Linear Lower Bounds on Real-World Implementations of Concurrent Objects

Faith Ellen Fich
University of Toronto
fich@cs.toronto.edu

Danny Hendler
University of Toronto
hendler@cs.toronto.edu

Nir Shavit
Sun Microsystems Laboratories
shanir@sun.com

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. They are widely available and have been extensively researched (see [18] for a detailed survey of the literature), but 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 non-blocking linearizable 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 $\Omega(\sqrt{n})$ [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 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.

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 [1, 3, 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 time 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.

In the next two sections, we discuss related work and we present our model. Next, we define a class of objects, G, for which we obtain a linear lower bound. Then we prove the lower bound for this class. Finally, we extend the applicability of the lower bound to stacks and queues.

2 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.

Apart from simple *read* and *write* operations, mod-

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

ern 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 [17], 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, 19, 20]. 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 than 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*.

3 Model

We consider a standard model of an asynchronous deterministic 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 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). 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. A *configuration* describes the value of each object and the state of each process.

An implementation of an object that is shared by a set of n processes provides a specific data-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 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&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&increment* is a special case of *fetch&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 $h(v, w) = ack$. 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.

An *event* is a specified primitive with specified input parameters applied by a specified process to a specified base object. We say that the process *applies* the event and that 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 Op on an implemented object O. The implementation of O pro-

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

vides an algorithm for performing Op, 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 is a function of the input parameters to the operation and may also depend on events that other processes apply.

An *execution* is a (finite or infinite) sequence of events in which, starting from an initial configuration, each process applies events and changes state (based on the responses it receives) according to its algorithm. 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 after* E. If no event in E changes the value of r, then r's value after E is the initial value of r.

An *operation instance* is a specified operation with specified input parameters performed by a specified process on the implemented object. In an execution, each process performs a sequence of operation instances. Each 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 Φ has been applied in an execution E, we say that Φ *completes in* E. In this case, we call the value returned by Φ in E the *response* of Φ *in* E. We say that a process p is *active after* E if p has applied at least one event of some operation instance that is not complete in E. If p is not active after E, we say that p is *idle after* E. 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, 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 apply* e *after* E.

Linearizability is a consistency condition for concurrent objects introduced by Herlihy and Wing [15]. An execution is *linearizable* if each operation instance that completes in the execution appears to take effect atomically at some point between when its first event is applied and when it completes. If the first event of an operation instance has been applied in the execution, but the operation instance is not complete, then either it appears to take effect atomically at some point after its first event or it appears to have no effect. The resulting sequence of all complete and possibly some incomplete operation instances is called a *linearization* of the execution. An implementation is linearizable if all its executions are linearizable. In this paper, we consider only linearizable implementations.

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. An implementation satisfies *solo-termination* [9] if, after each finite execution, for each active process, there is a finite solo extension in which the process completes its operation instance. This is a very weak progress condition. Lower bounds obtained assuming only this progress condition also apply to stronger progress conditions such as wait-freedom.

An implementation is *obstruction-free* [13], if it satisfies solo termination. Any implementation that is lock-free or wait-free is also obstruction-free. Obstruction-free implementations do not use locks. This is important when one process cannot wait for other processes to take steps, for example, on systems built for multi-threaded multi-core chips, which involve extensive context switching. Although obstruction-freedom makes no progress guarantee when processes contend for an implemented object, contention managers, using probabilistic mechanisms such as backoff, enable obstruction-free implementations to achieve good progress in real-world situations.

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 formal concept of memory stalls was introduced by Dwork, Herlihy, and Waarts [8]. The following definition is stricter than theirs.

Definition 1 *Let e be an event applied by a process p as it performs an operation instance Φ 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 *Φ in E is the number of memory stalls e incurs in E, summed over all events e of Φ in E.*

The difference between this definition and the original definition by Dwork, Herlihy, and Waarts is that their definition also counts stalls caused by events applying trivial primitives such as read. Thus our lower bounds also apply to their definition of stalls.

