Solo-Valency and the Cost of Coordination

Danny Hendler[*] Nir Shavit[†]

November 21, 2007

Abstract

This paper introduces *solo-valency*, a variation on the valency proof technique originated by Fischer, Lynch, and Paterson. The new technique focuses on critical events that influence the responses of solo runs by individual operations, rather than on critical events that influence a protocol's single decision value. It allows us to derive $\sqrt{n}$ lower bounds on the time to perform an operation for lock-free implementations of concurrent objects such as linearizable queues, stacks, sets, hash tables, counters, approximate agreement, and more. Time is measured as the number of distinct base objects accessed and the number of stalls caused by contention in accessing memory, incurred by a process as it performs a single operation.

We introduce the *influence level* metric that quantifies the extent to which the response of a solo execution of one process can be changed by other processes. We then prove the existence of a relationship between the space complexity, latency, contention and influence level of all lock-free object implementations. Our results are broad in that they hold for implementations that may use *any* collection of read-modify-write operations in addition to read and write, and in that they apply even if base objects have unbounded size.

1 Introduction

The design of concurrent data structures for shared-memory multiprocessor machines is an important area of research. There has been extensive work on lower bounds for concurrent data structures, and the reader can find a survey in [17]. The majority of the time lower bounds obtained for concurrent data structures only count the number of *step*s performed by processes (some examples are [1, 2, 7, 14, 38]). Each step consists of local computation and an *event*, which is an application of a synchronization primitive, such as *read*, *write* or *read-modify-write*, to a base object.

However, the number of steps performed is not the only factor that contributes to the time complexity of concurrent data structures. In practice, the performance of concurrent data structures is often limited by *memory contention*, the extent to which multiple processes access widely-shared memory locations simultaneously. The degradation in performance that is caused by contention is the result of limitations on the bandwidth of both memory and processor-to-memory interconnect. Reducing memory contention has been the focus of both hardware architecture design [21, 22, 31] and the design of concurrent data structures [5, 6, 19, 20, 27, 30, 35].

[*]Department of Computer Science, Ben-Gurion University. Part of this work was done while the first author was a graduate student in the Department of Computer Science, Tel-Aviv University. This work was supported in part by a grant from Sun Microsystems.

[†]Department of Computer Science, Tel-Aviv University.

Thus, a metric for the time complexity of shared memory concurrent data structures that does not take contention into consideration may be unrealistic. As an example, consider the question of implementing a concurrent counter. If the hardware supports a *fetch&increment* synchronization primitive, then there is a straightforward n-process implementation: all processes share a single base object. To get a number from the counter, a process simply performs a *fetch&increment* operation on that object. If only steps are counted, this implementation has worst-case operation time-complexity of 1. The implementation is, however, essentially sequential: if all processes access the single shared object simultaneously, then the last process to succeed will incur a time delay linear in n while waiting for all other processes to complete their operations.

In 1993, Dwork, Herlihy and Waarts [13] introduced a formal model to capture the phenomenon of memory contention in shared memory machines. Using FLP-style valency arguments, they proved that there are inherent tradeoffs between contention and the number of steps taken by a process in concurrent data structure design. Their work was extended in several directions, most notably in the context of mutual exclusion [3, 4, 11] and counting networks [8, 9].

This paper presents *solo-valency*, a variation of the valency proof technique of Fischer et. al (FLP) [18], and uses it to continue the above work in deriving contention-aware time complexity lower bounds for concurrent objects. As surveyed by Lynch [34] and by Fich and Ruppert [17], there are numerous elegant extensions and reformulations of the FLP-style valency technique. The main difference between the solo-valency arguments we use in this work and FLP-style arguments is in the problems to which they may be applied. Whereas FLP-valency arguments are applied to *decision problems* such as *consensus*, in which the operations of all processes must return the same response, solo-valency arguments can be applied to objects such as counters, stacks and queues, where different operations are allowed, in general, to return different responses.

The time metric we use, which we call *memory steps*, counts both the number of *distinct* base objects accessed by a process and the total number of memory stalls due to *contention in writing* incurred by it, as it performs a single operation. The number of distinct base objects accessed by a process is a lower bound on the number of accesses the process does that cannot be served from its local cache. This follows from the fact that when a process accesses a base object for the first time it incurs a cache-miss.

Similarly, stalls incurred by contention in writing cause a delay that is proportional to the number of contending processes in both distributed shared memory (DSM) and cache-coherent multiprocessor machines. This is not true for contention in reading, since in shared-bus cache-coherent systems the caches of all the processes that read the same base object simultaneously can be updated in a single bus cycle.

Thus the memory steps metric is stricter than the one used by [13], as the latter counts all shared memory references and also counts memory stalls due to contention in reading. Our metric is similar to the *communication cost* metric used by Cypher [11] and to the *remote memory references* metric used by Anderson and Yang [3], and by Anderson and Kim [4], in that a single unit of both metrics corresponds to a shared memory reference that cannot be served by a local cache. For DSM systems, however, the communication cost and the remote memory references metrics are stricter than the memory steps metric, since they do not count references to a process' local segment of shared memory, whereas such references *may* be counted by our metric.

We use solo-valency arguments to derive a collection of tradeoffs and lower bound results. Specifically, we are able to show an $\Omega(\sqrt{n})$ time-complexity lower bound on lock-free implementations of objects in a class we call *Influence*(n), a wide class of concurrent objects in which an operation of one process can be simultaneously influenced by operations of $\Omega(n)$ other processes. *Influence*(n) includes objects such as linearizable queues and double-ended queues [12, 36, 37, 39, 42], stacks [41], counters [26, 27], hash tables [23, 40], sets and multi-sets [32], and approximate agreement [7]. Our results are the first known time complexity lower bounds for implementing these objects using *any* RMW operation. Before listing our results in detail, let us now describe our proof technique.

