On the Inherent Sequentiality of Concurrent Objects

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1137/08072646x">http://dx.doi.org/10.1137/08072646x</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Society for Industrial and Applied Mathematics</td>
</tr>
<tr>
<td>Version</td>
<td>Final published version</td>
</tr>
<tr>
<td>Accessed</td>
<td>Wed Jan 02 21:45:47 EST 2019</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/75034">http://hdl.handle.net/1721.1/75034</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Article is made available in accordance with the publisher’s policy and may be subject to US copyright law. Please refer to the publisher’s site for terms of use.</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td></td>
</tr>
</tbody>
</table>
ON THE INHERENT SEQUENTIALITY OF CONCURRENT OBJECTS

FAITH ELLEN†, DANNY HENDLER‡, AND NIR SHAVIT§

Abstract. We present \( \Omega(n) \) lower bounds on the worst case time to perform a single instance of an operation in any nonblocking implementation of a large class of concurrent data structures shared by \( n \) processes. Time is measured by the number of stalls a process incurs as a result of contention with other processes. For standard data structures such as counters, stacks, and queues, our bounds are tight. The implementations considered may apply any primitives to a base object. No upper bounds are assumed on either the number of base objects or their size.

Key words. distributed data structures, lower bounds, covering, memory contention

AMS subject classifications. 68Q25, 68W15

DOI. 10.1137/08072646X

1. Introduction. The effective parallelization of shared data structures is a key element in the design of scalable applications for coming generations of multicore processors. A key question that arises is whether some widely used sequential data structures, such as counters, stacks, and queues, can be effectively parallelized. According to folklore, they are inherently sequential. Informally, this means that, in any parallel implementations of these data structures, the measurable (“wall clock”) time it takes to perform \( n \) operations, each by a different process, is no better than that of sequential implementations, that is, \( n \) times the cost of a single operation. This paper presents tight time complexity bounds that formally prove the inherently sequential nature of many shared data structures.

To formally model inherently sequential data structures, we address their consistency, progress, and time complexity. The standard consistency condition for shared data structures is linearizability [21]. It captures the notion that operations must appear to be instantaneous, even though they may overlap in time. The progress condition we consider is obstruction-freedom [18]. Lower bounds for it imply the same lower bounds for stronger nonblocking progress conditions such as wait-freedom and lock-freedom [19].

In a realistic multiprocessor setting, the time for performing data structure operations is dominated by the cost of accessing memory. Moreover, as suggested in the pioneering work of Dwork, Herlihy, and Waarts [10] and Gibbons, Matias, and Ramachandran [14, 15], realistic memory access costs must take into account not only the time to perform an operation on an object, but also the delays resulting from waiting for other processes that access the object at the same time. These delays, called
stalls, are an inherent part of multiprocessor behavior [7] and affect the measurable time of operations. To capture this real-world behavior, researchers such as Merritt and Taubenfeld [25], Cypher [9], and Anderson and Kim [2] use a worst case time complexity measure that counts both the number of accesses to shared objects and the number of stalls incurred.

As an example, consider the question of providing a nonblocking linearizable implementation of a shared counter. If the hardware supports a fetch&increment operation, then the simplest way of implementing a counter shared by \( n \) processes is to have all processes share a single object on which each performs fetch&increment operations. Unfortunately, this centralized implementation has a serious drawback: In the situation where all processes attempt to access the counter simultaneously, one process will have to wait for the other \( n - 1 \) processes to complete their fetch&increment operations. According to our measure, it will execute one fetch&increment and incur \( n - 1 \) stalls. One can think of a stall as taking the same time as it takes another process to complete its fetch&increment. In this case, the measurable time the \( n \) concurrent operations take from start to finish is at least the time it would take to execute the operations sequentially, one after the other. The literature does contain other, highly decentralized, nonblocking coordination structures, such as counting networks [4, 20], that use multiple objects supporting read-modify-write operations to implement shared counters. However, all such structures provide counters that either have linear worst case time complexity or are not linearizable [20, 26, 27]. The results we present in this paper show that this is not a coincidence: A counter is indeed an inherently sequential data structure. There is no decentralized implementation that has better worst case time complexity than the centralized solution.

The counter is just one example. Although nonblocking data structures are widely available and have been deployed in real-world software packages [23], in many cases, we still lack a basic understanding of the limitations to achieving high scalability in their design. For many of the standard concurrent data structures, including counters, queues, and stacks, the best nonblocking linearizable implementations known, using any read-modify-write primitives, have \( \Theta(n) \) time complexity. The best previous lower bounds on the time complexity of implementing these data structures from arbitrary read-modify-write primitives were \( \Omega(\sqrt{n}) \) [16]. Thus, it was open whether the linear upper bounds were inherent.

This paper provides linear lower bounds on the worst case number of stalls per operation in linearizable obstruction-free concurrent implementations of a large class of objects, including common data structures such as counters, queues, and stacks, from arbitrary read-modify-write primitives. Note that any operation on a single shared object can be expressed as a read-modify-write primitive.

We use a new variant of a covering argument [8, 13] to prove linear lower bounds for implementations of objects in a new class, \( \mathcal{G} \), that includes shared counters [20] and single-writer snapshots [1, 3, 6]. Covering arguments bring processes to states in which they are poised to overwrite certain shared objects, causing a loss of information, which leads to incorrect behavior. Unlike previous covering arguments, ours does not hide information. Rather, processes are brought to states where they will access objects concurrently with other processes, thus incurring memory stalls. We build an execution in which, in the course of performing a single high-level operation, we cause a process to incur a sequence of \( n - 1 \) stalls, one with every other process in the system. It does not matter for the proof whether these stalls are on the same or different objects. The implication is that, if measured by a wall clock, the time...
it would take for all the operations to complete is at least linear in the number of processes.

This lower bound proof does not apply to implementations of objects such as queues and stacks. However, we are able to prove a similar result for such implementations by way of a reduction. For example, if we initialize a queue with sufficiently many consecutive integers, we obtain an implementation of a counter that can support a bounded number of fetch\&increment operations, each simulated by a single dequeue operation. Under the assumption that each instance of the dequeue operation accesses a bounded number of distinct base objects, we construct an execution of bounded length, in which \( n - 1 \) stalls are incurred by a process performing a single instance of fetch\&increment.

The rest of this paper is organized as follows. Section 2 begins with a survey of recent related work. Section 3 describes the model we consider. We define the class of objects \( G \) and prove the lower bound for this class in sections 4 and 5, respectively. In section 6, we present the reduction that extends our linear time lower bound to stacks and queues.

2. Related work. There is an extensive body of work on lower bounds in shared memory computation. The interested reader can find a survey in [13]. For the sake of brevity, we focus on recent work aimed at deriving lower bounds for implementing common data structures on real machines.

Jayanti, Tan, and Toueg [22] also use a covering argument to prove linear time and space lower bounds for implementations of perturbable objects from historyless primitives [12] and resettable consensus. Their time lower bounds are different than ours for a number of reasons: They count the number of shared memory accesses, not stalls. Unlike the class \( G \), defined in section 4, the class of perturbable objects is defined in terms of low-level executions. In our opinion, this makes the definition of perturbable somewhat more difficult to understand and use. Finally, the set of historyless primitives they consider is a restricted subset of the class of all read-modify-write primitives, which we consider. Many real-world primitives, such as fetch\&increment, load-linked, store-conditional, and compare\&swap are not historyless. In fact, objects that support only historyless primitives have consensus number at most 2 [17].

Dwork, Herlihy, and Waarts [10] give the first formal complexity model for contention in shared memory multiprocessors. They introduce the notion of stalls in order to capture the delays incurred by processes while waiting to access shared locations. The idea was that stalls are reflected in the observed execution time for processes. They derive lower bounds on the number of stalls incurred by wait-free implementations of counting networks and consensus objects.

