and i' on the same object x are *concurrent* in a history H, if i and i' are both pending in some finite prefix of H. This implies that their intervals overlap. We say that two operations $[i, r]$ and $[i', r']$ (or i' if i' is pending) are *non-concurrent* if their intervals are non-overlapping: Either r appears before i' in H, in which case we say that $[i, r]$ *precedes* $[i', r']$, or r' appears before i in H, in which case we say that $[i, r]$ *follows* $[i', r']$.

A well-formed history H satisfies *extended linearizability* [15] (see also [6, Chapter 10]) if there is a permutation H' containing all the complete operations and completions of a subset of the pending operations in H, such that (1) H' is legal, and (2) H' respects the order of non-concurrent operations in H.

This paper explores the benefits induced by the scenarios in which contention is rare. Formally, we define the *step contention* of a fragment in execution e to be the number of processes that take steps in this fragment. An operation is *step contention-free in* e if step contention of its interval in e is 1. An operation is *eventually step contention-free in* e if its interval in e has a suffix with step contention 1.

Alternative ways to measure contention during an operation's interval were previously defined [2, 5]: The *interval contention* during is the number of processes whose operations are concurrent with an operation *op* in e. The *point contention* of *op* is the maximum number of operations *simultaneously* concurrent with *op* in e. Note that interval contention is always equal to or higher than step contention, but point contention is incomparable to step contention. Note also when no operation overlaps *op*, then both the point contention and the interval contention are 1.

3 Obstruction-Free Implementations

This section considers obstruction-free implementations [16,17], which guarantee progress only in the absence of step contention.

3.1 Definitions

Originally [16, 17], an implementation is called obstruction-free *"if it guarantees progress for every thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, ..."* [17, Page 522]. For deterministic implementations, this requirement is equivalent to *solo termination* [11], and it echoes the liveness correctness conditions stated for Paxos-style algorithms for state-machine replication [21].

Using our terminology, an implementation is *obstruction-free* if every operation that is eventually step contention-free eventually returns.

Obstruction-freedom is a very weak liveness condition, and it requires the operation to return only under very restricted conditions. In all other circumstances, we only require that an operation's response is legal, *if it returns a response at all.*

If an operation cannot return a legal response, it is useful to return control to a higher-level entity, which we call the *client*. The client may consult a system-specific mechanism called a *contention manager*, in order to expedite termination.

There are two ways in which an obstruction-free implementation returns control to the client, depending on whether the implementation is certain that the operation did not have any effect on the object or not. If the implementation is certain that the operation did not have an effect, a special *fail* value is returned, indicating that the operation was not applied, and the client is free to invoke any operation it wishes. Otherwise, a special *pause* value $\perp$ is returned, and the client must re-invoke the same operation until a non-$\perp$ response is received. (We discuss the need for the two indications below.)

We add to R, the set of responses of an object, a special *pause* value $\perp \notin R$ and a special *fail* value $\emptyset \notin R$. The definition of a *well-formed* local history is extended to require that if an invocation i is followed by the response $\perp$, then the subsequent event, if exists, is i.

The definition of extended linearizability is further extended so that invocations returning *fail* are removed from the linearized history, while a sequence of invocations returning *pause* are considered as one pending operation.

Formally, let H be any history, and $\bar{H}$ be any well-formed local history of H. Let i be an invocation in $\bar{H}$ on an object x. A fragment of the form i, r in $\bar{H}$, where $r \in R$, is called an *occurrence of* i (returning r). Since an invocation occurrence might return $\perp$ and be re-invoked later, there might be a number of occurrences of i in a history. Consider the longest fragment of the form i or $i, \perp, i, \ldots, \perp, i$ in $\bar{H}$. If the fragment is followed by a matching response $r \notin \{\perp, \emptyset\}$, we call i, r or, resp., $i, \perp, i \ldots, \perp, i, r$ a *complete operation.* If the fragment is followed by a fail response $\emptyset$, we call $i, \emptyset$ or, resp., $i, \perp, i \ldots, \perp, i, \emptyset$ a *failed operation.* If the fragment is followed by no event or by $\perp$, we call i or $i, \perp$ or, resp., $i, \perp, i, \ldots, \perp, i$ or $i, \perp, i \ldots, \perp, i, \perp$ a *pending operation.* Since $\bar{H}$ is well-formed, a pending operation is a suffix of $\bar{H}$ ($\perp$ cannot be followed by an invocation other than i). The operation's interval is the shortest fragment of