1.1 Solo-Valency

Valency arguments, introduced by Fischer et. al [18], have been used extensively to derive impossibility results and lower bounds for decision problems such as consensus (see [10, 13, 24, 33] for some examples). In the consensus problem, participating processes are required to eventually agree on a *single* protocol decision value. FLP-style valency classifies system configurations according to whether they are *univalent* or *multivalent*. A configuration is univalent if the protocol's decision value is the same in all the executions that start from the configuration and multivalent otherwise. The valency technique looks at critical events after which the system shifts from a multivalent configuration to a univalent one. Valency arguments are then applied with regard to these critical events to derive impossibility results or lower bounds.

Our technique focuses on the responses of solo executions rather than on a single protocol decision value. Assume that, after some execution E, a solo execution of an operation Op returns response v; then we say that the solo-valency of Op after E is v. The basic idea behind our solo-valency technique is to consider a class of critical events that are different from those used by FLP-style arguments. We name these events *modifying events*. A modifying event is an event by some process that atomically changes the solo-valency of an operation by another process. Thus, modifying events influence the responses of *individual operations* rather than a protocol's single decision value. Our results are obtained by arguing about the order of these events and the base objects they are about to modify.

To be more concrete, we now explain how our technique is applied in the context of a one-time counter. A *one-time* object is an object to which every process can only apply a single operation. Consider an implementation of a one-time n-process linearizable counter object allowing *fetch&increment* operations. Starting from an initial configuration, any process may start a solo execution that returns 1. In other words, the valency of a *fetch&increment* operation by any process in the initial configuration is 1.

Let E be an execution in which some process p does not participate, and assume some other process q completes its *fetch&increment* operation in E. Then the solo-valency of p's operation after E must be bigger than 1. We identify the critical modifying events - write or RMW events in E - following which the solo-valency of p's operation changes to a value bigger than 1. Our proof technique constructs executions in which such modifying events are outstanding and shows that the operations whose responses are about to be atomically modified by these events must read all the base objects on which they are outstanding, otherwise we can construct executions that violate object semantics.

An additional key difference between FLP-style valency arguments and solo-valency arguments is the following. FLP-style valency considers critical events that have *permanent* effect on a protocol's single decision value: before the critical event is executed, there exist two different execution extensions that yield two different decision values; after the critical event is executed, *all* execution extensions yield the same protocol decision value. Modifying events, on the other hand, have, in general, a *reversible* effect on the response of an operation, since they only influence the responses of operations that are performed in *solo* executions. Let us explain what we mean by 'reversible effect', by continuing the example of a one time counter. Assume that, in configuration C, process q has an outstanding modifying event, e, that will modify the solo-valency of an operation by process p. Then a solo execution of the *fetch&increment* operation by p from C returns some response v but, if e is performed after C, then a solo execution by p that immediately follows e returns a different response. The event e has a reversible effect in the sense that if e is followed by steps taken by processes other than p, then the solo-valency of p's operation may change to v again.

One may wonder why FLP-style critical events cannot be used directly for bounding the complexity of individual operations. A natural attempt to generalize FLP to this end might be to try and use critical events that have a permanent influence on the response of an operation: before the event is executed, there exist two different execution extensions in which the operation returns two different responses. After the event is executed, the operation returns the same response in all execution extensions.

This approach, however, does not seem to work. Consider again the example of an n-process implementation of a one-time counter. Let p be a process. Notice that even if the operations of up to $n-2$ other processes complete, no such critical event may have taken place with regard to p. This is because as long as there is at least one additional process whose *fetch&increment* operation did not start, the *fetch&increment* operation by p may still return different responses in different execution extensions. Hence, FLP-style critical events fail to capture the influence that these $n-2$ operations may have on the response of p's operation. This is not the case for implementations of decision problem, where an individual operation cannot return its response before the protocol's single decision value is determined.

1.2 Our Results

To characterize the coordination requirements of shared objects, we introduce the *influence level* metric $\mathbb{I}$, informally defined as the maximum number of processes whose operations can simultaneously influence the response of an operation by another process. For example, in an execution of a linearizable shared counter, the response of a given operation can be influenced by $n-1$ others: if it is performed solo from an initial configuration it will return response 1, but if an operation of any of the $n-1$ other processes precedes it, its response will be different.

1.2.1 New Fundamental Tradeoffs

We prove that the following fundamental relationships exist for all lock-free implementations. Let P be a lock-free implementation of a shared object with influence level $\mathbb{I}$. Let $\mathbb{L}(\mathrm{P})$ denote the maximum number of distinct objects accessed by a process as it performs a single operation (the maximum taken over all of P's executions). Let $\mathbb{S}(\mathrm{P})$ denote P's space complexity. Also, let $\mathbb{C}(\mathrm{P})$ denote P's *write contention*, namely the maximum number of processes that can simultaneously have outstanding write or RMW events about to access the same object (the maximum taken over all of P's executions). Then the following tradeoffs hold.

$$\mathbb{L}(\mathrm{P}) \geq \mathbb{I}/\mathbb{C}(\mathrm{P}), \qquad \mathbb{S}(\mathrm{P}) \geq \mathbb{I}/\mathbb{C}(\mathrm{P}) \tag{1}$$

For linearizable counting, this tradeoff strengthens a result of Herlihy et al. [27], which try to capture contention via a static measure of *capacity*: the maximum number of processes, $c(P)$, that access any particular base object (the maximum taken over all executions). They prove the existence of the tradeoff $\mathbb{L}(\mathrm{P}) \geq$ (n-1)/c(P) between the number of distinct base-objects accessed and the capacity of linearizable counter implementations. However, they note that high capacity does not necessarily imply high contention. Our tradeoff captures a stronger relationship between the number of distinct objects accessed and the actual write-contention for a broad class of objects. Specifically, for linearizable shared counters our tradeoffs imply the desired relationship $\mathbb{L}(\mathrm{P}) \geq$ (n-1)$/\mathbb{C}(\mathrm{P})$.

