Linear Lower Bounds on Real-World Implementations of Concurrent Objects

Faith Ellen Fich
University of Toronto

Danny Hendler
University of Toronto

Nir Shavit
Sun Microsystems Laboratories

July 26, 2005

Abstract

This paper proves $\Omega(n)$ lower bounds on the time to perform a single instance of an operation in any implementation of a large class of data structures shared by $n$ processes. For standard data structures such as counters, stacks, and queues, the bound is tight. The implementations considered may apply any deterministic primitives to a base object. No bounds are assumed on either the number of base objects or their size. Time is measured as the number of steps a process performs on base objects and the number of stalls it incurs as a result of contention with other processes.
1 Introduction

The design of concurrent data structures for multiprocessor machines is an important area of research. It is becoming crucial in practice with the advent of libraries such as the Java™ Concurrency Package [17] (of JDK 1.5.0, distributed onto 10 million machines) and the introduction of multi-threaded multi-core chips onto the desktop. Because these machines involve extensive context switching, their behavior is essentially asynchronous, one process cannot wait for other processes to take steps. To overcome the inherent limitations of programming on such machines, many of the designed data structures in libraries such as the Java Concurrency Package are non-blocking, that is, they avoid the use of locks. Unfortunately, even though these concurrent data structures are widely available and have been extensively researched (see [19] for a detailed survey of the literature), we lack a basic understanding of the limitations in achieving high scalability in their design. Even for standard concurrent data structures, such as counters, queues, and stacks, implemented using any read-modify-write synchronization primitives, known implementations require time linear in \( n \), the number of processes. The best lower bounds that had been attained for implementations that use arbitrary read-modify-write primitives were in \( \Omega(\sqrt{n}) \) [11]. Thus, it was open whether the linear upper bounds were inherent.

This paper provides a matching linear lower bound for non-blocking concurrent implementations of a large class of objects, including real-world 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.

At the core of our paper, we use a new variant of a covering argument [6, 10] to prove linear time lower bounds on a class of objects that includes shared counters [14] and single-writer snapshots [3, 1, 5]. Covering arguments bring processes to a state in which they are poised to overwrite certain shared objects, causing a loss of information, which leads to incorrect behavior. Our proof technique 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 each other process in the system. It does not matter for the proof whether these stalls are on the same or different objects.

This lower bound proof does not apply to objects such as queues and stacks. However, we are able to prove a linear lower bound on implementations of these objects by way of a reduction. For example, if we initialize a queue with sufficiently many consecutive integers and use dequeue operations to return these numbers, we obtain an implementation of a counter that can support a bounded number of fetch&increment operations. We construct an execution of bounded length, in which \( n - 1 \) stalls are incurred by a process performing a single instance of fetch&increment, under the assumption that any process performing an instance of fetch&increment accesses less than \( n \) distinct base objects. This gives us the desired lower bound, since either some process takes linear time to access the \( n \) base objects, or the length of the execution can be bounded so that a queue can be used to implement the counter.

1.1 Background

There has been extensive work on lower bounds in shared memory computation, and the reader can find a survey in [10]. In this extended abstract, we will focus on recent work aimed at deriving
lower bounds for implementing common data structures on real machines.

Obstruction-freedom, introduced by Herlihy, Luchangco, and Moir [13], is perhaps the weakest natural non-blocking progress condition. An obstruction-free implementation of a concurrent object guarantees that any process can complete an operation in a finite number of its own steps if it runs alone long enough. In other words, it prevents the use of critical sections, but makes no progress guarantee when processes contend for the object, leaving progress to real-world probabilistic mechanisms such as backoff. Linearizability is a consistency condition for concurrent objects introduced by Herlihy and Wing [15] which essentially says that all operations on the object appear to happen atomically.

Apart from simple read and write operations, modern multiprocessor machines support special read-modify-write (RMW) synchronization primitives such as fetch&increment, compare&swap, or load-linked/store-conditional. The time to complete an operation is influenced not only by the number of objects a process must access, but also by the amount of contention it incurs at an object when other processes access it concurrently. To capture this real world behavior, researchers such as Merritt and Taubenfeld [18], Cypher [7], and Anderson and Kim [2] have devised complexity measures, based on an approach suggested by Herlihy, Dwork, and Waarts [8], in which time complexity counts not only the number of accesses to shared objects, but also the number of stalls, the delays as a result of waiting for other processes that access the object at the same time.