Shared variables: *register* X, initially $\perp$, and *"only-fail" OF consensus object* C

```
Code for process $p_0$:
  upon $propose(v_0)$ do
    $d_0 \leftarrow C.propose(v_0)$
    if $d_0 = \emptyset$ then
      $d_0 \leftarrow X$
    return $d_0$

Code for process $p_1$:
  upon $propose(v_1)$ do
    $X \leftarrow v_1$
    repeat
      $d_1 \leftarrow C.propose(v_1)$
    until $d_1 \neq \emptyset$
    return $d_1$
```

Fig. 2. Wait-free consensus from "only-fail" obstruction-free consensus

H that includes all events of that operation. If the fragment $i, \perp, i, \ldots, \perp, i$ is followed by no event in $\bar{H}$ then the operation's *interval* is infinite.

As defined before, a well-formed history H is *linearizable* if there is a permutation H' containing all the complete operations in H and completions of a subset of the pending operations in H, such that H' is legal and it respects the order of non-concurrent operations in H. When taken in the context of the extended notions of complete and pending operations, this definition means that we order all non-failed operations, with the interval of a paused operation "spanned" across its re-invocations.

An implementation is *live* if every invocation occurrence return in a finite number of its own steps (although a value in $\{\perp, \emptyset\}$ can be returned). An implementation is *valid* if (1) an invocation occurrence returns $\perp$ (that is, *pause*) only when it is not step contention-free, and (2) an invocation occurrence i returns $\emptyset$ (that is, *fail*) only when the corresponding *operation* (the longest fragment of the form $i, \emptyset$ or $i, \perp, i, \ldots, \perp, i, \emptyset$ in the local history) is not step contention-free. It is immediate that any live and valid implementation is obstruction-free.

Ideally, an obstruction-free implementation should only be allowed to return valid responses and, in the case of step contention, *fail* indications to the client, enabling the client to either re-invoke the operation, or to invoke another operation. However, below we show that no obstruction-free consensus implementation from registers can enjoy this property. Thus, it is sometimes unavoidable to return $\perp$ in obstruction-free implementations.

Theorem 1. *There is no obstruction-free consensus implementation from registers that never returns* $\perp$.

Proof. By contradiction, consider an implementation of obstruction-free consensus that is allowed to return only $\emptyset$ in the case of step contention. Then it is possible to implement wait-free consensus for two processes p_0 and p_1 using one such consensus object, denoted C, and one register X, contradicting [10,23]. The algorithm is presented in Figure 2.

Validity of the algorithm follows from the fact that $\emptyset$ is returned only in case of step contention. If C returns $\emptyset$ at p_0, then p_1 can only decide its own value. Thus, Agreement is satisfied. Since p_1 eventually runs in the absence of contention, it eventually decides. Thus, Termination is also satisfied. $\square$

3.2 Obstruction-Free Generic Object Implementation

This section gives an algorithm that obstruction-free implements any object of type T, using only registers. Like previous universal implementations, it is built from consensus objects. (A simple obstruction-free consensus algorithm, derived from Paxos-style consensus algorithms, appears in the full version of the paper.)

The universal obstruction-free implementation relies on a sequential implementation of the object type T; it is live, valid and linearizable. Herlihy's universal nonblocking implementation [15] cannot be applied "off-the-shelf" since it does not handle re-invocations and failing. Instead, the algorithm builds on similar ideas, while making sure that *pause* or *fail* are returned only in the absence of step contention.

An object of type T is represented as a linked list; an element of the list represents an operation applied to the object. The list of operations clearly determines the list of corresponding responses. A process makes an invocation by appending a new element to the end of the list. The algorithm assumes a function *response*(*invs*, *inv*) that returns the response matching the invocation *inv* in a sequential execution of invocations from list *invs* (under the condition that $inv \in invs$).