Dwork et al. [13] pose the question of whether there exists a tradeoff between contention and the number of distinct objects accessed for the approximate agreement problem. The above tradeoff answer this question in the affirmative.

1.2.2 New Time Complexity Lower Bounds

We identify *Influence*(n), a broad class of objects with influence level $\mathbb{I} \in \Omega(n)$. This class includes well-known objects such as linearizable counters, queues, stacks, hash-tables, sets, approximate agreement, and consensus. We are actually not aware of any "natural" concurrent objects that do not belong to this class. We prove a lower bound of $\Omega(\sqrt{n})$ memory steps on lock-free implementations of all objects in *Influence*(n).

All our results hold even if implementations can use any combination of read, write, and RMW synchronization primitives and without any assumptions on the size of the base objects used. These results are obtained by considering executions in which every process performs at most a single operation. Consequently they hold also for the one-time versions of all the objects in *Influence*(n).

Though our bounds seem higher than Jayanti's interesting $\Theta(\log n)$ time bounds [29] on similar objects, they are in fact orthogonal. This is because Jayanti's time metric counts only steps and does not count stalls caused by memory contention. Unlike our results, Jayanti's bounds are also restricted to implementations that can only use the write, *load-locked/store-conditional, move and swap* synchronization primitives..

Finally, we show that there exists an object in *Influence*(n), which we name *First Generation*, for which our bound is tight, that is, it has $\Theta(\sqrt{n})$ memory steps complexity. However, we believe that the tight bounds on even the one-time versions of many well-known objects in *Influence*(n), such as counters, stacks and queues, are higher.

2 Preliminaries

2.1 Shared Memory System Model

We consider a standard model of an asynchronous shared memory system in which a set of deterministic *processes*, which are sequential threads of control, communicate through deterministic shared data structures called *objects*. Each process has a set of possible *states* and each object has a *type*, that determines the set of *values* it can assume and the set of *operations* supported by it. These operations provide the only means to manipulating the object. Each such operation may receive a number of *input parameters* and returns a single value called the operation's *response*. A *configuration* describes the value of each object and the state of each process.

The system makes available a set of *primitives base objects*, which are the building blocks from which higher-level objects may be constructed. An *implementation* of an object provides a specific data-representation for the object from a set of more basic shared base objects, and algorithms for the processes to apply each operation to the implemented object. The base objects used by an implementation may be either primitive base objects, or objects that are implemented (either directly or indirectly) from primitive base objects.

The application of an operation to an object may change both the value of the object to which the operation is applied and the state of the process that applies it. We call the application of a specific operation by a specific process to a specific base object with (optional) specific input parameters an *event*. We say that an event is an *application* of an operation to a base object. We also say that an event is *issued by* a process and that the event *accesses* the base object. An event returns the response of the operation applied by it to the calling process.

We consider base objects that may support *read*, *write* and *read-modify-write* operations. Read events do not receive an input parameter. A read event e, issued by process p, that accesses a base object o atomically reads the value of o and returns that value as its response to p. A write event e with a single input parameter w, issued by process p, that accesses base object o atomically writes the value w to o and returns response *ack*. We model read-modify-write operations similarly to [15]. A read-modify-write operation applied by process p to base object o with a (possibly empty) input parameters list wl atomically updates the value of o with a new value, which is a function *g(v,wl)* of the base object's value just before the operation is applied, v, and of wl, and returns a response, $h(v, wl)$, to the calling process. We call *g* and *h* the *update function* and the *response function* of the operation, respectively.

Fetch&add is an example of a read-modify-write operation. It receives a single input parameter. Its update function is $g(v, w) = v + wl[1]$ (where $wl[i]$ denotes the i'th component of *wl*) and its response function is v. *Compare&swap* is another example of a read-modify-write operation. Informally, the *compare&swap* operation receives two parameters, *old* and *new*. If the value of the base object to which it is applied equals *old*, the operation atomically changes the object's value to *new* and returns a response of *success*. Otherwise, the value of the base object is not modified and the operation returns a response of *failure*. More formally, the update function $g(v, wl)$ of the *compare-and-swap* operation atomically does the following: if $v \neq wl[1]$ (namely the base object's value does not equal *old*), then $g(v, wl)$ returns v, implying that the value of the base object to which the operation is applied is not modified. Otherwise, $g(v, wl)$ returns $wl[2]$, implying that the value of the base object is modified to *new*. The response function of the *compare-and-swap* operation, $h(v, wl)$, returns *success* if $v = wl[1]$ and *failure* otherwise.