One can thus formalize the goals of recent lower bound research on non-blocking real-world concurrent data structures as providing time lower bounds on implementations that take into consideration not just the number of steps performed by an operation but also the number of stalls it incurs.

To better understand this goal, consider, for example, the question of implementing a shared concurrent counter. If the hardware supports, say, a fetch&increment synchronization primitive, then the simplest way of implementing a concurrent counter, shared by \( n \) processes, is by using the following straight-forward non-blocking (and, in fact, wait-free [12]) implementation: all processes share a single object on which each performs a fetch&increment operation to get a number. Unfortunately, this implementation has a serious drawback: it is essentially sequential. In the unfortunate case where all processes attempt to get a number from the counter simultaneously, one unlucky process will incur a time delay linear in \( n \) while waiting for all other earlier processes to complete their operations. To overcome this problem, researchers have proposed using highly distributed non-blocking coordination structures such as counting networks [4, 14]. Counting networks use multiple base objects to implement shared counters that ensure that many processes can never access a single object at the same time. However, all such structures provide counters that are either not linearizable or require linear time [14]. For counters, the implication of the lower bounds in our paper is that, in the worst case, there is no implementation that has time complexity better than the straight-forward centralized solution.

Jayanti, Tan, and Toueg [16] prove linear time and space lower bounds for implementations of a class of objects, called perturbable, from historyless primitives [9] and resettable consensus. Some key objects in our class, such as counters and single-writer snapshots, are also perturbable. Their result is stronger that ours in the following sense: they only count shared memory accesses, not stalls. However, the set of historyless primitives is a proper subset of the read-modify-write primitives and does not include real-world primitives such as fetch&increment or compare&swap.
2 Model

We consider a standard model of an asynchronous shared memory system, in which processes communicate by applying operations to shared objects. An object is an instance of an abstract data type. It is specified by a set of possible values and by a set of operations that provide the only means to manipulate it. No bound is assumed on the size of an object (i.e. the number of different possible values the object can have). An implementation of an object that is shared by a set $P$ of $n$ processes provides a specific data-representation for the object from a set $B$ of shared base objects, each of which is assigned an initial value, and algorithms for each process in $P$ to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects primitives. We reserve the term operations for the objects being implemented. We also say that an operation of an implemented object is performed and that a primitive is applied to a base object.

We consider base objects that support a set of atomic read-modify-write (RMW) primitives. A RMW 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-and-add is an example of a RMW primitive. Its update function is $g(v, w) = v + w$, and its response value is $v$, the previous value of the base object. Fetch-and-increment is a special case of fetch-and-add where $w$ always equals 1. Read is also a RMW 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 a RMW primitive. Its update function is $g(v, w) = w$, and its response function is $h(v, w) = \text{ack}$. A RMW primitive is nontrivial if it may change the value of the base object to which it is applied. Read is an example of a trivial primitive.

We consider obstruction-free implementations, in which each process is guaranteed to complete an operation within a finite number of its own steps if it runs alone long enough. Each step consists of some local computation and one shared memory event, which is a RMW primitive applied to a base object $r \in B$. For brevity, we may say that a process applies a RMW event $e$ to $r$ instead of saying that $e$ is an application of a RMW primitive by $p$ to $r$. We also say that event $e$ accesses $r$.

An execution is a (finite or infinite) sequence of events. For a finite execution $E$ and an execution $E'$, the execution $EE'$ denotes the concatenation of $E$ and $E'$. Unless stated otherwise, an execution starts from an initial configuration, in which all base objects in $B$ have their initial values and all processes are idle. If $r \in B$ is a base object and $E$ is a finite execution, then $r$’s value after $E$ is the value of $r$ at the end of execution $E$. If no event in $E$ changes the value of $r$, then $r$’s value after $E$ is the initial value of $r$. We say that an execution $E$ is $p$-free if process $p$ applies no events in $E$. In a solo execution, all events are by the same process.

Suppose a process $p$ wants to perform an operation $Op$ on an implemented object $O$. The implementation of $O$ provides an algorithm for performing $Op$, which $p$ executes. While executing this algorithm, $p$ applies a sequence of events to base objects. Which events are applied by $p$ is a function of the input parameters to the operation and may also depend on events that other processes apply. In general, $p$ may perform many operations on the implemented object in an execution. We call each of these an operation instance.