The algorithm (Figure 3) uses the following shared variables:

– n atomic single-writer, multi-reader registers $L_1, \ldots, L_n$. Process p_i stores in L_i its last view of the object state in the form of a linked list of operations that p_i witnessed to be applied on the object.
– $C[\,]$ is an unbounded array of obstruction-free consensus objects. The array is used to agree on the order in which invocations are put into the linked list of operations.

Roughly, the algorithm works as follows. When a process p_i executes an invocation *inv*, it identifies the longest list L_j (let $k = |L_j|$). If *inv* is already in L_j, the response associated with *inv* in L_i is returned (line 5). This ensures that an operation takes effect at most once, even if repeated several times. If it is not the first instance of *inv*, and $k > |L_i|$ (i.e., *inv* was not decided in any OF Consensus to which it was proposed), p_i returns $\emptyset$ (line 9). Otherwise, p_i proposes *inv* to $C[k+1]$ (line 10). If $C[k+1]$ returns $\perp$ (step contention is detected), then p_i returns $\perp$ (line 13). If the propose operation fails, or returns a non-*inv* response while it is not the first instance of *inv*, then p_i returns $\emptyset$ (line 16). If $C[k+1]$ returns *inv*, then p_i returns the response associated with *inv* (line 20). Otherwise, the procedure is repeated, now at position $k+2$. Now if $C[k+2]$ returns a non-$\{inv, \perp\}$ response, then p_i returns $\emptyset$ (line 29). The second consensus operation ensures *validity* of the implementation, namely, that $\emptyset$ is never returned in line 29 if the corresponding operation is step contention-free.

This algorithm implies the next theorem (the correctness proof appears in the full version of the paper).

Theorem 2. *Every sequential type T has an obstruction-free linearizable implementation from registers.*

```
Shared variables:
    Register $L_1, \dots, L_n \leftarrow \emptyset, \dots, \emptyset$
    OF-Consensus $C[\,]$

upon Invoking $inv$ do
    $invs \leftarrow longest(\{L_1, \dots, L_n\})$          { Select the longest invocation list }
    if $inv \in invs$ then
        $check \leftarrow false$
        return $response(invs, inv)$          { Return if inv is already completed }
    $k \leftarrow |invs|$
    if $(k > |L_i|)$ and $check$ then
        $check \leftarrow false$
        return $\emptyset$          { Fail the operation }
    $dec \leftarrow C[k+1].propose(inv)$          { The 1st consensus operation }
    if $dec = \perp$ then
        $check \leftarrow true$
        return $\perp$
    if $(dec = \emptyset)$ or $(dec \neq inv$ and $check)$ then
        $check \leftarrow false$
        return $\emptyset$          { Fail the operation }
    $invs \leftarrow invs \cdot dec$; $L_i \leftarrow invs$          { Update $L_i$ }
    if $dec = inv$ then
        $check \leftarrow false$
        return $response(invs, inv)$          { Return if inv is decided }
    $dec \leftarrow C[k+2].propose(inv)$          { The 2nd consensus operation }
    if $dec = \perp$ then
        $check \leftarrow true$
        return $\perp$
    if $dec \neq \emptyset$ then $invs \leftarrow invs \cdot dec$; $L_i \leftarrow invs$
    if $dec = inv$ then
        $check \leftarrow false$
        return $response(invs, inv)$          { Return if inv is decided }
    return $\emptyset$          { Fail if inv is ignored twice }
```

Fig. 3. An obstruction-free implementation of T: code for process p_i

Remark. Our algorithm satisfies one additional property. In any execution, every operation takes effect (if it does) before it stops taking steps in that execution. In other words, the implementation stays linearizable even if we restrict an operation's interval to the shortest fragment of the execution which contains all steps of that operation. As a result, an operation invoked by a faulty process takes effect (if it does) before the process fails, which makes our implementations *strictly linearizable* [3].