An *execution-fragment* is a (finite or infinite) sequence of events in which each process issues events and changes state (based on the responses it receives from these events) according to the algorithm specified by the implementation. An *execution* is an execution-fragment that starts from an *initial configuration*, in which all base objects have their initial values and all processes have their initial states. Any prefix of an execution is also an execution. We let $\prec$ denote the prefix relation between executions. For any finite execution E and sequence of events E' we let EE' denote the concatenation of the events in E and E'. EE' may or may not be an execution. We denote by $E|p$ the subsequence of E that includes all the events of E that were issued by process p. If $E|p$ is the empty sequence we say that E is *p-free*. A *solo execution fragment* is an execution fragment all of whose events are issued by a single process. We say that an event e is *outstanding* after a finite execution E if Ee is an execution. Two executions are *indistinguishable* to a process p, if p issues the same sequence of events and gets the same responses from these events in both executions and the values of all base objects are the same after these executions. Even though two events that appear in an execution may have the same input parameters (i.e. they are applications by the same process of the same operation with the same parameters to the same base object), we assume that all events that comprise an execution are unique. That is, an event in an execution may be thought of as tagged by its ordinal place in the execution.

To avoid confusion between operations that are applied to base objects and operations that are applied to an implemented object, we call the latter *high-level operations*. We consider executions in which every process *performs* at most a single high-level operation.[1] If an execution E is not p-free, we denote by $\Phi(E, p)$ the single high-level operation performed by p in E. We say that $\Phi(E, p)$ *occurs* in E. We say that $\Phi(E, p)$ *completes* in E if the last event of $\Phi(E, p)$ has been issued in E. In this case, we call the value returned by $\Phi(E, p)$ the *response* of Φ in E. If p applies a read or read-modify-write operation to a base object o while performing $\Phi(E, p)$, we say that $\Phi(E, p)$ *reads o*. We denote by $\mathcal{R}(E, p)$ the number of distinct base objects that p reads in E. Let $\Phi(E, p)$ and $\Phi(E, q)$ be two high-level operations that complete in an execution E. We say that $\Phi(E, p)$ *precedes* $\Phi(E, q)$ in E if all the events of $E|p$ precede all the events of $E|q$.

We say that an execution is *quiescent* if all the operations that occur in E complete in E. A *history* is a sequence of high-level operations that are applied to an object and their responses are consistent with the semantics of the object. For a history H of an object and a sequence of high-level operations on the object,

[1]Clearly, this assumption can only strengthen our tradeoffs and lower bounds.

H', we let HH' denote the concatenation of H and H'. The sequence HH' may or may not be a history of the object. The *sequential specification* of an object is the set that contains all of the object histories.

We say that p has an *enabled operation* Φ after E, if p has an outstanding event e after E that p is about to issue as it performs the algorithm implementing the high-level operation Φ. We also say that e is an *outstanding event of* Φ. In other words, p has an enabled operation Φ after E, if the next event that p will issue after E is an event p issues while executing Φ.

We only consider executions in which every process performs at most a single predetermined operation, as this suffices to obtain our results. Hence, if p has not completed its operation in E, then p has a single enabled operation after E.

The safety property that is required from the implementations we consider in this paper is *linearizability* [28]. An execution is *linearizable* if every high-level operation that completes in it appears to occur atomically at some point between when its first event is issued and when it completes. The liveness requirement from the implementations we consider is lock-freedom [25]. An implementation is *lock-free*, if it guarantees that whenever a process issues some finite number of events, some process completes a high-level operation.

2.2 Memory Steps

Our time complexity metric counts the worst-case number of *memory steps* that a single high-level operation may incur. Our metric counts both the number of distinct base objects accessed and the number of *stalls* that are incurred when multiple processes concurrently attempt to apply write or read-modify-write operations to the same base object. Formal definitions follow.

Definition 2.1 *Let E be an execution and let $e_1, \ldots, e_l$, for some $l \geq 1$, be a maximal sequence of consecutive write and RMW events in E that are issued by distinct processes and access the same base object. Let p be the process that issues event e_j, for some $j \in \{1, \cdots, l\}$. We say that $\Phi(E,p)$ incurs $j-1$ stalls in E on account of e_j. Let e be an event of E issued by process p. We denote by $stalls(E,e)$ the number of stalls incurred by $\Phi(E,p)$ in E on account of e.*

The above definition of stalls captures the fact that in shared memory multiprocessors, when a group of processes have outstanding write and read-modify-write events all about to access the same base object, these events can be "released" simultaneously, thus causing the process that issued the second event to incur a single stall, the process that issued the third event to incur two stalls, and so on. [2]

Definition 2.2 *Let E be an execution and let p be a process that issues at least one event in E. The* memory steps complexity *of $\Phi(E,p)$, denoted $mem\_steps(E,p)$, is defined as follows.*

$$mem\_steps(E,p) = \mathcal{R}(E,p) + \sum_{e \in E|p} stalls(e)$$

The memory steps complexity of an implementation *is the maximum over the memory steps complexity of all the high-level operations that occur in all of the implementation's executions.*

[2] This definition of stalls does *not* assume that concurrently outstanding write and read-modify-write events about to access the same base object are serviced in a first-come-first-served order, or in any other specific order.

3 The *Influence* Metric for Coordination Level

In this section we define a metric which is a measure of the coordination level of concurrent object implementations. More specifically, the *influence level* metric is a measure of the extent to which concurrently executing high-level operations can influence the response of a high-level operation performed by another process. To get a feel for this metric, consider an n-process implementation of a linearizable stack that supports the $push$ and pop operations. Consider a quiescent execution, E, in which the high-level operations performed by processes $p_1 \ldots p_i$, for some $i \leq n-2$, were completed. Assume that after E the stack contains a single item - the number 1. Assume also that after E each of the processes $p_{i+1} \ldots p_{n-1}$ has an enabled *push* operation with input 2 and process p_n has an enabled pop operation.