Formally, an operation instance, $\Phi = (O, Op, p, w)$, is an application by process $p$ of operation $Op$ with input parameters $w$ to the implemented object $O$. In an execution, each process performs a sequence of operation instances to the implemented object. 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 $E$. 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.

We say that a process $p$ is active after $E$ if $p$ is in the middle of performing some operation instance $\Phi$. If $p$ is not active after $E$, we say that $p$ is idle after $E$. If a process is active in the configuration resulting from a finite execution, it has exactly one enabled event, which is the next event it will apply, as specified by the algorithm it is using to apply its current operation instance to the implemented object. If a process is idle and has not yet begun a new operation instance then it has no enabled event. If a process is idle but has begun a new operation instance, then the first event of that operation instance is enabled. If a process $p$ has an enabled event $e$ after execution $E$ we say that $p$ is poised to perform $e$ after $E$.

In all 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 caused by contention in accessing the object. The concept of memory stalls was introduced in [8]. We use the following definition of memory stalls. It is stricter than that of [8], as it does not count stalls caused by read events. Thus our lower bounds also apply using the definition of stalls in [8].

**Definition 1** Let $e$ be an event applied by a process $p$ as it performs an operation instance $\Phi$ in execution $E$. Let $r$ be the base object accessed by $e$. Also let $E = E_0e_1 \ldots e_ke_{E_1}$, where $e_1 \ldots e_k$ is a maximal sequence of $k \geq 1$ nontrivial consecutive events, by distinct processes other than $p$, that access $r$. Then we say that $\Phi$ incurs $k$ memory stalls in $E$ on account of $e$. The number of stalls incurred by $\Phi$ in $E$ is the sum of all memory stalls $\Phi$ incurs in $E$ over all the events of $\Phi$ in $E$.

The primary correctness condition for implementations of concurrent objects is linearizability. Linearizability was defined formally by Herlihy and Wing [15]. Informally, an execution is linearizable if every operation instance appears to take atomic effect at some instance during the interval in which it is performed. The resulting sequence of operation instances is the linearization of the execution. An implementation is linearizable if all its executions are linearizable. All our proofs apply to linearizable implementations. In this extended abstract, we only consider deterministic implementations.

### 3 The Class $\mathcal{G}$

In this section, we define a general class $\mathcal{G}$ of objects to which our result applies. Roughly, objects in this class have an operation whose response can be changed by a sequence of operations performed before it.

**Definition 2** An object $O$ shared by $n$ processes is in the class $\mathcal{G}$ if it supports an operation $Op$ such that, for every sequence of operation instances $A\Phi A'$ on $O$, where

- $\Phi$ is an instance of $Op$ by process $p$,
- no operation instance in $AA'$ is by $p$,
- there are at most $n - 2$ different processes that perform operation instances in $AA'$, and
- each operation instance in $A'$ is by a different process,
there is a sequence of operation instances $Q$ on $O$ by a process $q$ that performs none of the operation instances in $A \Phi A'$ such that, in every sequence of operation instances $H \Phi H'$, where

- $HH'$ is an interleaving of $Q$ and the sequences of operation instances performed by each process in $AA'$,
- $H'$ contains no operation instances by $q$, and
- each operation instance in $H'$ is by a different process,

the response of $\Phi$ is different than in $A \Phi$.

Many common objects are in $G$. Two examples are counters and single-writer snapshots. A \textit{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, \textit{fetch$\&$increment modulo $m$}. The \textit{fetch$\&$increment} modulo $m$ operation atomically increments the value of the object to which it is applied modulo $m$ and returns the previous value of the object.

\textbf{Theorem 3} A \textit{modulo-$m$ counter} object shared by $n \leq m$ processes is in $G$.