4 The Class $\mathcal{G}$

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

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

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

- *Φ is an instance of Op by process p,*
- *no operation instance in AA' is by p or q, and*
- *each operation instance in A' is by a different process,*

there is a sequence of operation instances Q on $\mathcal{O}$ by process q such that, for 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 responses of Φ are different when $A\Phi$ and $H\Phi$ are each performed on $\mathcal{O}$ starting with its initial value.

In other words, if the operation instances in Q are performed before Φ, together with all the operation instances in AA', except for possibly the last operation instance by each process, then Φ is guaranteed to have a different response.

Many common objects are in $\mathcal{G}$. Furthermore, determining whether an object is in $\mathcal{G}$ is relatively easy. We present two examples of such proofs.

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, *fetch&increment modulo m*. The *fetch&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 the value $m-1$, in which case, it sets the value of the object to 0 (and returns $m-1$).

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

Proof: Consider a modulo-m counter with initial value 0. 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 Φ is an instance of *Op* by process p, all other operation instances are by neither p nor q, and 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 $a \bmod m$ is the response of Φ 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 and H contains between $a + a' + b - (n-2) = a+1$ and $a + a' + b = a + n - 1$ operation instances. Thus the response of Φ in $H\Phi$ must be one of the values $(a+1) \bmod m, (a+2) \bmod m, \ldots, (a+n-1) \bmod m$. Since $n \leq m$, none of these values is equal to $a \bmod m$. ■

A *counter* is an object whose set of values is the integers. It supports a single parameterless operation, *fetch&increment*, that atomically increments the value of the object to which it is applied and returns the previous value of the object. It follows from Theorem 3 that a counter object shared by any number of processes is in $\mathcal{G}$.

The value of a *single-writer snapshot* object over sets of elements $V_1, \ldots, V_n$ is a vector of n components, $C_1, \cdots, C_n$, 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: *scan* and *update*. The operation instance *update(v)* by process p_i sets the value of component C_i to v. A *scan* operation instance returns a vector consisting of the values of the n components.

Theorem 4 *A single-writer snapshot object is in $\mathcal{G}$.*

Proof: Consider a single-writer snapshot object with any initial value. Let *Op* be a *scan* operation. Let $A\Phi A'$ be a sequence of *scan* and *update* operation instances on this object such that Φ is a scan operation instance by process p and all other operation instances are by neither p nor q. Let Q be a sequence that consists of a single instance of *update* by process q which changes the value of its component to something other than 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 responses of Φ in $A\Phi$ and $H\Phi$ differ in component q. ■

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

Likewise, a queue is not in $\mathcal{G}$. Since an *enqueue* operation only returns an acknowledgement, Φ would have to be a *dequeue*. Let $A = d_1 \cdots d_i e$ consist of a sequence of i *dequeue* operation instances, where i is the number of elements initially in the queue, followed by a single *enqueue*

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

of some element v. Then Φ returns v in $A\Phi$. For any sequence of operation instances Q by process q, let ℓ denote the size of the queue after Q is performed starting with an empty queue. Let $Q = Q'Q''$, where Q'' is the shortest suffix of Q that contains ℓ *enqueue* operations. Then Φ returns v in $d_1 \cdots d_i Q' e Q'' \Phi$, since, after $d_1 \cdots d_i Q' e Q''$, the queue contains $\ell + 1$ elements, the first of which is v.

5 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 $\mathcal{G}$. 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 an operation instance by process p that incurs one stall arising from contention 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 *An execution $E\sigma_1 \cdots \sigma_i$ is a* k-stall execution 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 after E, and*
  - *in σ_j, process p applies events by itself until it is first poised to apply an 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 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 of E, no process applies a nontrivial 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 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 the 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 $\bigcup_{i=1}^{5} S_i$ are active in this configuration. In σ_1, p's operation instance will access O_1 and incur five stalls from the events of the processes in S_1. Then, in σ_2, p's operation instance will incur four additional stalls when it accesses O_2. In total, p will incur 19 stalls in the execution $E\sigma_1 \cdots \sigma_5$.