Clearly, the response of the pop operation by p_n can be influenced by the $n-i-1$ $push$ operations: if p_n performs the pop operation solo after E then, from linearizability, it has to return 1. On the other hand, let E' be a p_n-free execution-fragment that starts after E such that some *push* operation by a process not in $\{p_1, \ldots, p_i\}$ completes in EE'. Then - again from linearizability - a solo execution of the *pop* operation by p_n that starts after EE' returns response 2 or higher. Thus the number of concurrently enabled high-level operations by different processes that can change the response of p_n's *pop* operation after E is $n-i-1$. Dependencies of this type are what we capture in the following definitions and lemmata.

Definition 3.1 *Let E be an execution and let R be a high-level operation by process p that is enabled after E. We define the* solo-valency of R after E *to be the response returned by R in the solo execution by p that starts immediately after E and ends when p completes R. We let $\mathcal{S}(E,R)$ denote this value. If a solo execution by p that starts immediately after E does not terminate, we say that the solo-valency of R after E is* undefined *and write $\mathcal{S}(E,R) = \perp$.*

When confining attention to lock-free algorithms, a solo execution of a high-level operation always terminates. We get the following.

Fact 3.1 *Let E be an execution of a lock-free algorithm and let R be a high-level operation that is enabled after E, then $\mathcal{S}(E,R) \neq \perp$ holds.*

Definition 3.2 *Let E be an execution and let R be a high-level operation by process p that is enabled after E. We say that R has* influence level K after E *(and write $\mathbb{IA}(E,R) = K$) if K is the maximum such that there exist K high-level operations, $W_1 \cdots W_K$, that are enabled after E by distinct processes other than p, each of which has an enabled event e_i such that $\mathcal{S}(Ee_i,R) \neq \mathcal{S}(E,R)$ holds. We call the events e_i* modifying events for R. *We define the* influence level after E, *denoted by $\mathbb{IA}(E)$, as the maximum influence level over all the high-level operations that are enabled after E. Formally:*

$$\mathbb{IA}(E) = \max\left\{\mathbb{IA}(E,Op) | \text{Op is enabled after E}\right\}$$

It is easily seen that the influence level after any execution of an n-process implementation is always between 0 and $n-1$.

Next, we extend the definition of influence level to concurrent object implementations.

Definition 3.3 *The* influence level during *a finite execution E, denoted by $\mathbb{I}(E)$, is the maximum influence level after all the prefixes of E. Formally:*

$$\mathbb{I}(E) = \max\left\{\mathbb{IA}(E') | E' \prec E\right\}$$

The influence level *of an implementation P, denoted by $\mathbb{I}(P)$, is the maximum influence level during all the executions that result when processes perform high-level operations using the algorithms of the implementation.*

Slightly abusing notation, we now define the influence level of concurrent objects.

Definition 3.4 *A concurrent object O has* influence level $\mathbb{I}$, *if the influence level of every lock-free implementation of O is at least $\mathbb{I}$.*

Based on the above definitions, the next two lemmas show that we can determine a lower bound on the influence level of lock-free linearizable object implementations based on the object's sequential specification.

Lemma 3.2 *Let P be lock-free implementation that has an execution E such that*

- *there is a process that has a high-level operation R enabled after E with $\mathcal{S}(E, R) = v$, and*
- *there are K other processes, $p_{i_1} \dots p_{i_K}$, each of which has a high-level operation W_j, for $j \in \{1 \dots K\}$, that is enabled after E so that for any execution EE', where E' consists of events issued by $p_{i_1} \dots p_{i_K}$ and at least one of $W_1 \dots W_K$ completes in EE', $\mathcal{S}(E, R) \neq v$ holds.*

Then P has influence level K or more.

Proof

We iteratively construct an execution after which the influence level is at least K as follows. We let E_0 denote the empty execution. In iteration i, for $i \geq 1$, we pick some high-level operation W_j that has an outstanding event e after E_{i-1} that is not a modifying event for R, if such an operation exists. The following cases exist.

1. Every process in $\{p_{i_1} \dots p_{i_K}\}$ has an outstanding modifying event for R after E_{i-1}. From Definition 3.2, the influence level after E is at least K. The lemma now follows from Definition 3.3.
2. There exists a process p_{i_j}, for some $j \in \{1, \dots K\}$, that has an outstanding event after E_{i-1} that is not a modifying event for R. We let $E_i = E_{i-1}e$.

From assumptions, none of the operations W_j can complete before a modifying event is issued by a process in $\{p_{i_1} \dots p_{i_K}\}$. However, as P is lock-free, one of these operations must complete after a finite number of events is issued. Thus there exists a finite l so that the conditions of case (1) above hold after E_l. The lemma follows. Q.E.D.

Lemma 3.3 *Let O be an object and let H be a history of O such that*

- *there exists a high-level operation R such that* HR *is a history of O and the response of R in* HR *is v, and*
- *there exist high-level operations $W_1 \dots W_K$ such that for any non-empty subset of indices $T \subseteq \{1, \cdots, K\}$ and every permutation σ_T of T, $H_{\sigma_T} = HW_{\sigma_T(1)} \cdots W_{\sigma_T(|T|)}R$ is a history of O and the response of R in H_{σ_T} is not v.*

Then any linearizable implementation of O has influence level K or more.

Proof Immediate from Lemma 3.2 and from the linearizability of the implementation. Q.E.D.