A simpler proof of Theorem 2 can be obtained by presenting an algorithm that returns only $\perp$ indications in the case of step contention. However, our algorithm is better in the sense that it carefully detects the scenarios in which an applied operation did not take effect, and thus $\emptyset$ can be returned, which makes our algorithm more convenient to use.

3.3 Obstruction-Free Implementations are Slow

The universal construction presented in Figure 3 is not very efficient: finding the longest list of invocations requires to collect information from all processes. The next theorem shows that this is inherent in obstruction-free universal implementations from read/write base objects, by proving a lower bound of $\Omega(n)$ on the number of steps and on the number of registers for implementing a compare&swap object.

Theorem 3. *Let A be any obstruction-free implementation of n-valued compare&swap from registers, then A has an execution in which a step contention-free operation takes $n-1$ or more steps and accesses $n-1$ or more different objects.*

Proof. Follows directly from the result of Jayanti, Tan and Toueg [19]. They show that any implementation of n-valued compare&swap that satisfies the solo termination property has an execution in which a solo operation (i.e., an operation that does not observe step contention) takes $n-1$ or more steps and accesses at $n-1$ or more different objects. Since any obstruction-free implementation ensures the solo termination property, we immediately have the theorem. $\square$

3.4 Leveraging Obstruction-Free Objects

The next two subsections discuss how obstruction-free implementations can be turned into nonblocking or wait-free ones using a contention manager. The contention manager we consider provides the client with a binary indication whether to continue or not. The contention manager works well when it indicates only to a single client to continue. Formally, in response to the client's query, the contention manager returns either 0 or 1, telling the client to abort or to continue (respectively); in the latter case, we say that the client is a *leader*. The *eventual* contention manager, denoted Ω, guarantees that eventually exactly one correct client with a pending operation (if such a client exists) is a leader; it is deliberately similar to the "sloppy leader" failure detector and can be implemented using partial synchrony assumptions [8].

A single obstruction-free consensus object and Ω can implement *nonblocking* consensus using the following simple algorithm: A process queries the contention manager and, if it is a leader, the process makes a *propose* invocation on the underlying obstruction-free consensus object. If the response is neither $\perp$ nor $\emptyset$, it is returned; otherwise, the process repeats. This implies the following result:

Theorem 4. *Consensus has a nonblocking implementation from (only) obstruction-free consensus and Ω.*

3.5 Graceful Degradation of Obstruction-Free Implementations

Obstruction-free consensus can be implemented from registers only [4]. On the other hand, *wait-free* consensus can be implemented from registers using Ω [22]. However, these two liveness properties cannot be combined in the same implementation, namely, there is no wait-free consensus implementation using registers and Ω which becomes (at least) obstruction-free when the contention manager fails to eventually elect a single correct leader. In fact, we prove the claim even for *nonblocking* consensus implementations.

Theorem 5. *There is no nonblocking consensus implementation using registers and Ω that ensures obstruction-freedom when the contention manager fails to eventually elect a single correct leader.*

Proof. Suppose, by contradiction, that an algorithm A provides such an implementation. We show that it is then possible to devise an algorithm A' that implements nonblocking consensus for two processes, p_1 and p_2 with registers only — a contradiction with [10, 23].

In A', processes take steps like in A, except that, instead of using Ω, processes assume that Ω always indicates p_1 as the only leader. In doing so, processes cyclically invoke *propose* operations until a non-$\{\perp, \emptyset\}$ value is returned. Note that A' cannot violate safety properties of consensus, since every finite execution of A' is also an execution of A. To establish a contradiction, it is thus sufficient to show that at least one correct process eventually terminates in A', i.e., obtains a non-$\{\perp, \emptyset\}$ value from the underlying algorithm A.