Gibbons, Matias, and Ramachandran introduce the queue-read queue-write (QRQW) asynchronous PRAM model [14]. Their model allows concurrent reads and writes to shared memory locations, each of which is viewed as having a queue which can service a single request at a time.

Hendler and Shavit prove \( \Omega(\sqrt{n}) \) time lower bounds on a class of objects called Influence(\( n \)) that includes objects such as linearizable queues, stacks, counters, and hash tables [16]. They prove that any lock-free implementation of such objects has an execution in which some operation either accesses \( \Omega(\sqrt{n}) \) distinct objects or incurs \( \Omega(\sqrt{n}) \) memory stalls.

3. Model. We consider a standard model of an asynchronous shared memory system [17, 24], in which processes communicate by applying operations to shared objects.
An object is an instance of an abstract data type. It is specified by a set of possible values, an initial value (which may differ according to the algorithm), and a set of operations that provide the only means of manipulating the object. The application of an operation by a process to a shared object can change the value of the object. It also returns a response to the process that can change its state. The resulting value of the object and the response can depend on the value of the object prior to the application of the operation and which process applies it.

An implementation of an object that is shared by a set of n processes provides a representation for the object from a set of shared base objects, each of which is assigned an initial value, and algorithms for each process to perform each operation on the object being implemented. To avoid confusion, we say that an operation is performed on an implemented object and a primitive is applied to a base object. Since we are considering lower bounds on worst case complexity, we may assume, without loss of generality, that all primitives are deterministic: Allowing an adversary to choose how primitives behave can’t make the proofs harder.

We consider RMW base objects, which support a set of read-modify-write primitives. An read-modify-write primitive applied by a process to a base object atomically updates the value of the object with a new value, which is a function g(v, w) of the old value v and any input parameters w, and returns a response h(v, w) to the process.

Fetch&add is an example of an read-modify-write primitive. Its update function is \( g(v, w) = v + w \), and its response value is \( v \), the previous value of the base object. Fetch&increment is a special case of fetch&add where \( w \) always equals 1. Read is also an read-modify-write primitive. It takes no input, its update function is \( g(v) = v \), and its response function is \( h(v) = v \). Write is another example of an read-modify-write primitive. Its update function is \( g(v, w) = w \), and its response function is \( h(v, w) = \text{ack} \). An read-modify-write primitive is trivial if \( g(v, w) = v \), so it never changes the value of the base object to which it is applied; otherwise, it is nontrivial. Read is an example of a trivial primitive; write and Fetch&increment are examples of nontrivial primitives.

A historyless primitive either never changes the value of the object (i.e., it is trivial) or always changes it to a new value that does not depend on its current value (i.e., \( g(v, w) \) is not a function of \( v \)). Read and write are examples of historyless primitives. Fetch&add is an example of a primitive that is not historyless. An object is historyless if it supports only historyless primitives.

An event consists of a process, a base object, and a primitive, together with values for its input parameters, which is applied by the process to the base object. We say that the process applies the event and that the process or the event accesses the base object. An event whose primitive is nontrivial is called a nontrivial event.

Suppose a process \( p \) wants to perform an operation on an implemented object \( O \). The implementation of \( O \) provides an algorithm for performing this operation, which \( p \) executes. While executing this algorithm, \( p \) does local computation and applies primitives to base objects. Which events are applied by \( p \) while it is performing an operation on an implemented object may depend on input parameters to the operation, responses it received from events that it applied previously, and, hence, indirectly, on events that other processes applied.

A configuration describes the value of each object and the state of each process. An execution is a (finite or infinite) sequence of events in which each process applies events starting from an initial configuration and changes state according to its algorithm, based on the responses it receives. Any prefix of an execution is an execution.
If \( EE' \) is an execution, then the sequence of events \( E' \) is called an extension of \( E \). The value of a base object \( r \) in the configuration that results from applying all the events in a finite execution \( E \) is called \( r \)'s value immediately after \( E \). If no event in \( E \) changes the value of \( r \), then \( r \)'s value immediately after \( E \) is the initial value of \( r \).

An operation instance is an operation (together with values for its input parameters) by a specified process on the implemented object. In an execution, each process performs a sequence of operation instances. A process can perform only one operation instance at a time. The events of an operation instance applied by some process can be interleaved with events applied by other processes. If the last event of an operation instance \( \Phi \) has been applied in an execution \( E \), we say that \( \Phi \) completes in \( E \). In this case, we call the value returned by \( \Phi \) in \( E \) the response of \( \Phi \) in \( E \). We say that a process \( p \) is active immediately after a finite execution \( E \) if, in \( E \), \( p \) has applied at least one event of some operation instance \( \Phi \) and \( \Phi \) is not complete in \( E \). We say that a process is idle immediately after \( E \) if, during \( E \), it completes every operation instance that it starts. If \( p \) is not active, then we can treat it as if it is idle, since any local steps it has performed since last completing an operation instance do not affect other processes.

In an initial configuration, each base object has its initial value, and all processes are idle. If a process is active in the configuration resulting from a finite execution, then it has exactly one enabled event. This is the next event the process will apply, according to the algorithm it is using to apply its current operation instance to the implemented object. If a process is idle, the first event of any operation instance that can be performed by that process is enabled. If a process \( p \) has an enabled event \( e \) immediately after execution \( E \), we say that \( p \) is poised to apply \( e \) immediately after \( E \).

If the sequence of events applied by a process is the same in two executions and each of these events returns the same response in both executions, then the executions are indistinguishable to the process. If two finite executions are indistinguishable to a set \( P \) of processes and the value of each base object in a set \( S \) is the same at the end of both executions, then any sequence of events by processes in \( P \) that only apply primitives to objects in \( S \) is an extension of either both of these executions or neither of them. In the first case, the resulting executions are indistinguishable to all processes in \( P \).

A linearization of an execution [21] is an ordering of all complete operation instances and a subset of the incomplete operation instances in the execution such that, if \( \Phi \) completes in the execution before \( \Phi' \) begins, then \( \Phi \) appears before \( \Phi' \) in the ordering. Furthermore, the results of each of these operation instances in the execution are the same as the results of the corresponding operation instance in the sequential execution in which the operation instances are performed in that order and which is consistent with the specification of the implemented object. An implementation is linearizable if all its executions are linearizable. Throughout this paper, we assume that all implementations are linearizable unless otherwise noted.

A sequence of events \( E \) is \( p \)-free if process \( p \) applies no events in \( E \). In a solo sequence of events, all events are by the same process. An implementation satisfies solo termination [12] if, for each finite execution and each process active immediately after that execution, there is a finite solo extension in which the process completes its operation instance. An implementation is obstruction-free [18] if it satisfies solo termination.

In shared memory systems, when multiple processes attempt to apply nontrivial events to the same object simultaneously, the events are serialized and operation instances incur stalls. Stalls capture the real-world behavior of a multiprocessor ma-
machine’s memory and interconnection medium, which handle multiple accesses to a single shared memory location sequentially.

**Definition 3.1.** Let \( e \) be an event applied by a process \( p \) as it performs an operation instance \( \Phi \) in an execution \( E = E_0 e_1 \cdots e_k e E_1 \), where \( e_1 \cdots e_k \) is the maximal consecutive sequence of events immediately preceding \( e \) that apply nontrivial primitives to the same base object accessed by \( e \) and that are applied by distinct processes different than \( p \). Then \( e \) incurs \( k \) memory stalls in \( E \). The number of stalls incurred by \( \Phi \) in \( E \) is the sum of the number of memory stalls that \( e \) incurs in \( E \), over all events \( e \) of \( \Phi \) in \( E \).