\textbf{Proof:} The only operation $Op$ supported by a modulo-$m$ counter is fetch$\&$increment modulo $m$. Let $A \Phi A'$ be a sequence of operation instances on this object such that $\Phi$ is an instance of $Op$ by process $p$ and all other operation instances are by some subset of at most $n-2$ other processes. Suppose each operation instance in $A'$ is by a different process. 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 $Q$ be a sequence of $b = n - a' - 1$ instances of $Op$ by a process $q$ that performs none of the operation instances in $A \Phi A'$. Consider any sequence of operation instances $H \Phi H'$ where $HH'$ is an interleaving of $Q$ and the sequences of operation instances performed by each process in $AA'$. Suppose that $H'$ contains no operation instances by $q$ and each operation instance in $H'$ is by a different process. Then $H'$ contains at most $n - 2$ operation instances by $q$ and $H$ contains between $a + a' + b - (n-2) = a + 1$ and $a + a' + b = a + n - 1$ operation instances. Thus, by linearizability, the response of $\Phi$ in $H \Phi H'$ 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 is equal to $a \mod m$. $\blacksquare$

A \textit{counter} is an object whose set of values is the integers. It supports a single parameterless operation, \textit{fetch$\&$increment}, that atomically increments the value of the object to which it is applied and returns the previous value of the object. The following theorem is an immediate corollary of Theorem 3.

\textbf{Theorem 4} A \textit{counter} object shared by any number of processes is in $G$.

The value of a \textit{single-writer snapshot} object over sets of elements $V_1, \ldots, V_n$ is a vector of $n$ components, $C_1, \ldots, C_n$, where component $C_i$ assumes an element from $V_i$. We assume that $|V_i| \geq 2$ for all $i \in \{1, \ldots, n\}$. A single-writer snapshot object supports two operations: \textit{scan} and \textit{update}. The operation instance $update(v)$ by process $i$ sets the value of component $C_i$ to $v$. A \textit{scan} operation instance returns a vector consisting of the values of the $n$ components.

\textbf{Theorem 5} A \textit{single-writer snapshot} object is in $G$. 

6
Proof: Let $Op$ be a scan operation. Let $A\Phi A'$ be a sequence of scan and update operation instances on this object such that $\Phi$ is a scan operation instance by process $p$ and all other operation instances are by some subset of at most $n-2$ other processes. Let $Q$ be the sequence that consists of a single instance of the update operation by a process $q$ that performs none of the operation instances in $A\Phi A'$ and which changes the value of its component from its initial value.

Consider any sequence of operation instances $H\Phi H'$ where $HH'$ is an interleaving of $Q$ and the sequences of operation instances performed by each process in $AA'$. Suppose that $H$ contains the instance of update by $q$. Then the response of $\Phi$ in $A\Phi$ and in $H\Phi H'$ differ in component $q$.

However, there are common objects that are not in $G$. One example is a stack. Since a push operation only returns an acknowledgement, $\Phi$ would have to be a pop. Let $A$ consist of a single push of value 0. Then for any sequence $Q$ of operation instances, the same value is returned by $\Phi$ in $A\Phi$ and $QA\Phi$.

Likewise, a queue is not in $G$. Since an enqueue operation only returns an acknowledgement, $\Phi$ would have to be a dequeue. Let $A$ consist of a single enqueue of value $v$. Then $\Phi$ returns $v$ in $A\Phi$. For any sequence of operation instances $Q$, let $b$ denote the size of the queue after $Q$ and let $Q = Q'Q''$, where $Q''$ is the shortest suffix of $Q$ that contains $b$ enqueue operations. Then $\Phi$ returns $v$ in $Q'AQ''\Phi$.

There are also perturbable [16] objects that are not in $G$. A $k$-valued compare&swap object, for some $k > 1$, supports the operations read and $c&s(u,v)$ for all $u, v \in \{1, \ldots, k\}$, $u \neq v$. The set of values the object can assume is $\{1, \ldots, k\}$. If the value of the object is $u$, the $c&s(u,v)$ 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. A $k$-valued compare&swap object is perturbable for $k \geq n$ [16].

Theorem 6 A $k$-valued compare&swap object shared by 3 or more processes is not in $G$.

Proof: Let $A$ be the sequence of instances $c&s(2,1)c&s(3,1)\cdots c&s(k,1)$ performed by process $r$. Then, regardless of the initial value of the object, the value of the object after $A$ is 1. Let $Q$ be any sequence of read and/or compare&swap instances by some process $q \neq r$. The value of the object after $QA$ is 1. Let $\Phi$ be an instance of the operation that witnesses the membership of the object in $G$, performed by a process $p \notin \{q, r\}$. $\Phi$ returns the same value in $A\Phi$ and in $QA\Phi$ regardless of whether it is an instance of read or an instance of compare&swap, hence the object is not in $G$.