Every execution of A' belongs to one of the following classes:
(1) Executions in which p_1 is correct, i.e., the assumed output of the contention manager complies with the specification of Ω. Such an execution is indistinguishable to p_1 and p_2 from executions of A in which processes $p_3, \ldots, p_n$ are initially faulty, and p_1 is the only correct leader. Since A implements a nonblocking consensus using Ω, some correct process (p_1 or p_2) eventually obtains a non-$\{\perp, \emptyset\}$ value from A and decides.
(2) Executions in which p_1 is faulty, i.e., the assumed output of the contention manager does not comply with the specification of Ω. Assume that p_2 is correct in such an execution (if both p_1 and p_2 are faulty, consensus is trivially solved). Any finite prefix of our execution is indistinguishable to p_2 from an execution of A in which processes $p_3, \ldots, p_n$ are initially faulty, and the contention manager malfunctions. Since p_2 is eventually running in the absence of step contention, and A ensures obstruction-freedom even when the contention manager is incorrect, p_2 eventually obtains a non-$\{\perp, \emptyset\}$ value from A and decides.

In other words, A' guarantees that whenever there is at least one correct process, some correct process eventually decides — a contradiction. □

4 Solo-Fast Implementations

We say that a wait-free linearizable implementation of a sequential type T from registers and other objects (e.g., compare&swap) is *solo-fast* if only read and write operations are invoked by any step contention-free operation on it.

4.1 Solo-Fast Generic Object Implementation

Figure 4 presents a solo-fast consensus implementation. The algorithm proceeds in rounds (lines 13–25). Starting the algorithm, every process first computes in line 3 the smallest round k in which a value can be *fixed*, i.e., returned in line 19 (we say that p_i *joins* in round k). The algorithm guarantees that if any process

```
Shared variables:
    Registers $\{A_j\}, \{B_j\}, j \in \{1, 2, \ldots, n\}$, initially $\perp$
    C&S $C_1, \ldots C_{n-1}$, initially $\perp$

upon $propose(input_i)$ do
    $V \leftarrow$ collect $A$                                    { $\perp$'s are ignored in each collect }
    $k_i \leftarrow \min\{k \geq 1 \mid \forall (k', v') \in V : k' \leq k \wedge \forall (k, v'), (k, v'') \in V : v' = v''\}$
    if $\exists (k, v) \in V$ then
        $v_i \leftarrow v$
    else
        $V' \leftarrow$ collect $B$
        if $V' \neq \emptyset$ then
            $v_i \leftarrow$ the highest timestamped value in $V'$
        else
            $v_i \leftarrow input_i$
    while (true) do
        $A_i \leftarrow (k_i, v_i)$
        $V \leftarrow$ collect $A$
        if $\forall (k', v') \in V : k' < k_i \vee (k' = k_i \wedge v' = v_i)$ then
            $B_i \leftarrow (k_i, v_i)$
            $V \leftarrow$ collect $A$
            if $\forall (k', v') \in V : k' < k_i \vee (k' = k_i \wedge v' = v_i)$ then
                return $v_i$
        $V' \leftarrow$ collect $B$
        if $V' \neq \emptyset$ then
            $v_i \leftarrow$ the highest timestamped value in $V'$
        $v' \leftarrow C_{k_i}.CS(\perp, v_i)$
        if $v' \neq \perp$ then $v_i \leftarrow v'$
        $k_i \leftarrow k_i + 1$
```

Fig. 4. An n-process solo-fast consensus: code for process p_i

fixes a value, then no process can ever fix a different value. In every round, starting from round k, p_i tries to fix its current estimate. It is ensured that if no other process tries to fix concurrently a different value in the current or higher round, then the estimate must be fixed. If p_i is not able to fix the estimate in the current round (we say that p_i *aborts* in that round), which can only happen when there is step contention, it updates the estimate using a C&S operation and goes to the next round. The algorithm guarantees that whenever process p_i aborts in round k, and no process joins in round $k+1$, then p_i fixes its estimate in round $k+1$ (C&S ensures that no two processes that abort in round k try to fix different values in round $k+1$). We show that no process can join in round n or later, and thus p_i fixes its estimate in round $k \leq n$. The algorithm is solo-fast, since no process can abort in a round (and thus use a C&S operation) in the absence of step contention.

This algorithm implies the next theorem (the correctness proof appears in the full version of the paper).