On a real machine, the stalls incurred by a process and the events the process applies are reflected in the observed “wall clock” execution time for the process. For many of the objects we consider, each operation can be implemented using a single read-modify-write event. In this case, when \( k \) operations are performed concurrently, the worst case time complexity of one operation is proportional to the time to complete all \( k \) operations in a sequential implementation. This is because there is an execution in which the event associated with one of these operations incurs \( k \) memory stalls.

The interested reader should note that Definition 3.1 is slightly stricter than the original definition by Dwork, Herlihy, and Waarts [10]. (Thus, our lower bounds also apply to their definition.) Their definition also includes stalls caused by events applying trivial primitives such as read. Our definition is also stricter than that of the asynchronous QRQW model of Gibbons, Matias, and Ramachandran [14], which only allows read, write, and test\&set primitives and counts stalls due to all of these.

**4. The class \( \mathcal{G} \).** In this section, we define a general class \( \mathcal{G} \) of objects to which our lower bound applies. It is closely related to the class of perturbable objects [22].

Whether an object belongs to the class \( \mathcal{G} \) depends only on its sequential specification. Roughly, any object in this class has an initial value, a process \( p \), and an operation such that, for sequences of operation instances in which \( p \) performs one instance \( \Phi \) of this operation (and no instances of any other operations), it is possible to change \( \Phi \)’s response by having another process \( q \) perform additional operation instances prior to \( \Phi \).

**Definition 4.1.** An object \( \mathcal{O} \) shared by \( n \) processes is in the class \( \mathcal{G} \) if it has an initial value, a process \( p \), an instance \( \Phi \) of an operation on \( \mathcal{O} \) by \( p \), and an infinite sequence \( \Upsilon_q \) of operation instances on \( \mathcal{O} \) by each process \( q \neq p \) such that, for every process \( q \neq p \) and for every interleaving \( AA' \) of finite prefixes of \( \Upsilon_q \), one for each \( q \neq p \), where

- each operation instance in \( A' \) is by a different process, and
- \( A' \) is \( q' \)-free,

there is a finite prefix \( QQ' \) of \( \Upsilon_q \) such that \( Q \) is the sequence of operation instances performed by \( q \) in \( A \) and, for every interleaving \( HH' \) of \( QQ' \) and the sequences of operation instances performed by each of the other processes in \( AA' \), where

- each operation instance in \( H' \) is by a different process,
- \( H' \) is \( q' \)-free, and
- for all \( q \neq p, q', \) all or all but one of the operation instances by process \( p \) precede \( Q' \) in \( HH' \),

the responses of \( \Phi \) are different when \( A\Phi \) and \( H\Phi \) are each performed on \( \mathcal{O} \) starting with its initial value.

Informally, if the operation instances in \( Q' \) are performed before \( \Phi \) and after the other operation instances in \( AA' \), except for possibly the last operation instance by each process \( q \neq q' \), then \( \Phi \) is guaranteed to have a different response. The reason for
allowing different interleavings is to capture the possibility of different linearizations in different executions.

The following result is a simple consequence of Definition 4.1.

**Proposition 4.2.** If an object \( O \in G \) can be implemented from an object \( O' \) so that each instance of an operation on \( O \) involves one instance of an operation on \( O' \), then \( O' \in G \).

In particular, if every operation supported by an object \( O \in G \) is also supported by object \( O' \), then \( O' \in G \).

Many common objects are in \( G \). Furthermore, determining whether an object is in \( G \) is relatively easy. We present a few examples of such proofs. They are similar to, but simpler than, analogous proofs in [22].

A **modulo-m counter** is an object whose set of values is the set \( \{0, 1, \ldots, m - 1\} \) for some \( m > 1 \). It supports a single parameterless operation, \( \text{fetch}\&\text{increment modulo } m \). The \( \text{fetch}\&\text{increment modulo } m \) operation atomically increments the value of the object to which it is applied and returns the previous value of the object, unless the object has value \( m - 1 \), in which case, it sets the value of the object to 0 (and returns \( m - 1 \)).

**Proposition 4.3.** A modulo-m counter object shared by \( n \leq m \) processes is in \( G \).

**Proof.** Consider a modulo-m counter with initial value 0 and any two processes \( p \) and \( q' \). The only operation supported by a modulo-m counter is \( \text{fetch}\&\text{increment modulo } m \).

Let \( AA' \) be any finite \( p \)-free sequence of instances of \( \text{fetch}\&\text{increment modulo } m \) performed on this object such that each operation instance in \( A' \) is by a different process and \( A' \) is \( q' \)-free. Let \( a \) and \( a' \) denote the number of operation instances in \( A \) and \( A' \), respectively. Then \( a \mod m \) is the response of \( \Phi \) in \( A\Phi \) and \( a' \leq n - 2 \).

Let \( Q \) denote the (possibly empty) sequence of operation instances performed by \( q' \) in \( A \). Let \( Q' \) be a sequence of \( b = n - a' - 1 \) instances of \( \text{fetch}\&\text{increment modulo } m \) by process \( q' \). Consider any interleaving \( HH' \) of \( QQ' \) and the sequences of operation instances performed by each of the other processes in \( AA' \). Suppose that each operation instance in \( H' \) is by a different process and \( H' \) is \( q' \)-free. Then \( H' \) contains at most \( n - 2 \) operation instances and \( H \) contains between \( a + a' + b - (n - 2) = a + 1 \) and \( a + a' + b = a + n - 1 \) operation instances. Thus the response of \( \Phi \) in \( H\Phi \) must be one of the values \( (a + 1) \mod m, (a + 2) \mod m, \ldots, (a + n - 1) \mod m \). Since \( n \leq m \), none of these values are equal to \( a \mod m \).

A **counter** is an object whose set of values is the integers. It supports a single parameterless operation, \( \text{fetch}\&\text{increment} \), that atomically increments the value of the object to which it is applied and returns the previous value of the object. A modulo-\( m \) counter can be implemented from a counter: To perform \( \text{fetch}\&\text{increment modulo } m \), it suffices to perform \( \text{fetch}\&\text{increment} \) and take the remainder when the value returned is divided by \( m \). Thus, it follows from Propositions 4.2 and 4.3 that a counter shared by any number of processes is in \( G \). \( \text{Fetch}\&\text{add} \) takes one integer parameter and adds it to the object to which it is applied. Since it is a generalization of \( \text{fetch}\&\text{increment} \), \( \text{fetch}\&\text{add} \) objects are also in \( G \).

The value of a **single-writer binary snapshot** object is a binary vector of components, one for each process. It supports two operations: \( \text{scan} \) and \( \text{update} \). For each \( v \in \{0, 1\} \), the operation instance \( \text{update}(v) \) by process \( p \) sets the value of the component for \( p \) to \( v \). A \( \text{scan} \) operation instance returns a vector consisting of the values of the \( n \) components.
Proposition 4.4. A single-writer binary snapshot object is in G.

Proof. Consider a single-writer binary snapshot object with initial value 0 in every component. Let Φ be an instance of a scan operation by some process p, and, for each q ≠ p, let Υ_q be an alternating sequence of instances of update(1) and update(0) operations by process q. Let AA' be any interleaving of finite prefixes of Υ_q, one for each q ≠ p, such that each operation instance in AA' is by a different process and AA' is q'-free for some process q' ≠ p. Let Q be the (possibly empty) sequence of operation instances performed by q' in A. Let QQ' be the prefix of Υ_q that contains one more operation instance than Q.

Consider any interleaving HH' of QQ' and the sequences of operation instances performed by each of the other processes in AA' such that H' is q'-free. Then the responses of Φ in AΦ and HΦ differ in the component for q'.

An m-valued compare&swap object, for any positive integer m, has the set of values \{0, 1, \ldots, m - 1\}. It supports the operations read and compare&swap(u, v) for all u, v ∈ \{0, 1, \ldots, m - 1\}. If the value of the object is u, the compare&swap(u, v) operation atomically changes the value of the object to v and returns true; otherwise the object’s value is not changed and the operation returns false.