4 A Time Lower Bound

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 stalls.

Definition 7 An execution $E\sigma_1\cdots\sigma_i$ is a k-stall execution for process $p$ if

- $E$ is $p$-free,
Figure 1: The configuration after the prefix $E$ of a 19-stall execution $E\sigma_1 \cdots \sigma_5$ is executed.

- 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 RMW event to $O_j$ at the end of $E$, and
  - in $\sigma_j$, process $p$ applies events by itself until it is first poised to apply a RMW 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 at the end of $E$,
- $p$ 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 execution that immediately follows $E$, no process applies a nontrivial RMW event to any base object accessed in $\sigma_1 \cdots \sigma_i$.

We say that $E$ is a $k$-stall execution when $p$ is understood. Note that the empty execution is a 0-stall execution for any process. In a $k$-stall execution, an operation instance performed by process $p$ incurs at least $k$ stalls, since it incurs $|S_j|$ stalls when it accesses $O_j$, for $j = 1, \ldots, i$.

Figure 1 depicts the configuration that is reached after prefix $E$ of a 19-stall execution $E\sigma_1 \cdots \sigma_5$ is executed. Each of the processes depicted above base object $O_i$ is poised to apply a nontrivial event to $O_i$. The arrows show the path taken by $p$ when it incurs all of the stalls caused by these events. In this configuration, process $p$ has not yet begun to perform its operation instance. Only processes in the set $\bigcup_{i=1}^5 S_i$ may be active in this configuration. Figure 2 continues this example and depicts the configuration that is reached after the prefix $E\sigma_1\sigma_2$ of that same 19-stall execution has been executed. In $\sigma_1$, $p$’s operation instance has accessed $O_1$ and incurred 5 stalls from the events of the processes in $S_1$. Then, in $\sigma_2$, $p$’s operation instance incurred additional 4 stalls when it accessed $O_2$. In this example, at the end of $E\sigma_1\sigma_2$ process $p$ is poised to access base object $O_3$. Thus, $\sigma_3$ consists of the three events applied to $O_3$ by the processes in $S_3$, followed by the event of $p$ that accesses $O_3$. Consequently, the operation instance performed by $p$ incurs 3 more stalls in $\sigma_3$.

**Theorem 8** Consider any obstruction-free $n$-process implementation of an object $O$ in class $G$ from RMW base objects. Then the worst case number of stalls incurred by a single operation instance is at least $n - 1$.

**Proof:** Fix a process $p$. It suffices to prove the existence of an $(n - 1)$-stall execution for $p$. To obtain a contradiction, we suppose there is no such execution.

Let $0 \leq k < n - 1$ be the largest integer for which there exists a $k$-stall execution for process $p$ in which $p$ is performing an instance $\Phi$ of the operation that witnesses the membership of $O$ in $G$. 

8
Let $E\sigma_1 \cdots \sigma_i$ be such a $k$-stall execution with base objects $O_1, \ldots, O_i$ accessed by sets of processes $S_1, \ldots, S_i$, where $|S_1 \cup \cdots \cup S_i| = k$.

Let $E\sigma_1 \cdots \sigma_i \sigma_p \sigma$ be an execution in which, following $E\sigma_1 \cdots \sigma_i$, $p$ applies events by itself in $\sigma_p$ until it completes its first operation instance $\Phi$ and then, in $\sigma$, 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 $\sigma_p$ is finite and that a finite $\sigma$ exists. Let $v$ be the value returned by $\Phi$ in $E\sigma_1 \cdots \sigma_i \sigma_p$.

Consider a linearization $A \Phi A'$ of the operation instances performed in $E\sigma_1 \cdots \sigma_i \sigma_p \sigma$. Then $\Phi$ returns value $v$ in $A \Phi$. Furthermore, in $E\sigma_1 \cdots \sigma_i \sigma_p \sigma$, there are $k \leq n - 2$ active processes when $\Phi$ begins and no operation instance begins after $\Phi$. Therefore $A'$ contains at most $n - 2$ operation instances, each performed by a different process.