4 Tradeoffs and Lower Bounds

In the proofs provided in this section, we consider a lock-free implementation P, shared by $n > 1$ processes, that has influence level K.

We let $\mathbb{L}(\mathrm{P})$ (respectively $\mathbb{L}_{\mathbb{R}}(\mathrm{P})$) denote the maximum number of distinct base objects accessed (respectively, read) by a process as it performs a single high-level operation, the maximum taken over all of P's executions. We let $\mathbb{S}(\mathrm{P})$ denote P's *space complexity*, namely the number of distinct base objects that are accessed by all the events issued in all of P's executions. We let $\mathbb{C}(\mathrm{P})$ denote P's *write contention*, namely the maximum number of concurrently outstanding write or RMW events about to access the same base object, the maximum taken over all of P's executions.

The following lemma proves that modifying events cannot be read events.

Lemma 4.1 *All modifying events are either write or read-modify-write events.*

Proof Let e be a modifying event for a high-level operation R after an execution E. Also, let p be the process executing R. To obtain a contradiction, assume that e is a read event.

Since e does not change the value of the base object which it accesses and since e is not an event issued by p, executions E and Ee are indistinguishable to p. It follows that $\mathcal{S}(E,R)$ $=\mathcal{S}(Ee,R)$ must hold. However, from Definition 3.2, this implies that e is not a modifying event, a contradiction. Q.E.D.

We now prove a tradeoff between the space complexity and write contention of lock-free implementations.

Theorem 4.2 *Let O be an object with influence level $\mathbb{I}$ and let P be a lock-free implementation of O, then the following holds:*

$$\mathbb{S}(\mathrm{P}) \geq \lceil \mathbb{I}/\mathbb{C}(\mathrm{P}) \rceil$$

Proof Since P implements O, from Definition 3.4 we have $\mathbb{I}(P) \geq \mathbb{I}$. Consequently, from Lemma 4.1 and Definition 3.3, P has an execution after which there are at least $\mathbb{I}$ write or RMW outstanding events. The claim now follows from the fact that at most $\mathbb{C}(P)$ such events can be simultaneously outstanding on any single base object. Q.E.D.

We next prove a tradeoff between the number of distinct base objects read by a process as it performs a single high-level operation and between the write-contention of the implementation.

Theorem 4.3 *Let O be an object with influence level $\mathbb{I}$ and let P be a lock-free implementation of O, then the following holds:*

$$\mathbb{L}(\mathrm{P}) \geq \mathbb{L}_{\mathbb{R}}(\mathrm{P}) \geq \lceil \mathbb{I}/\mathbb{C}(\mathrm{P}) \rceil$$

Proof The left-hand inequality is obvious from the definitions of $\mathbb{L}(\mathrm{P})$ and $\mathbb{L}_{\mathbb{R}}(\mathrm{P})$. We now prove the right-hand inequality. Since P implements O, from Definition 3.4, we have $\mathbb{I}(P) \geq \mathbb{I}$. From Definition 3.3, there is an execution E of P such that the influence level after E is at least $\mathbb{I}$. Let R be a high-level operation that has influence level $\mathbb{I}$ or more after E, let p be the process that performs R and let $\mathcal{S}(E,R) = v$. Also let $e_1 \dots e_{\mathbb{I}}$ be $\mathbb{I}$ outstanding modifying events for R after E and B be the set of base objects that are accessed by $e_1 \dots e_{\mathbb{I}}$. Clearly, $|B| \geq \lceil \mathbb{I}/\mathbb{C}(\mathrm{P}) \rceil$.

Let E' be the execution fragment that results when we let p run solo as it performs R after E. From Definition 3.1, the response of R in EE' is v. We prove the theorem by showing that E' must include read or read-modify-write events that access each of the base objects of B. Assume otherwise to obtain a contradiction. Then there is an object $o \in B$ so that no read or read-modify-write event in E' accesses o. Let e be a modifying event for R that is outstanding after E and accesses o. As R does not read o, $EE'|p$

and $EeE'|p$ are the same. Consequently $S(EE', R) = S(EeE', R)$ must hold. This is a contradiction to our assumption that e is a modifying event. Q.E.D.

Based on Theorem 4.3, we can now establish a lower bound on the memory steps complexity of concurrent objects, as a function of their influence level.

Theorem 4.4 *Let O be an object with influence level $\mathbb{I}$ and let P be a lock-free implementation of O. Then the memory steps complexity of P is at least $\lfloor\sqrt{\mathbb{I}}\rfloor$.*

Proof Consider the write-contention of P. The following two possibilities exist.

- $\mathbb{C}(\mathrm{P}) \geq \lfloor\sqrt{\mathbb{I}}\rfloor$. In this case, from the definition of write contention, there is an execution E of P after which there are $\lfloor\sqrt{\mathbb{I}}\rfloor$ outstanding write and read-modify-write events, all of which about to access the same base object. Let E' be some ordering of these events, let e be the last event of E', let p be the process that issues e, and let o be the base object accessed by e. From Definitions 2.1 and 2.2, $\Phi(EE', p)$ incurs $\lfloor\sqrt{\mathbb{I}}\rfloor - 1$ memory steps on account of the stalls incurred by e plus an additional memory step on account of the access of o.

- Otherwise $\mathbb{C}(\mathrm{P}) < \lfloor\sqrt{\mathbb{I}}\rfloor$. From Theorem 4.3 we get that $\mathbb{L}_{\mathbb{R}}(\mathrm{P}) \geq \lceil\sqrt{\mathbb{I}}\rceil$. Hence there is an execution E and a process p such that $\mathcal{R}(E, p) \geq \lceil\sqrt{\mathbb{I}}\rceil$ and the claim follows.