Proposition 4.5. An m-valued compare&swap object shared by v ≤ m processes is in G.

Proof. Consider an m-valued compare&swap object with initial value 0. Let Φ be an instance of a read operation by some process p. For each j ∈ \{0, \ldots, m - 1\}, let α_j denote the sequence of operation instances compare&swap(1, j), compare&swap(2, j), \ldots, compare&swap(m - 1, j), and, for each q ≠ p, let Υ_q = (α_0 \cdots α_{m-1})^*.

Let AA' be any interleaving of finite prefixes of Υ_q, one for each q ≠ p, such that each operation instance in AA' is by a different process and AA' is q'-free for some process q' ≠ p. Let Q be the (possibly empty) sequence of operation instances performed by q' in A. Let u be the value returned by Φ in AΦ.

Let v ∈ \{0, \ldots, m - 1\} - \{u\} be different from the first argument of the last instance of compare&swap performed by process q in AA' for all q ≠ p, q' such that AA' is not q-free. The existence of v follows from the assumption that m ≥ n. Let QQ' be any finite prefix of Υ_q' that contains one more operation instance than Q. Consider any interleaving HH' of QQ' and the sequences of operation instances performed by each of the other processes in AA', where each operation instance in HH' is by a different process, H' is q'-free, and for all q ≠ p, q', all or all but one of the operation instances by process q precede Qq' in HH'. Then H = H''α_vα_v', where each instance of compare&swap in α'' has either its first component different than v or its second component equal to v. Regardless of what value the m-valued compare&swap object has immediately after H'', its value immediately after H''α_v is v and it remains v after every remaining operation instance in H. Then the response of Φ in HΦ is v ≠ u.

A binary LL/SC object has values (b, S), where b ∈ \{0, 1\} and S is any subset of the processes. It supports two operations: load-linked and store-conditional. Suppose a binary LL/SC object has value (b, S). If process p performs load-linked, then p is added to the set S, if it is not already in S, and the value b is returned. Now consider the result when process p performs store-conditional(b'), for b' ∈ \{0, 1\}. If p ∈ S, then true is returned and the new value of the object is (b', φ). If p ∉ S, then false is returned and the value of the object does not change.

Proposition 4.6. A binary LL/SC object is in G.
Proof. Consider a binary LL/SC object with initial value \((0, \phi)\). Let \(\Phi\) be an instance of a load-linked operation by some process \(p\). For each process \(q \neq p\), let
\[
\Upsilon_q = ((\text{load-linked, store-conditional}(1))^{n-1} (\text{load-linked, store-conditional}(0))^{n-1})^*.
\]
Let \(AA'\) be any interleaving of finite prefixes of \(\Upsilon_q\), one for each \(q \neq p\), such that each operation instance in \(A'\) is by a different process and \(A'\) is \(q'\)-free for some process \(q' \neq p\). Let \(Q\) be the (possibly empty) sequence of operation instances performed by \(q'\) in \(A\). Let \(u\) be the value returned by \(\Phi\) in \(A\). Let \(QQ'\) be any finite prefix of \(\Upsilon_q\) such that \(Q'\) ends in \((\text{load-linked, store-conditional}(1 - u))^{n-1}\).

Consider any interleaving \(HH'\) of \(QQ'\) and the sequences of operation instances performed by each of the other processes in \(AA'\), where each operation instance in \(H'\) is by a different process, \(H'\) is \(q'\)-free, and for all \(q \neq p, q'\), all or all but one of the operation instances by process \(q\) precede \(Q'\) in \(HH'\). Then \(H = H''\alpha\alpha''\), where \(\alpha = \text{load-linked, store-conditional}(1 - u)\) by process \(q'\) and \(\alpha''\) consists of instances of load-linked, instances of store-conditional\((1 - u)\) by process \(q'\), and instances of store-conditional by processes \(q \neq p, q'\) that do not perform load-linked in \(\alpha''\).

Regardless of what value the binary LL/SC object has immediately after \(H''\), its value immediately after \(H''\alpha\) is \((1 - u, \phi)\), and the value of its first component remains \(1 - u\) after every remaining operation instance in \(H\). Thus the response of \(\Phi\) in \(H\Phi\) is \(1 - u \neq u\).

Next, we prove that two other common objects are not in \(G\). A stack is an object whose value is any finite list of elements from some domain \(V\). It supports two operations. For \(v \in V\), \(\text{push}(v)\) appends \(v\) to the end of the list and returns an acknowledgment. If the list is nonempty, \(\text{pop}\) removes the last element from the list and returns it; otherwise, it simply returns a special symbol \(\bot \notin V\).

**Proposition 4.7.** A stack is not in \(G\).

**Proof.** Suppose a stack is in \(G\). Then there exist an initial configuration, an operation instance \(\Phi\) by some process \(p\), and a sequence of operation instances \(\Upsilon_q\), for each process \(q \neq p\), that satisfy the conditions of Definition 4.1.

Since a \(\text{push}\) operation returns only an acknowledgment, \(\Phi\) has to be an instance of \(\text{pop}\). If \(\Upsilon_q\) contains only instances of \(\text{pop}\) for all \(q \neq p\), then, starting from the initial configuration, after any sufficiently long interleaving of prefixes of \(\Upsilon_q\), one for each \(q \neq p\), the stack is empty and \(\Phi\) returns \(\bot\). This contradicts Definition 4.1. Therefore, there is a sequence \(\Upsilon_q\) that contains at least one instance of \(\text{push}\).

Let \(A\) be the shortest prefix of \(\Upsilon_q\) that ends in an instance of \(\text{push}\). Let \(a\) be the value pushed by the last operation instance in \(A\). Let \(Q'\) be any finite prefix of \(\Upsilon_q\) for some \(q' \neq p, q\). Let \(H\) be any interleaving of \(A\) and \(Q'\) that ends in an instance of \(\text{push}(a)\). Then \(\Phi\) returns \(a\) in both \(A\Phi\) and \(H\Phi\). This contradicts Definition 4.1.

A queue is also an object whose value is any finite list of elements from some domain \(V\) and which supports two operations. For \(v \in V\), \(\text{enqueue}(v)\) appends \(v\) to the end of the list and returns an acknowledgment. If the list is nonempty, \(\text{dequeue}\) removes the first element from the list and returns it; otherwise, it simply returns a special symbol \(\bot \notin V\).

**Proposition 4.8.** A queue is not in \(G\).

**Proof.** Suppose a queue is in \(G\). Then there are an initial configuration, an operation instance \(\Phi\) by some process \(p\), and a sequence of operation instances \(\Upsilon_q\), for each process \(q \neq p\) that satisfy the conditions of Definition 4.1.
Since an enqueue operation returns only an acknowledgment, $\Phi$ has to be an instance of dequeue. If $\Upsilon_q$ contains only instances of dequeue for all $q \neq p$, then, starting from the initial configuration, after any sufficiently long interleaving of prefixes of $\Upsilon_q$, one for each $q \neq p$, the queue is empty. Thus, $\Phi$ returns $\bot$ in both $A\Phi$ and $AQ\Phi$, contrary to Definition 4.1. Therefore, there is a sequence $\Upsilon_q$, for some process $q \neq p$, which contains at least one instance of enqueue.

Let $\delta e$ be the shortest prefix of $\Upsilon_q$ that ends in an instance of enqueue. Let $a$ be the value enqueued in $e$. Let $\ell \geq 0$ be the number of elements in the queue immediately after $\delta$ has been performed starting from the initial configuration.