Since the object $O$ is in class $G$, Definition 2 implies that there is a sequence of operation instances $Q$ by a process $q \not\in S_1 \cup \cdots \cup S_i \cup \{p\}$ such that value $v$ is not returned by $\Phi$ in $H \Phi H'$, where $H H'$ is an interleaving of $Q$ and the sequences of operation instances performed by each process in $A A'$, $H'$ contains no operation instances by $q$, and each operation instance in $H'$ is by a different process.

Let $E \tau$ be the execution in which, following $E$, process $q$ applies events by itself until it completes 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 execution and $\tau$ is $\{\{p\} \cup S_1 \cup \cdots \cup S_i\}$-free, $\tau$ applies no nontrivial RMW event to any base object accessed in $\sigma_1 \cdots \sigma_i$. Consequently, $E \tau \sigma_1 \cdots \sigma_i$ exists. Let $E \tau \sigma_1 \cdots \sigma_i \sigma_p' \sigma'$ be an execution in which, following $E \tau \sigma_1 \cdots \sigma_i$, $p$ applies events by itself in $\sigma_p'$ until it completes its first operation instance $\Phi$ and then, in $\sigma'$, 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 $E \tau \sigma_1 \cdots \sigma_i \sigma_p' \sigma'$. Then $H H'$ is an interleaving of $Q$ and the sequences of operation instances performed by each process in $A A'$. When $\Phi$ begins, $q$ is not active, there are $k \leq n - 2$ active processes, and no operation instance begins after $\Phi$. Therefore $H'$ contains no operation instances by $q$, and each operation instance in $H'$ is by a different process.

We claim that during $\tau$, process $q$ applies a nontrivial RMW event to some base object accessed by $p$ in $\sigma_p$. Suppose not. As $\tau$ applies no nontrivial RMW event to any base object accessed in $\sigma_1 \cdots \sigma_i$, the values of all base objects accessed in $\sigma_1 \cdots \sigma_i$ are the same at the end of executions $E$ and $E \tau$. Consequently, the values of these base objects are also the same at the end of executions $E \sigma_1 \cdots \sigma_i$ and $E \tau \sigma_1 \cdots \sigma_i$. Thus $\sigma_p = \sigma_p'$ and $p$ will also return the value $v$ in execution $E \tau \sigma_1 \cdots \sigma_i \sigma_p$. This implies that $v$ is the response of $\Phi$ in $H \Phi H'$, which contradicts the fact that $O$ is in $G$.

![Figure 2: The configuration after the prefix $E\sigma_1 \sigma_2$ of a 19-stall execution $E\sigma_1 \cdots \sigma_5$ is executed.](image-url)
Let $\mathcal{F}$ be the set of all finite $(\{p\} \cup S_1 \cup \cdots \cup S_i)$-free executions starting immediately after $E$. Let $O_{i+1}$ be the first base object accessed by $p$ in $\sigma_p$ to which some process applies a nontrivial RMW event during executions in $\mathcal{F}$. $O_{i+1}$ is well-defined, since $\tau \in \mathcal{F}$ and, during $\tau$, $q$ applies a nontrivial RMW event to some base object accessed by $p$ in $\sigma_p$. Since $E\sigma_1 \cdots \sigma_i$ is a $k$-stall execution, no execution in $\mathcal{F}$ applies a nontrivial RMW event to $O_{i+1}$, so $O_{i+1}$ is distinct from these base objects. Let $k'$ be the maximum number of processes that are simultaneously poised to apply nontrivial RMW events to $O_{i+1}$ in executions in $\mathcal{F}$. Let $E'$ be a minimal length execution in $\mathcal{F}$ such that a set $S_{i+1}$ of $k'$ processes are simultaneously poised to apply nontrivial RMW events to $O_{i+1}$ at the end of $E'$.

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 RMW event to $O_j$ at the end of $E$ and hence at the end of $EE'$.

Let $\sigma_{i+1}$ be the prefix of $\sigma_p$ 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 $O_{i+1}$. 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 execution starting immediately after $EE'$, then $E'\alpha \in \mathcal{F}$. Since $E\sigma_1 \cdots \sigma_i$ is a $k$-stall execution, $\alpha$ applies no nontrivial RMW 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 RMW events to any base object accessed in $\sigma_{i+1}$.

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

5 Stacks and Queues

We do not know whether the result of Theorem 8 holds for stacks or queues. However, we are able to prove that, for any obstruction-free $n$-process implementation of a stack or queue, the worst-case number of events and stalls incurred by a process as it performs an operation instance is $n$. We derive this result using a reduction from counters to stacks and queues.