Q.E.D.

4.1 The *Influence(n)* Objects Class

We now define the *Influence(n)* class of concurrent objects, that contains objects for which every lock-free n-process implementation has influence level in $\Omega(n)$. We then show that many well-known concurrent objects belong to this class and thus have worst-case complexity of $\Omega(\sqrt{n})$ memory steps. We conclude this section by presenting the *First Generation* object. We prove that this object belongs to the *Influence(n)* class and that it has an $O(\sqrt{n})$ memory steps lock-free implementation. This proves that the $\Omega(\sqrt{n})$ bound for the *Influence(n)* class is tight. [3]

Definition 4.1 A generic object *O is an object that is specified for any number of processes. The* influence function *of O, denoted $\mathbb{I}_O$, is defined as follows: $\mathbb{I}_O(n) = K$, if the influence level of every lock-free n-process implementation of O is at least K.*

Definition 4.2 Influence(n) *is the objects class that contains all generic objects O such that $\mathbb{I}_O$ is in $\Omega(n)$.*

Influence(n) is a broad class and it is easily shown that the following objects are in it: linearizable counters, stacks, queues and double-ended queues, hash-tables, sets, multi-sets, and approximate agreement. Moreover, we are not aware of any "natural" shared objects that do not belong to *Influence(n)*. In the following we prove that linearizable counters, queues, and approximate agreement objects belong to *Influence(n)*.

A concurrent counter object assumes values from the set N^+ of the positive integers. It supports a single operation called *fetch&increment*. The responses of *fetch&increment* operations are required to be unique natural numbers. It is also required that the responses of all the operations performed in any non-empty quiescent execution constitute a contiguous range of natural numbers starting from 1. *Linearizable*

[3]This does not imply, however, that the bound is tight for *all* the objects in *Influence(n)*.

concurrent counters are also required to be linearizable, i.e. if $\Phi(E,p)$ and $\Phi(E,q)$ are two *fetch&increment* operations that complete in an execution E and $\Phi(E,p)$ precedes $\Phi(E,q)$ then the response of $\Phi_(E,p)$ is smaller than that of $\Phi_(E,q)$.

Lemma 4.5 *The linearizable concurrent counter object is in* Influence(n).

Proof We prove that any lock-free linearizable n-process counter implementation has influence level $n-1$. In the following we denote by Φ_i, for $i \in \{1,\ldots,n\}$, a *fetch&increment* operation performed by process p_i. The sequential specification of the linearizable concurrent counter object contains a history Φ_1 where the response of Φ_1 is 1. Additionally, for any non-empty $T \subseteq \{2\cdots n\}$, and for any permutation σ_T of T, the sequential specification contains the following history: $H_{\sigma_T} = \Phi_{\sigma_1}\cdots\Phi_{\sigma_{|T|}}\Phi_1$, and the response of Φ_1 in H_{σ_T} is bigger than 1. Hence the conditions of Lemma 3.3 hold and the claim follows. Q.E.D.

A concurrent queue object supports two operations: *enqueue* and *dequeue*. Each *enqueue* operation receives input v from a non-empty set of values *V*. Each *dequeue* operation applied to a non-empty queue returns a value $v \in V$. The state of a queue is a sequence of items $S = \langle v_0,\cdots,v_k\rangle$, each of which is a value from *V*. A concurrent queue implementation is required to be linearizable to a standard sequential queue, whose operations are specified as follows.

- $enqueue(v_{new})$ changes S to be the sequence $S = \langle v_0,\cdots,v_k,v_{new}\rangle$.
- if S is not empty, a *dequeue* operation changes S to be the sequence $S = \langle v_1,\cdots,v_k\rangle$ and returns v_0. If S is empty, *dequeue* returns the special value *empty*.

Lemma 4.6 *The queue object is in* Influence(n).

Proof We prove that any lock-free n-process queue implementation has influence level $n-1$. In what follows we denote by by Deq_1 a dequeue operation by p_1 and by Enq_i, for $i \in \{2,\ldots,n\}$, an enqueue operation by process p_i that enqueues some value $v \in V$.

The sequential specification of a queue contains a history Deq_1 in which the response of Deq_1 is *empty*. Additionally, for any non-empty $T \subseteq \{2\cdots n\}$, and for any permutation σ_T of T, the sequential specification contains the following history: $H_{\sigma_T} = Enq_{\sigma_1}\cdots Enq_{\sigma_{|T|}}Deq_1$, and the response of Deq_1 in H_{σ_T} is not *empty*. Hence the conditions of Lemma 3.3 hold and the claim follows. Q.E.D.

An approximate agreement object supports a single operation called *decide* that every process can perform at most once. The *decide* operation receives a single real number as its input. An implementation of approximate agreement is correct if the following two conditions hold for any execution E.

Agreement : the responses of any two *decide* operations must differ by at most ϵ.

Validity : the response of any *decide* operation must be within the range of the inputs of all the *decide* operations that occur in the execution.

Lemma 4.7 *The approximate-agreement object is in* Influence(n).