Let $q' \neq q, p$ be any other process. If $\Upsilon_{q'}$ contains fewer than $\ell$ dequeues, let $A = \delta e Q$, where $Q$ is the shortest prefix of $\Upsilon_{q'}$ that contains all of its dequeues. Then for any finite prefix $QQ'$ of $\Upsilon_{q'}$, the response of $\Phi$ is the same in $A\Phi$ and $AQ\Phi$, contrary to Definition 4.1. Therefore, $\Upsilon_{q'}$ contains at least $\ell$ dequeues.

Let $Q$ be the shortest prefix of $\Upsilon_{q'}$ that contains $\ell$ dequeues, and let $A = \delta e Q$. Then $\Phi$ returns $a$ in $A\Phi$. Consider any prefix $QQ'$ of $\Upsilon_{q'}$. Let $\ell' \geq 0$ be the number of elements in the queue immediately after $\delta QQ'$ is performed starting from the initial configuration. Then $QQ'$ contains at least $\ell'$ enqueues. Let $\sigma'$ be the shortest suffix of $QQ'$ that contains $\ell'$ enqueues, and let $\sigma$ denote the remainder of $QQ'$, i.e., $\sigma\sigma' = QQ'$. After $H = \delta e \sigma\sigma'$, the queue contains $\ell' + 1$ elements, beginning with $a$. Thus, $\Phi$ returns $a$ in $H\Phi$. This contradicts Definition 4.1. □

5. A time lower bound for objects in $G$. In this section, we prove a linear lower bound on the worst case number of stalls incurred by an operation instance in any obstruction-free implementation of an object in class $G$. To do this, we use a covering argument. However, instead of using poised processes to hide information from a certain process, we use them to cause an operation instance by this process to incur $n - 1$ stalls. Specifically, we construct an execution containing a single operation instance performed by process $p$ that incurs one stall as a result of contending for an object with a single nontrivial event by each of the other processes. We call this an $(n - 1)$-stall execution. It is formally defined as follows.

**Definition 5.1.** Let $E$ be a set of executions of an implementation of an object $O$. An execution $E\sigma_1 \cdots \sigma_i \in E$ is a $k$-stall $E$-execution of object $O$ for process $p$ if

- $E$ 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 has size $k$ such that, for $j = 1, \ldots, i$,
  - each process in $S_j$ is poised to apply a nontrivial event to $O_j$ immediately after $E$; and
  - in $\sigma_j$, process $p$ applies events by itself until it is poised to apply its first event to $O_j$, then each of the processes in $S_j$ accesses $O_j$, and, finally, $p$ accesses $O_j$;
- all processes not in $S_1 \cup \cdots \cup S_i$ are idle immediately after $E$;
- $p$ 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 $E'$ of $E$, with $EE' \in E$, no process applies a nontrivial event to any base object accessed in $\sigma_1 \cdots \sigma_i$.

In a $k$-stall $E$-execution for $p$, the operation instance $\Phi$ performed by process $p$ incurs $k$ stalls, since it incurs $|S_j|$ stalls when it accesses $O_j$ for $j = 1, \ldots, i$. Note that, if the empty execution is in $E$, then it is a 0-stall $E$-execution for any process $p$. In this case, $p$ starts no operation instance. In all other cases, $p$ starts exactly one operation instance in $\sigma_1 \cdots \sigma_i$. We say that $E$ is a $k$-stall execution when $p$ and $O$ are understood and $E$ is the set of all executions of an implementation of $O$. 

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
To obtain a contradiction, suppose there is no such execution.

For process $E_{\sigma}$ execution instance $\Phi$ by some process $p$, let $\sigma$ begin in a single operation instance is at least a prefix of $\Upsilon$ it completes its operation instance $\Phi$, and then each process in accessed by sets of processes $O_i$ written above base object $O_i$ is poised to apply a nontrivial event to $O_i$. The arrow into the base object $O_i$ represents the prefix of $\sigma_i$ consisting of the solo execution by $p$ until it is poised to apply its first event to $O_i$. In $\sigma_i$, $p$ will access $O_i$ and incur $|S_i|$ stalls from the events of the processes in $S_i$. In total, $p$ will incur 8 stalls in the execution $E_{\sigma_1 \cdots \sigma_4}$.

To prove the following lower bound, we inductively construct an $(n-1)$-stall execution for some process $p$. At each step of this construction, we use Definition 4.1 and the last property of Definition 5.1 to show there is an extension in which each process $p$ accesses an additional base object at which it incurs stalls from nontrivial primitives applied by another set of processes. Among all such extensions, we choose one in which a maximal number of processes are poised at this additional base object, to ensure that the last property of Definition 5.1 will continue to hold.

**Theorem 5.2.** In any obstruction-free $n$-process linearizable implementation of an object in class $G$ from RMW base objects, the worst case number of stalls incurred by a single operation instance is at least $n-1$.

**Proof.** Let $O$ be an object in $G$. Then there are an initial value, an operation instance $\Phi$ by some process $p$, and an infinite sequence of operation instances $\Upsilon_q$ for each process $q \neq p$ that satisfy the conditions of Definition 4.1.

Let $E$ be the set consisting of the empty execution and all executions of some implementation of $O$ in which $p$ performs $\Phi$ and every process $q \neq p$ performs a finite prefix of $\Upsilon_q$. It suffices to prove the existence of an $(n-1)$-stall $E$-execution for $p$. To obtain a contradiction, suppose there is no such execution.

Let $0 \leq k \leq n-2$ be the largest integer for which there exists a $k$-stall $E$-execution for process $p$. Let $E_{\sigma_1 \cdots \sigma_i}$ be such a $k$-stall $E$-execution with base objects $O_1, \ldots, O_i$ accessed by sets of processes $S_1, \ldots, S_i$, where $|S_1 \cup \cdots \cup S_i| = k$. We will prove that there exists a $(k+k')$-stall execution for some $k' \geq 1$.

Let $\sigma$ be an extension of $E_{\sigma_1 \cdots \sigma_i}$ in which process $p$ applies events by itself until it completes its operation instance $\Phi$, and then each process in $S_1 \cup \cdots \cup S_i$ applies events by itself until it completes its operation instance. The obstruction-freedom of the implementation guarantees that $\sigma$ is finite. Let $v$ be the value returned by $\Phi$ in $E_{\sigma_1 \cdots \sigma_i}$.\!

Consider a linearization $A\Phi A'$ of the $E$-execution $E_{\sigma_1 \cdots \sigma_i}$. Then $\Phi$ returns value $v$ in $A\Phi$, and $AA'$ is an interleaving of a finite prefix of $\Upsilon_q$ for each $q \neq p$. Since all processes not in $S_1 \cup \cdots \cup S_i$ are idle immediately after $E$ and no operation instance begins in $E_{\sigma_1 \cdots \sigma_i}$ after $\Phi$’s first event, $A'$ contains at most $k \leq n-2$ operation instances, each performed by a different process in $S_1 \cup \cdots \cup S_i$.

---

$S_1 = \{p_1, p_2, p_7\} \quad S_2 = \{p_6, p_8\} \quad S_3 = \{p_4\} \quad S_4 = \{p_5, p_6\}$

**Fig. 5.1.** The configuration after the prefix $E$ of an 8-stall execution $E_{\sigma_1 \cdots \sigma_4}$ is executed.
Let \( q' \) be a process not in \( S_1 \cup \cdots \cup S_i \cup \{p\} \), and let \( Q \) be the (possibly empty) sequence of operation instances performed by \( q' \) in \( A \). Since the object \( O \) is in class \( G \) and \( A' \) is \( q' \)-free, there is a finite prefix \( QQ' \) of \( \Upsilon q' \) that satisfies the requirements of Definition 4.1.