Theorem 6 *Consider any obstruction-free n-process implementation of an object $\mathcal{O}$ in class $\mathcal{G}$ 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 $0 \leq k \leq n-2$ be the largest integer for which there exists a k-stall execution for process p in which p is performing an instance Φ of the operation that witnesses the membership of $\mathcal{O}$ in $\mathcal{G}$. 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 $|S_1 \cup \cdots \cup S_i| = k$.

We will prove that there exists a $(k+k')$-stall execution for some $k' \geq 1$. To do this, we use properties of the class $\mathcal{G}$ to show there is some other process q that applies a nontrivial event to a base object that is accessed by process p in a solo extension of $E\sigma_1 \cdots \sigma_i$.

Let σ be an extension of $E\sigma_1 \cdots \sigma_i$ in which process p applies events by itself until it completes its operation instance Φ 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 σ is finite. Let v be the value returned by Φ in $E\sigma_1 \cdots \sigma_i\sigma$.

Consider a linearization $A\Phi A'$ of the operation instances performed in $E\sigma_1 \cdots \sigma_i\sigma$. Then Φ returns value v in $A\Phi$. Since all processes not in $S_1 \cup \cdots \cup S_i$ are idle after E and no operation instance begins in $E\sigma_1 \cdots \sigma_i\sigma$ after Φ's first event, A' contains at most $k \leq n-2$ operation instances, each performed by a different process.

Let q be a process not in $S_1 \cup \cdots \cup S_i \cup \{p\}$. Since the object $\mathcal{O}$ is in class $\mathcal{G}$, Definition 2 implies that there is a sequence of operation instances Q by q such that v is not returned by Φ in $H\Phi$, for every interleaving HH' of Q and the sequences of operation instances performed by each process in AA', if no operation instance by q is in H' and each operation instance in H' is by a different process.

Let τ be the solo extension of E by process q in which it completes all of the operation instances in Q. The obstruction-freedom of the implementation guarantees that τ is finite. Because $E\sigma_1 \cdots \sigma_i$ is a k-stall execution and τ is $(\{p\} \cup S_1 \cup \cdots \cup S_i)$-free, τ applies no nontrivial event

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

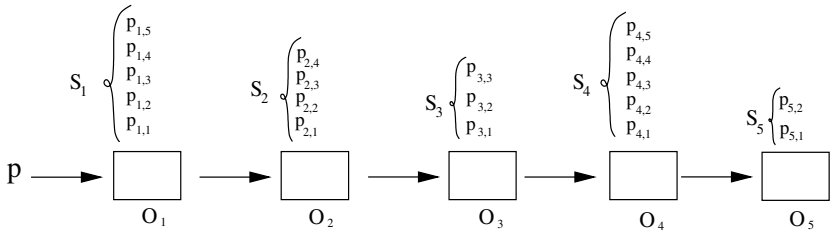

Figure 1. The configuration after the prefix E of a 19-stall execution $E\sigma_1 \cdots \sigma_5$ is executed.

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 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 after $E\sigma_1 \cdots \sigma_i$ and $E\tau\sigma_1 \cdots \sigma_i$.

Let σ' be an extension of $E\tau\sigma_1 \cdots \sigma_i$ in which p applies events by itself until it completes its operation instance Φ 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 HH' is an interleaving of Q and the sequences of operation instances performed by each process in AA'. Since all processes not in $S_1 \cup \cdots \cup S_i$ are idle after $E\tau$ and no operation instance begins in $E\tau\sigma_1 \cdots \sigma_i\sigma$ after Φ's first event, H' contains no operation instances by q, and each operation instance in H' is by a different process.

We claim that during τ, process q applies a nontrivial event to some base object accessed by p in σ. Suppose not. Then p applies exactly the same sequence of events in σ' and gets the same responses from each as it does in σ. 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 Φ in $H\Phi$, which contradicts the fact that $\mathcal{O}$ is in $\mathcal{G}$.