We begin by showing that, for any obstruction-free implementation of a counter, either there is an execution in which a process accesses a linear number of different base objects while performing a single instance of fetch&increment, or there is an execution of bounded length in which a process incurs a linear number of memory stalls while performing a single instance of fetch&increment.

Then we show that a stack or queue can be used to implement a counter that supports any bounded number of operation instances.

**Lemma 9** Consider any obstruction-free implementation of a counter shared by $n$ processes, in which the maximum number of distinct objects accessed by a process while performing a single instance of fetch&increment is at most $d$. Then there exists an execution that contains at most $(n-2)(n-1)^d + n$ operation instances and in which some process incurs $(n-1)$ stalls while performing one instance of fetch&increment.

**Proof:** Fix a process $p$ and an instance $\Phi$ of fetch&increment by $p$. The construction proceeds in phases.

Let $E_0$ denote the empty execution, let $i_0 = k_0 = 0$, and let $E_0\rho_0$ denote the solo execution by $p$ in which it performs $\Phi$ (until completed).

In phase $r \geq 1$, 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 perform a nontrivial RMW event on $O_{r,j}$ at the end of $E_r$, and

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

• $\rho_r$ is a solo execution by process $p$ in which it completes $\Phi$.

In this execution, $p$ incurs $k_r$ stalls.

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 $S_{r,1} \cup \cdots \cup S_{r,i_r}$ are not necessarily idle at the end of $E_r$, and, in executions following $E_r$, these processes may perform nontrivial RMW events on the objects accessed in $\sigma_{r,1} \cdots \sigma_{r,i_r}$.

We assign an integer value $\Psi_r \in [0, (n - 2)(n - 1)^{d-1} - 1]$ to each phase. If $k_r < n - 1$, we are able to do the next phase and assign it a value $\Psi_{r+1} > \Psi_r$. This implies that, eventually, there is a phase $r \leq (n - 2)(n - 1)^{d-1} + 1$ with $k_r = n - 1$.

Specifically, let $\Pi_r$ denote the sequence of objects accessed by $p$ in the execution $E_r \sigma_{r,1} \cdots \sigma_{r,i_r} \rho_r$, ordered according to when they are first accessed by $p$. In particular, $O_{r,j}$ precedes $O_{r,j'}$ in $\Pi_r$ if $j < j'$. Then $\Psi_r = \sum\{(n - 1)^{w_r(q)} | q \in S_{r,1} \cup \cdots \cup S_{r,i_r}, \text{ where } w_r(q) = \sum\{1 \cdots u \text{ if } q \in S_{r,j} \text{ and } O_{r,j} \text{ is } u \text{th object in } \Pi_r \text{. Note that } \Psi_0 = 0 \}$.

Given $E_r \sigma_{r,1} \cdots \sigma_{r,i_r} \rho_r$ with $k_r < n - 1$, we construct $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}} \rho_{r+1}$ such that $\Psi_{r+1} > \Psi_r$. Since $k_r < n - 1$, there is a process $q' \notin S_{r,1} \cup \cdots \cup S_{r,i_r} \cup \{p\}$. After $E_r$, perform a solo execution $\gamma$ by $q'$ until it is poised to apply a nontrivial RMW event to some base object accessed by $p$ in execution $E_r \sigma_{r,1} \cdots \sigma_{r,i_r} \rho_r$. Call this execution $E_{r+1}$. Since $E_r$ is $p$-free, so is $E_{r+1}$.

As in the proof of Theorem 3, it can be shown that $\gamma$ contains at most $n - 1$ instances of fetch&increment. Hence $E_{r+1}$ contains at most $n - 1$ more instances of fetch&increment than $E_r$ does.