Let \( \tau \) be the solo extension of \( E \) by process \( q' \) in which it performs all of the operation instances in \( Q \). The obstruction-freedom of the implementation guarantees that \( \tau \) is finite. Because \( E\sigma_1 \cdots \sigma_i \) is a \( k \)-stall \( E \)-execution and \( \tau \) is \( (\{p\} \cup S_1 \cup \cdots \cup S_i) \)-free, \( \tau \) applies no nontrivial event to any base object accessed in \( \sigma_1 \cdots \sigma_i \). Therefore the value of each base object accessed in \( \sigma_1 \cdots \sigma_i \) is the same immediately after \( E \) and \( E\tau \). Consequently, \( \sigma_1 \cdots \sigma_i \) is an extension of \( E\tau \). Furthermore, the value of each base object accessed in \( \sigma_1 \cdots \sigma_i \) is the same immediately after \( E\sigma_1 \cdots \sigma_i \) and \( E\tau\sigma_1 \cdots \sigma_i \).

Let \( \sigma' \) be an extension of \( E\tau\sigma_1 \cdots \sigma_i \) in which \( p \) applies events by itself until it completes its operation instance \( \Phi \), and then each process in \( S_1 \cup \cdots \cup S_i \) applies events by itself until it completes its operation instance.

Let \( H\Phi H' \) be a linearization of the operation instances performed in \( E\tau\sigma_1 \cdots \sigma_i\sigma' \). Then \( H\Phi H' \) is an interleaving of \( QQ' \) and the sequences of operation instances performed by each of the other processes in \( AA' \). Since all processes not in \( S_1 \cup \cdots \cup S_i \) are idle immediately after \( E \) and \( E\tau \) and no operation instance begins in \( E\tau\sigma_1 \cdots \sigma_i\sigma' \) after \( \Phi \)'s first event, \( H' \) contains no operation instances by \( q' \), each operation instance in \( H' \) is by a different process, and for all \( q \neq p, q' \), all or all but one of the operation instances by process \( q \) precede \( Q' \) in \( H \).

We claim that during \( \tau \), process \( q' \) applies a nontrivial event to some base object accessed by \( p \) in \( \sigma \). Suppose not. Then \( p \) applies exactly the same sequence of events in \( \sigma' \) and gets the same responses from each, as it does in \( \sigma \). Hence \( p \) will also return the value \( v \) in execution \( E\tau\sigma_1 \cdots \sigma_i\sigma' \). This implies that \( v \) is the response of \( \Phi \) in \( H\Phi \), which contradicts the fact that \( O \) is in \( G \).

Now, we construct a \((k + k')\)-stall execution \( EE'\sigma_1 \cdots \sigma_i\sigma_{i+1} \) for some \( k' \geq 1 \), with one additional base object, \( O_{i+1} \). This base object is not necessarily the base object accessed by \( p \) in \( \sigma \) to which \( q' \) applies a nontrivial event. The construction is illustrated in Figure 5.2. To ensure that the resulting execution satisfies the last requirement in Definition 5.1, as many processes as possible are poised to perform nontrivial events to \( O_{i+1} \). To achieve this, we use properties of the class \( G \).

Let \( \mathcal{F} \) be the set of all finite \((\{p\} \cup S_1 \cup \cdots \cup S_i)\)-free extensions \( F \) of \( E \) such that \( EF \in \mathcal{E} \). Let \( O_{i+1} \) be the first base object accessed by \( p \) in \( \sigma \) to which some process applies a nontrivial event during some \( F \in \mathcal{F} \). \( O_{i+1} \) is well-defined since \( \tau \in \mathcal{F} \) and, during \( \tau \), process \( q' \) applies a nontrivial event to some base object accessed by \( p \) in \( \sigma \). Since \( E\sigma_1 \cdots \sigma_i \) is a \( k \)-stall \( E \)-execution, no \( F \in \mathcal{F} \) applies a nontrivial event to any of \( O_1, \ldots, O_i \), so \( O_{i+1} \) is distinct from these base objects. Let \( k' \) be the maximum

Fig. 5.2. The construction in the proof of Theorem 5.2.
number of processes that are simultaneously poised to apply nontrivial events to $O_{i+1}$ in event sequences in $\mathcal{F}$. Let $E'$ be an extension of $E$ in $\mathcal{F}$ such that a set $S_{i+1}$ of $k'$ processes are simultaneously poised to apply nontrivial events to $O_{i+1}$ immediately after $EE'$ and all processes not in $\{p\} \cup S_1 \cup \cdots \cup S_i \cup S_{i+1}$ are idle immediately after $EE'$. Note that, by obstruction-freedom, we can extend any execution to one in which each process in a given set is idle. Specifically, for each process in the set that is not idle, append a solo sequence of events by that process, in which it completes its pending operation.

Since $E'$ is $\{p\} \cup S_1 \cup \cdots \cup S_i$-free and $E$ is $p$-free, $EE'$ is also $p$-free. Furthermore, for $j = 1, \ldots, i$, each process in $S_j$ is poised to apply a nontrivial event to $O_j$ immediately after $E$ and, hence, immediately after $EE'$.

Let $\sigma_{i+1}$ be the prefix of $\sigma$ up to, but not including $p$’s first access to $O_{i+1}$, followed by an access to $O_{i+1}$ by each of the $k'$ processes in $S_{i+1}$, followed by $p$’s first access to $O_{i+1}$. Then $\sigma_1 \cdots \sigma_{i+1}$ is an extension of $EE'$. Note that $p$ starts only one operation instance in $EE'\sigma_1 \cdots \sigma_{i+1}$.

If $\alpha$ is a $\{p\} \cup S_1 \cup \cdots \cup S_i \cup S_{i+1}$-free extension of $EE'$ with $EE'\alpha \in \mathcal{E}$, then $E'\alpha \in \mathcal{F}$. Since $E\sigma_1 \cdots \sigma_i$ is a $k$-stall $\mathcal{E}$-execution, $E'\alpha$ applies no nontrivial events to any base object accessed in $\sigma_1 \cdots \sigma_i$. By definition of $O_{i+1}$ and the maximality of $k'$, $\alpha$ applies no nontrivial events to any base object accessed in $\sigma_{i+1}$.

Hence $EE'\sigma_1 \cdots \sigma_{i+1}$ is a $(k + k')$-stall $\mathcal{E}$-execution. Since $k < k + k' \leq n - 1$, this contradicts the maximality of $k$.

6. A time lower bound for stacks and queues. Stacks and queues are not in $\mathcal{G}$. Nevertheless, in this section, we prove the same lower bound as in Theorem 5.2 on the worst case number of stalls incurred by a single instance of $\text{pop}$ or $\text{dequeue}$, provided there is a bound on the number of distinct base objects it accesses. In particular, this assumption holds for any implementation that uses a bounded amount of shared memory. We derive this lower bound using a reduction from a counter to a stack or a queue.

First, we consider any obstruction-free implementation of a counter in which there is a bound on the number of distinct base objects accessed by a single instance of $\text{fetch} & \text{increment}$. We show that there exists an execution of bounded length in which some process $p$ incurs $n - 1$ stalls while performing a single instance of $\text{fetch} & \text{increment}$. This execution is constructed inductively. However, the number of stalls does not necessarily increase at successive steps of our construction. Instead, we use a potential function and show that its value increases. This function gives more weight to stalls that $p$ incurs earlier.

**Lemma 6.1.** Consider any obstruction-free linearizable implementation of a counter with initial value 0, shared by $n$ processes, from RMW base objects. Suppose there exists a constant $d$ (which may depend on $n$) such that, in every execution, each instance of $\text{fetch} & \text{increment}$ accesses at most $d$ different base objects. Then there exists an execution that contains at most $n(n - 1)^d + n$ instances of $\text{fetch} & \text{increment}$, in which some process incurs $n - 1$ stalls while performing one of these instances.