Let $\mathcal{F}$ be the set of all finite $(\{p\} \cup S_1 \cup \cdots \cup S_i)$-free extensions of E. Let O_{i+1} be the first base object accessed by p in σ to which some process applies a nontrivial event during event sequences in $\mathcal{F}$. O_{i+1} is well-defined since $\tau \in \mathcal{F}$ and, during τ, process q applies a nontrivial event to some base object accessed by p in σ. Since $E\sigma_1 \cdots \sigma_i$ is a k-stall execution, no event sequence 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 number of processes that are simultaneously poised to apply nontrivial events to O_{i+1} in event sequences in $\mathcal{F}$. Let E' be a minimal length event sequence in $\mathcal{F}$ such that a set S_{i+1} of k' processes are simultaneously poised to apply nontrivial events to O_{i+1} after EE'.

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 after E and hence after EE'.

Let σ_{i+1} be the prefix of σ 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 $EE'\sigma_1 \cdots \sigma_{i+1}$ is an execution. Note that p starts only one operation instance in $EE'\sigma_1 \cdots \sigma_i\sigma_{i+1}$.

If α is a $(\{p\} \cup S_1 \cup \cdots \cup S_i \cup S_{i+1})$-free extension of EE', then $E'\alpha \in \mathcal{F}$. Since $E\sigma_1 \cdots \sigma_i$ is a k-stall 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', α applies no nontrivial events to any base object accessed in σ_{i+1}.

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

6 Stacks and Queues

We do not know whether the result of Theorem 6 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 at least 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 7 *Consider any obstruction-free implementation of a counter shared by n processes, from RMW base objects. Suppose that 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 $n(n-1)^d + n$ operation instances and in which some process incurs $n - 1$ stalls while performing one instance of fetch&increment.*

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

Proof: Fix a process p and an instance Φ of *fetch&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 on $O_{r,j}$ 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 *fetch&increment*, and
- ρ_r is a solo execution by process p in which it completes Φ.

In this execution, p incurs k_r stalls. Thus it suffices to 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 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 an integral progress parameter, $\Psi_r \in [0,(n-1)^d]$, 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 Ψ_r, let π_r denote the sequence of all base objects accessed by p in the execution $E_r\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$ 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 π_r. Moreover, if $j < j'$, then $O_{r,j}$ precedes $O_{r,j'}$ in π_r. Suppose that $O_{r,j}$ occurs in position $w_r(j)$ of π_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 Ψ_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 π_r. (The only exception is when $k_r = n-1$ and all these processes are poised at the same object.) Let E_0 denote the empty execution, which contains no instances of *fetch&increment*. Let $i_0 = k_0 = 0$ and let ρ_0 denote the solo extension of E_0 in which p performs Φ 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 $k_r < n-1$. Then we will 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 exists a process $q \notin S_{r,1}\cup\cdots\cup S_{r,i_r}\cup\{p\}$.

Consider the solo extension γ of E_r by q in which γ completes n instances of *fetch&increment*. If γ applies no nontrivial event to the base objects accessed by p in $E_r\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$, then $\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$ is an extension $E_r\gamma$ 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. Thus p must receive the same response in both of these executions. Let a be the number of *fetch&increment* instances that complete in E_r. Then there are $a+n$ *fetch&increment* instances that complete in $E_r\gamma$. Since p applies its first event after $E_r\gamma$, linearizability implies that p's response in $E_r\gamma\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$ is at least $a+n$. However, there are at most $n-1$ active processes after E_r. So, by linearizability, p'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 γ 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 γ' be the shortest prefix of γ such that q is poised to perform a nontrivial event at one of these base objects 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&increment*, and γ' contains at most n instances, it follows that E_{r+1} contains at most $n(r+1)$ instances.