Suppose that the base object at which $q'$ is poised at the end of $E_{r+1}$ is the $u$th object in $\Pi_r$. Let $i_{r+1}$ be one more than the number of objects $O_{r,j}$ that occur before position $u$ in $\Pi_r$. Then define $O_{r+1,j} = O_{r,j}$, $S_{r+1,j} = S_{r,j}$, and $\sigma_{r+1,j} = \sigma_{r,j}$, for $j = 1, \ldots, i_{r+1} - 1$. Let $O_{r+1,i_{r+1}}$ be the object at which $q'$ is poised at the end of $\gamma$. If $O_{r+1,i_{r+1}} = O_{r,i_{r+1}}$, 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. Otherwise, $O_{r+1,i_{r+1}} \notin \{O_{r,1}, \ldots, O_{r,i_r}\}$. In this case, we let $S_{r+1,i_{r+1}} = \{q'\}$ and we let $\sigma_{r+1,i_{r+1}}$ denote the execution following $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1} - 1}$ in which process $p$ applies events until it is poised to apply its first RMW event to $O_{r+1,i_{r+1}}$, then $q'$ accesses $O_{r+1,i_{r+1}}$, and, finally, $p$ accesses $O_{r+1,i_{r+1}}$.

Then, for $j = 1, \ldots, i_{r+1}$, each process in $S_{r+1,j}$ is poised to perform a nontrivial RMW event on $O_{r+1,j}$ at the end of $E_{r+1}$ and in $\sigma_{r+1,j}$, process $p$ applies events until it is poised to apply its first RMW 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 a solo execution by process $p$ following $E_{r+1} \sigma_{r+1,1} \cdots \sigma_{r+1,i_{r+1}}$ in which it completes $\Phi$. Obstruction-freedom guarantees the existence of $\rho_{r+1}$.
Note that \( w_{r+1}(q') = d - u \) and \( w_{r+1}(q) = w_r(q) \) for all other processes \( q \in S_{r+1,1} \cup \cdots \cup S_{r+1,i_r+1} \). Furthermore, each process \( q \in S_{r,1} \cup \cdots \cup S_{r,i_r} \setminus (S_{r+1,1} \cup \cdots \cup S_{r+1,i_r+1}) \) is poised at an object at position \( u+1 \) or greater in \( \Pi_r \), so \( w_r(q) \leq d - u - 1 \). Since \( |S_{r,j}| \leq n - 2 \) for all \( j \), we have 

\[
(n - 1)^{d - u} > \sum \{(n - 1)^{w_r(q)} \mid q \in S_{r,1} \cup \cdots \cup S_{r,i_r} \setminus (S_{r+1,1} \cup \cdots \cup S_{r+1,i_r+1})\}.
\]

It follows that 

\[
\Psi_{r+1} = \sum \{(n - 1)^{w_r(q)} \mid q \in S_{r+1,1} \cup \cdots \cup S_{r+1,i_r+1}\} > \sum \{(n - 1)^{w_r(q)} \mid q \in S_{r,1} \cup \cdots \cup S_{r,i_r}\} = \Psi_r.
\]

Hence there is an execution \( E_r \sigma_r \cdots \sigma_{r,r} p_r \), with \( r \leq (n - 2)(n - 1)^{d - 1} + 1 \) in which \( p \) incurs \( n - 1 \) stalls. The total number of operation instances contained in this execution is at most 

\[(n - 2)(n - 1)^d + n.\]

**Theorem 10** Consider any implementation of an obstruction free linearizable queue or stack, \( R \), from RMW objects, that is shared by \( n \) processes. Then the worst case number of events and stalls incurred by a process as it performs an operation instance on \( R \) is at least \( n \).

**Proof:** If the maximum number of distinct objects accessed by a process while performing an instance of an operation on \( R \) is \( n \) or more, we are done. Otherwise, we use \( R \) to implement a counter shared by \( n \) processes on which \( (n - 2)(n - 1)^d + n \) operation instances can be performed. Specifically, \( R \) is initialized by enqueuing the consecutive numbers 0, 1, \( \ldots \), \( (n - 2)(n - 1)^d + n \) or pushing the consecutive numbers \( (n - 2)(n - 1)^d + n \), \( \ldots \), 1, 0.

To perform a fetch&increment operation on the counter, a process simply applies a dequeue or pop. The response it receives will be the number of instances of fetch&increment that have occurred earlier.

By Lemma 9, there is an execution in which some process incurs \( n - 1 \) stalls while performing an instance of fetch&increment. This implies that there is an execution of \( R \) in which some instance of dequeue or pop incurs at least \( n - 1 \) stalls. In addition, each instance of dequeue or pop performs at least one event. Thus the worst case number of events and stalls incurred by a process as it performs an operation instance on \( R \) is at least \( n \).