**Proof.** Fix a process $p$ and an instance $\Phi$ of $\text{fetch} & \text{increment}$ by $p$. The construction proceeds in phases. In phase $r \geq 0$, we construct an execution $E_r \sigma_{r,1} \cdots \sigma_{r,i_r} \rho_r$ with the following properties:

- $E_r$ is $p$-free;
- there are distinct objects $O_{r,1}, \ldots, O_{r,i_r}$ and disjoint sets of processes $S_{r,1}, \ldots, S_{r,i_r}$ whose union has size $k_r$, such that, for $j = 1, \ldots, i_r$,
– each process in \( S_{r,j} \) is poised to apply a nontrivial event to \( O_{r,j} \) immediately after \( E_r \), and

– in \( \sigma_{r,j} \), process \( p \) applies events until it is poised to apply its first event to \( O_{r,j} \), then each of the processes in \( S_{r,j} \) accesses \( O_{r,j} \), and, finally, \( p \) accesses \( O_{r,j} \);

\( E_r \) contains at most \( nr \) instances of \texttt{fetch\&increment}; and

\( \rho_r \) is a solo execution by process \( p \) in which it completes \( \Phi \).

In this instance, \( p \) incurs \( k_r \) stalls. We construct such an execution with \( k_r = n - 1 \).

Note that \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \) is not necessarily a \( k_r \)-stall execution. In particular, processes not in \( S_{r,1} \cup \cdots \cup S_{r,i_r} \) may be active immediately after \( E_r \), and there may be \( \{\{p\} \cup S_{r,1} \cup \cdots \cup S_{r,i_r}\} \)-free extensions of \( E_r \) containing nontrivial events applied to objects accessed in \( \sigma_{r,1}\cdots\sigma_{r,i_r} \).

Since the number of stalls, \( k_r \), is an integer between 0 and \( n - 1 \), proving that \( k_r \) increases with \( r \) would imply that there is a phase \( r \leq n - 1 \) such that \( k_r = n - 1 \). But \( k_{r+1} \) may be smaller than \( k_r \) in our construction. Instead, we define a potential function \( \Psi : \mathbb{N} \rightarrow \{0, \ldots, (n-1)^d\} \) and prove that, if \( k_r < n - 1 \), then \( \Psi(r) < \Psi(r+1) \). This implies that there is a phase \( r \leq (n-1)^d \) such that \( k_r = n - 1 \).

To define \( \Psi(r) \), let \( \pi_r \) denote the sequence of the at most \( d \) different base objects accessed by \( p \) in the execution \( \sigma_{r,1}\cdots\sigma_{r,i_r} \) in the order they are first accessed by \( p \). In particular, each of the objects \( O_{r,1}, \ldots, O_{r,i_r} \) occurs in \( \pi_r \). Moreover, if \( j < j' \), then \( O_{r,j} \) precedes \( O_{r,j'} \) in \( \pi_r \). Suppose that \( O_{r,j} \) occurs in position \( w_r(j) \) of \( \pi_r \) for \( j = 1, \ldots, i_r \). Then let

\[
\Psi(r) = \sum_{j=1}^{i_r} |S_{r,j}| \cdot (n-1)^{d-w_r(j)}.
\]

Note that \( \Psi(r) \) can usually be viewed as a \( d \)-digit number in base \( n - 1 \) whose \( u \)th most significant digit is the number of processes in \( S_{r,1} \cup \cdots \cup S_{r,i_r} \) poised at the \( u \)th object in \( \pi_r \). (The only exception is when \( k_r = n - 1 \) processes are poised at the same object.) Thus an additional stall to an object \( O \) contributes more to the potential function than any number of stalls to objects that \( p \) first accesses after accessing \( O \).

Let \( E_0 \) denote the empty execution, which contains no instances of \texttt{fetch\&increment}. Let \( i_0 = k_0 = 0 \) and let \( p_0 \) denote the solo extension of \( E_0 \) in which \( p \) performs \( \Phi \) until it completes. Then \( \Psi(0) = 0 \).

Suppose that, for some \( r \geq 0 \), we have constructed \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) with \( k_r < n - 1 \). We will construct \( E_{r+1} \sigma_{r,1+1}\cdots\sigma_{r,i_{r+1}} \rho_{r+1} \) such that \( \Psi(r+1) > \Psi(r) \).

Since \( k_r < n - 1 \), there exists a process \( q \notin S_{r,1} \cup \cdots \cup S_{r,i_r} \cup \{p\} \). Consider the solo extension \( \gamma \) of \( E_r \) by \( q \) in which \( \gamma \) completes \( n \) instances of \texttt{fetch\&increment}. We prove that \( \gamma \) applies a nontrivial event to some base object accessed by \( p \) in \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \).

Assume not. Then \( E_r \) and \( E_r \gamma \) are indistinguishable to process \( p \). It follows that \( \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) is an extension of \( E_r \gamma \) and \( E_r \gamma \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) is indistinguishable from \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) to process \( p \). In particular, \( p \) receives the same response from \( \Phi \) in both of these executions. Let \( a \) be the number of \texttt{fetch\&increment} instances that complete in \( E_r \). Then there are \( a \) instances of \texttt{fetch\&increment} that complete in \( E_r \gamma \). Since \( p \) invokes \( \Phi \) after \( E_r \gamma \), linearizability implies that \( \Phi \)'s response in \( E_r \gamma \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) is at least \( a + n \). Since \( p \) is idle immediately after \( E_r \) and \( p \) is the only process that invokes an instance of \texttt{fetch\&increment} in \( \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \), there are at most \( a + n \) instances of \texttt{fetch\&increment} that are invoked in \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \).

By linearizability, \( \Phi \)'s response in \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \) is at most \( a + n - 1 \). This is a contradiction. Thus \( \gamma \) applies at least one nontrivial event to one of the base objects accessed by \( p \) in \( E_r \sigma_{r,1}\cdots\sigma_{r,i_r} \rho_r \).
Let $\gamma'$ be the shortest prefix of $\gamma$ such that $q$ is poised to perform a nontrivial event at one of these base objects immediately after $E_r \gamma'$. Let $E_{r+1} = E_r \gamma'$. Since $E_r$ is p-free, so is $E_{r+1}$. Since $E_r$ contains at most $nr$ instances of $fetch$ or $increment$ and $\gamma'$ contains at most $n$ instances, it follows that $E_{r+1}$ contains at most $n(r+1)$ instances.

Suppose that, immediately after $E_{r+1}$, process $q$ is poised at the object in position $u$ of $\pi_r$. Let $i_{r+1} = 1 + \#\{j \mid w_r(j) < u\}$, so $i_{r+1} - 1 \leq i_r$ is the number of objects $O_{r,j}$ that occur before position $u$ in $\pi_r$. For $j = 1, \ldots, i_{r+1} - 1$, define $O_{r+1,j} = O_{r,j}$, $S_{r+1,j} = S_{r,j}$, and $\sigma_{r+1,j} = \sigma_{r,j}$. Let $O_{r+1,i_{r+1}}$ be the object at which $q$ is poised immediately after $E_{r+1}$. There are two cases: If $O_{r+1,i_{r+1}} \in \{O_{r,1}, \ldots, O_{r,i_r}\}$, then $i_r \geq i_{r+1}$, and $O_{r+1,i_{r+1}} = O_{r,i_{r+1}}$. In this case, let $S_{r+1,i_{r+1}} = S_{r,i_{r+1}} \cup \{q\}$, and let $\sigma_{r+1,i_{r+1}}$ be the same as $\sigma_{r,i_{r+1}}$, except that $q$ accesses $O_{r+1,i_{r+1}}$ immediately before $p$ does. If $O_{r+1,i_{r+1}} \not\in \{O_{r,1}, \ldots, O_{r,i_r}\}$, let $S_{r+1,i_{r+1}} = \{q\}$, and let $\sigma_{r+1,i_{r+1}}$ denote the extension of $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}-1}$ in which process $p$ applies events until it is poised to apply its first event to $O_{r+1,i_{r+1}}$, then $q$ accesses $O_{r+1,i_{r+1}}$, and, finally, $p$ accesses $O_{r+1,i_{r+1}}$.