Suppose that, after E_{r+1}, q is poised at the object in position u of π_r. Let $i_{r+1} = 1+\#\{j \mid w_r(j) < u\}$, so $i_{r+1}-1$ is the number of objects $O_{r,j}$ that occur before position u in π_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 after E_{r+1}. There are two cases: If $O_{r+1,i_{r+1}} \in \{O_{r,1},\ldots,O_{r,i_r}\}$, 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. The other case is when $O_{r+1,i_{r+1}} \notin \{O_{r,1},\ldots,O_{r,i_r}\}$. Then we let $S_{r+1,i_{r+1}} = \{q\}$ and we 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 $O_{r+1,j}$ 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 ρ_{r+1} be the solo extension of $E_{r+1}\sigma_{r+1,1}\cdots\sigma_{r+1,i_{r+1}}$ in which p completes Φ. Obstruction-freedom guarantees the existence of ρ_{r+1}.

Note that $w_{r+1}(i_{r+1}) = u$. If $w_r(j) \leq u$, then

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

$S_{r,j} \subseteq S_{r+1,1} \cup \cdots \cup S_{r+1,i_{r+1}}$ and $w_{r+1}(j) = w_r(j)$. If $w_r(j) > u$, then $S_{r,j}$ is disjoint from $S_{r+1,1} \cup \cdots \cup S_{r+1,i_{r+1}}$. Since $|S_{r,j}| \leq n-2$ for all j, it follows that $\sum\{\, |S_{r,j}| \cdot (n-1)^{d-w_r(j)} \mid w_r(j) > u\} < (n-1)^{d-u}$. Thus $\Psi_{r+1} = \sum_{j=1}^{i_{r+1}} |S_{r+1,j}| \cdot (n-1)^{d-w_{r+1}(j)} = \sum_{j=1}^{i_r} |S_{r,j}| \cdot (n-1)^{d-w_r(j)} - \sum\{\, |S_{r,j}| \cdot (n-1)^{d-w_r(j)} \mid w_r(j) > u\} + (n-1)^{d-u} > \Psi_r$.

Hence there is an execution $E_r\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$, with $r \leq (n-1)^d$ in which p incurs $n-1$ stalls. The total number of operation instances contained in E_r is at most $n(n-1)^d$ and no process performs more than one operation instance in $\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$. Therefore the total number of operation instances contained in $E_r\sigma_{r,1}\cdots\sigma_{r,i_r}\rho_r$ is at most $n(n-1)^d + n$. ■

Theorem 8 *Consider any implementation of an obstruction free linearizable queue or stack, $\mathcal{R}$, from RMW objects, that is shared by n processes. Then, in the worst case, the total number of events applied and stalls incurred by a process as it performs an operation instance on $\mathcal{R}$ is at least n.*

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

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 7, there is an execution containing at most N operation instances, in which some process incurs $n-1$ stalls while performing an instance of *fetch&increment*. This implies that there is an execution of $\mathcal{R}$ in which some instance of dequeue or pop incurs at least $n-1$ stalls. In addition, each instance of dequeue or pop applies at least one event. Thus, in the worst case, the total number of events applied by this process plus the total number of stalls incurred by this process as it performs an operation instance on $\mathcal{R}$ is at least n. ■

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada and Sun Microsystems.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] J. Anderson and Y. Kim. An improved lower bound for the time complexity of mutual exclusion. In *ACM Symposium on Principles of Distributed Computing*, pages 90–99, 2001.

[3] J. H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[4] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[5] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, Mar. 1998.

[6] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[7] R. Cypher. The communication requirements of mutual exclusion. In *ACM Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.

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

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

[10] F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16:121–163, 2003.

[11] D. Hendler and N. Shavit. Operation-valency and the cost of coordination. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 84–91, New York, NY, USA, 2003. ACM Press.

[12] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, Jan. 1991.

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

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE

[14] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, February 1996.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. *Siam J. Comput.*, 30(2):438–456, 2000.

[17] M. Merrit and G. Taubenfeld. Knowledge in shared memory systems. In *ACM Symp. on Principles of Distributed Computing*, pages 189–200, 1991.

[18] M. Moir and N. Shavit. *Chapter 47 – Concurrent Data Structures – Handbook of Data Structures and Applications*. Chapman and Hall/CRC, first edition edition, 2004.

[19] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[20] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

Proceedings of the 2005 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)
0-7695-2468-0/05 $20.00 © 2005 IEEE