Theorem 6. *There is a solo-fast consensus implementation from registers and C&S objects, that takes $O(n)$ steps in the solo path.*

From Theorem 6 and Herlihy's universal construction [15], we immediately obtain:

Corollary 1. *Every sequential type T has a solo-fast implementation from registers and C&S objects.*

4.2 Lower Bounds and Reductions

Our implementation has linear space and contention-free step complexity. Proving that it is asymptotically optimal is not straightforward. Unlike obstruction-free implementations (see the proof of Theorem 3), the lower bound of [19] cannot be applied to solo-fast implementations, since processes *can* access non-historyless objects such as C&S, which is not allowed in [19]. Nevertheless, if we assume that the C&S objects are *non-readable*, then a simple variation of [19] implies that the step and space complexity for solo-fast is at least linear. (This matches the complexity of our implementation.)

The apparent similarity between obstruction-free and solo-fast implementations tempts to think that a linear lower bound on space and step complexity of a solo-fast implementation can be obtained by a reduction from an obstruction-free one (applying Theorem 3). However, despite of the similarity, such a reduction seems difficult to achieve. For instance, a seemingly straightforward transformation from a solo-fast implementation to an obstruction-free one in which hardware C&S objects are recursively substituted with their solo-fast implementations does not guarantee solo termination if the underlying C&S objects are readable. Indeed, even on a solo path, any read operation on a C&S object would recursively call a solo-fast version of it, and so on.

5 Discussion

This paper studies the notion of step contention, which inherently does not charge for processes stalled, e.g., due to failures or swap-outs, and is, in this sense, fundamentally different from point or interval contention. We show that registers are powerful enough to ensure liveness in the absence of step contention (which leads to a wider set of executions than when looking at other forms of contention). However, we suggest that building such implementations using only registers in the absence of step contention is inherently expensive and of limited benefit.

There are several interesting avenues for further research:

Complexity of obstruction-free consensus. We have shown tight bounds on the cost of *generic* obstruction-free implementations. However, there might be more efficient obstruction-free solutions for specific problems. For obstruction-free consensus, for example, an $\Omega(\sqrt{n})$ lower bound on the number of registers (or historyless objects) can be derived from the lower bound of Fich, Herlihy and Shavit [11]. This bound is not tight (the upper bound is $O(n)$) and moreover, it does not bound the *contention-free* step complexity of obstruction-free consensus.

Complexity of solo-fast implementations. Our solo-fast implementation performs $O(n)$ read and write steps, even in the absence of step contention; by employing adaptive collect [1, 5], the step complexity can be made to depend only on the point contention; by employing adaptive collect for unbounded concurrency [13], it can be made independent of the number of processes.

We conjecture that a non-constant lower bound on the contention-free step complexity of any generic solo-fast implementation holds even if underlying compare&swap objects are readable, making solo-fast implementations rather inefficient. On the other hand, it is possible that the step and space complexities of solo-fast consensus can be made constant if objects *slightly* more powerful than read/write registers, e.g., counters or queues, are used on a solo path.

Contention management. The contention manager we considered is fundamentally different from those considered in [14, 17, 25, 26]. It is easy to see that none of those can transform any obstruction-free implementation into a nonblocking one. Those contention managers do not provide any worst case nonblocking deterministic guarantees (with the exception of [14] in the absence of failures), and were actually rather designed to provide a high throughput in the average case. Devising a contention manager that would provide deterministic worst case guarantees with acceptable average case throughput is an interesting research direction.

Acknowledgments: We would like to thank Partha Dutta, Danny Hendler, Ron Levy, and Eric Ruppert for important discussions on the topic of this paper, and the anonymous reviewers for helpful comments.

References

1. Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 262–272, 1999.
2. Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.
3. M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical report, HP Laboratories Palo Alto, 2003.
4. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, 1990.
5. H. Attiya and A. Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, 2003.
6. H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.
7. 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.
8. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
9. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
10. 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.

11. F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.
12. 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.
13. E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 161–169, 2001.
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 (PODC)*, 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, 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.
17. 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.
18. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.
19. P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.
20. 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.
21. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
22. W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, volume 857 of *LNCS*, pages 280–295. Springer Verlag, 1994.
23. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.
24. V. Luchango, 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.
25. 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.
26. M. L. Scott and W. N. Scherer III. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.