For $j = 1, \ldots, i_{r+1}$, each process in $S_{r+1,j}$ is poised to apply a nontrivial event to $O_{r+1,j}$ immediately after $E_{r+1}$ and, in $\sigma_{r+1,j}$, process $p$ applies events until it is poised to apply its first event to $O_{r+1,j}$, then each of the processes in $S_{r+1,j}$ accesses $O_{r+1,j}$, and, finally, $p$ accesses $O_{r+1,j}$. Let $k_{r+1} = |S_{r+1,1} \cup \cdots \cup S_{r+1,i_{r+1}}|$, and let $\rho_{r+1}$ be the solo extension of $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}-1}$ in which process $p$ completes $\Phi$. Obstruction-freedom guarantees the existence of $\rho_{r+1}$. Let $\pi_{r+1}$ be the sequence of all the different base objects accessed by process $p$ in $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}-1} \rho_{r+1}$ in the order they are first accessed by $p$. For $j = 1, \ldots, i_{r+1}$, let $w_{r+1}(j)$ denote the position of $O_{r+1,j}$ in $\pi_{r+1}$. Note that $w_{r+1}(i_{r+1}) = u$ and $\sum_{j=1}^{i_r} |S_{r,j}| = k_r < n - 1$.

Since $\sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}-1} = \sigma_{r,1} \cdots \sigma_{r,i_r-1}$, it follows that $w_{r+1}(j) = w_r(j)$ for $j = 1, \ldots, i_{r+1} - 1$. If $O_{r+1,i_{r+1}} \in \{O_{r,1}, \ldots, O_{r,i_r}\}$, then $|S_{r+1,i_{r+1}}| = |S_{r,i_{r+1}}| + 1$ and $u = w_{r+1}(i_{r+1}) = w_r(i_{r+1}) \leq w_r(j) - 1$ for $i_r + 1 < j \leq i_r$. Hence,

\[
\Psi(r) = \sum_{j=1}^{i_{r+1}-1} |S_{r,j}| \cdot (n - 1)^{d-w_r(j)} + |S_{r,i_{r+1}}| \cdot (n - 1)^{d-w_r(i_{r+1})} + \sum_{j=i_{r+1}+1}^{i_r} |S_{r,j}| \cdot (n - 1)^{d-w_r(j)} \\
\leq \sum_{j=1}^{i_r} |S_{r,j}| \cdot (n - 1)^{d-w_r(j)} + |S_{r,i_{r+1}}| \cdot (n - 1)^{d-u} + (n - 1)^{d-u} \cdot \sum_{j=i_{r+1}+1}^{i_r} |S_{r,j}| \\
\leq \sum_{j=1}^{i_{r+1}-1} |S_{r+1,j}| \cdot (n - 1)^{d-w_{r+1}(j)} + |S_{r,i_{r+1}}| \cdot (n - 1)^{d-u} + (n - 1)^{d-u} \cdot (n - 1)^{d-w_{r+1}(i_{r+1})} = \Psi(r + 1).
\]
If \( O_{r+1, i_{r+1}} \notin \{ O_{r, 1}, \ldots, O_{r, i_r} \} \), then \( |S_{r+1, i_{r+1}}| = 1 \) and either \( i_r = i_{r+1} - 1 \), in which case \( \Psi(r + 1) = \Psi(r) + |S_{r+1, i_{r+1}}| \cdot (n - 1)^{d-u} > \Psi(r) \), or \( u = w_r(i_{r+1}) < w_r(i_{r+1}) \leq w_r(j) \) for \( i_{r+1} \leq j \leq i_r \). In this last case,

\[
\Psi(r) = \sum_{j=1}^{i_{r+1}-1} |S_{r,j}| \cdot (n-1)^{d-w_r(j)} + \sum_{j=i_{r+1}}^{i_r} |S_{r,j}| \cdot (n-1)^{d-w_r(j)} \leq \sum_{j=1}^{i_{r+1}-1} |S_{r,j}| \cdot (n-1)^{d-w_r(j)} + (n-1)^{d-u-1} \sum_{j=i_{r+1}}^{i_r} |S_{r,j}| < \sum_{j=1}^{i_{r+1}-1} |S_{r,j}| \cdot (n-1)^{d-w_r(j)} + (n-1)^{d-u} = \sum_{j=1}^{i_{r+1}-1} |S_{r+1,j}| \cdot (n-1)^{d-w_r+1(j)} + |S_{r+1, i_{r+1}}| \cdot (n-1)^{d-w_r+1(i_{r+1})} = \Psi(r + 1).
\]

Thus \( k_r < n-1 \) implies \( \Psi(r) < \Psi(r+1) \). Since the range of \( \Psi \) is \( \{0, \ldots, (n-1)^d\} \), it follows that \( k_r = n-1 \) for some \( r \leq (n-1)^d \). The total number of instances of fetch\&increment contained in \( E_r \) is at most \( nr \leq n(n-1)^d \). No process performs more than one instance in \( \sigma_{r,1} \cdots \sigma_{r,i_r}, \rho_r \). Therefore the total number of instances of fetch\&increment contained in \( E_r \sigma_{r,1} \cdots \sigma_{r,i_r}, \rho_r \) is at most \( n(n-1)^d + n \).

To obtain our lower bound, we show that a stack or queue can be used to implement a counter that supports any bounded number of instances of fetch\&increment.

**Theorem 6.2.** In any obstruction-free \( n \)-process linearizable implementation of a stack or queue from RMW base objects, either the worst case number of stalls incurred by a single instance of pop or dequeue is at least \( n-1 \) or there is no bound on the number of different base objects that a single instance of pop or dequeue can access.

**Proof.** Assume there is an obstruction-free \( n \)-process linearizable implementation of a stack or queue from RMW objects and a constant \( d \) (which can depend on \( n \)) such that each instance of pop or dequeue accesses at most \( d \) different base objects. A stack or queue can be used to implement a counter with initial value 0 shared by \( n \) processes on which up to \( N = n(n-1)^d + n \) instances of fetch\&increment can be performed. Specifically, the stack is initialized with the list of elements \( N-1, \ldots, 1, 0 \), and the queue is initialized with the list of elements \( 0, 1, \ldots, N-1 \). To perform fetch\&increment on the counter, a process simply applies pop or dequeue. The response it receives will be the number of instances of fetch\&increment that were linearized before it.

An implementation of a counter from RMW objects can be obtained by composing the implementation of counter from a stack (or queue) with the implementation of a stack (or queue) from RMW objects. By Lemma 6.1, there is an execution of this implementation that contains at most \( N \) instances of fetch\&increment and in which some process incurs \( n-1 \) stalls while performing one of these instances. This implies that the number of stalls incurred by the corresponding instance of pop or dequeue incurs at least \( n-1 \) stalls.

**7. Conclusions.** We formally prove the intuitive idea that there are objects whose nonblocking linearizable implementations are inherently sequential. The results in this paper suggest that, as multicore machines grow in size, it might be beneficial to replace linearizable implementations of strongly ordered data structures, such as stacks and queues, with more relaxed data structures, such as pools and bags.
On a technical level, we note that the technique we used in the proof of Lemma 6.1 was employed in subsequent work by Attiya et. al [5], in which they studied the complexity of obstruction-free implementations of data structures. One of the issues they investigated was the complexity of solo-fast implementations, in which processes can apply only historyless primitives when there is no contention but can apply additional, stronger, primitives upon encountering contention. Using a variation of our proof technique, they prove a logarithmic lower bound on the contention-free complexity of solo-fast implementations. Our hope is that the basic techniques we have presented will find their way into further results in the field.

Acknowledgment. We thank the anonymous referees for their many helpful comments.

REFERENCES