Proof We prove that any lock-free n-process approximate-agreement implementation has influence level $n-1$. Consider the execution in which the input of the *decide* operation performed by process p_i, for $i \in \{1,\cdots n\}$, is $2(i-1)\epsilon$. We now show that the empty execution meets the conditions of Lemma 3.2. From the validity requirement, if p_1 runs solo as it performs its *decide* operation starting from an initial configuration it must get a response of 0. On the other hand, consider a p_1-free execution, E, in which a *decide* operation by a process other than p_1, Φ, completes. From the validity requirement, the response of

Φ in E is in the range $[2\epsilon, 2(n-1)\epsilon]$. Let E' be an execution fragment that starts after E, in which p_1 runs solo as it performs its *decide* operation. Then from the agreement requirement, the response of p_1's *decide* operation in EE' is at least ϵ. Consequently, from Lemma 3.2, the influence level of any n-process lock-free implementation of approximate-agreement is $n-1$. Q.E.D.

The proofs that linearizable stack, hash-table, double-ended queue, set and multi-set are in *Influence(n)* are almost identical to the proof of Lemma 4.6, and so we just state the result.

Lemma 4.8 *The linearizable stack, hash-table, double-ended queue, set and multi-set objects are in* Influence(n).

We next present the *First Generation* object. We prove that it belongs to *Influence(n)* and that it can be implemented in $\Theta(\sqrt{n})$ memory steps. Hence there are objects in *Influence(n)* for which our bound is tight.

Let E be an execution and $\Phi(E,p)$ be a high-level operation that occurs in E. We say that $\Phi(E,p)$ *is in the first generation of* E if it is not preceded by any other high-level operation in E.

Definition 4.3 *A* First Generation *object supports a single operation called* First*. A process can perform the* First *operation at most once. The operation returns a boolean value. Any correct implementation must meet the following requirements for every non-empty execution* E*:*

- ***R1*** *: The response of a* First *operation that completes in* E *and is not in the first generation of* E *is* false.
- ***R2*** *: If all the* First *operations that are in the first generation of* E *complete in* E *then the response of at least one of them is* true.

Lemma 4.9 *The* First Generation *object is in* Influence(n).

Proof Let E be a solo execution in which process p_1 completes its *First* operation. From requirement **R2**, the response of p_1's operation in E is *true*. Let E' be a p_1-free execution in which at least one of the other $n-1$ processes completes its *First* operation and let E'' be a solo execution fragment that starts after E', in which p_1 performs its *First* operation to completion. From requirement **R1**, the response of p_1's operation in $E'E''$ is *false*. The claim follows from Lemma 3.2. Q.E.D.

We now present a simple $\Theta(\sqrt{n})$ memory steps lock-free n-process implementation of a *First Generation* object. The implementation uses an array of $\lceil\sqrt{n}\rceil$ multi-reader multi-writer atomic registers named *mark*. The entries of the *mark* array are initialized to *false*. The code implementing the *First* operation is shown in Figure 1. The unique identifier of each process that shares the implementation is stored in a local register called *myId*.

```
boolean First()
   {
    for (k=0; k< (sizeof mark); k++)
      if (mark(k) == true)
         return false;
   mark[myId /sqrt(n)] = true;
   return true;
   }
```

Figure 1: First Operation Code

In the following we prove that the code shown in Figure 1 is a correct implementation of the *First Generation* object.

Lemma 4.10 *Let E be a non-empty execution of the implementation shown in Figure 1, such that all the* First *operations in the first generation of E complete in E. Then at least one of these operations returns response* true *in line 5.*

Proof Let Φ be the first operation in the first generation of E to exit the loop of lines 1-3. Then Φ returns response *true* in line 4. Assume otherwise to obtain a contradiction, then Φ returns response *false* in line 3. This implies that another *First* operation, Φ', performed by another process, has set its entry in the *mark* array before Φ has read this entry in line 2. Hence Φ' exits the loop of lines 1-3 before Φ. This is a contradiction. Q.E.D.

Lemma 4.11 *Let E be a non-empty execution and let Φ be a* First *operation that is not in the first generation of E. Then Φ does not return response* true *in E.*

Proof As Φ is not in the first generation of E, there is another operation Φ' that precedes Φ in E. The following two possibilities exist.

- The response of Φ' is *true*. Hence Φ' sets the *mark* flag in line 4 before Φ starts. Consequently Φ reads *true* from that flag and returns response *false* in line 3.
- The response of Φ' is *false*. Hence Φ' reads *true* from at least one element of the *mark* array; as Φ' precedes Φ so does Φ. Hence Φ returns response *false* in line 3.

Q.E.D.

Lemma 4.12 *The code shown in Figure 1 is a correct implementation of the* First Generation *object and has memory steps complexity of $\Theta(\sqrt{n})$.*

Proof Correctness stems from Lemmas 4.10, 4.11. As for the memory steps complexity, we need to sum-up the worst case number of distinct registers accessed and stalls incurred by a single *First* operation. The write contention of the implementation of Figure 1 is at most $\lfloor\sqrt{n}\rfloor$. As every operation issues at most a *single* write event (in line 4), an operation incurs at most $\lfloor\sqrt{n}\lfloor-1$ stalls. Also, an operation performs at most $\lfloor\sqrt{n}\rfloor$ iterations of the loop in lines 1-3. Consequently no operation accesses more than $\lfloor\sqrt{n}\lfloor$ distinct registers. Q.E.D.

4.2 Memory Steps and Time

In most modern shared memory multiprocessors, different types of memory accesses result in different time delays. If processors have local caches, then an object whose most recent value is stored in a processor's local cache can be accessed an order of magnitude faster than an object whose value is not in the local cache. This is because the values of objects that are not in a processor's local cache need to be sent over the relatively slow memory-to-processor interconnect. We call an access of an object whose most recent value is present (respectively not present) in the local cache of the process that makes the access a *local access* (respectively a *non-local access*). We now discuss how the $\Omega(\sqrt{n})$ memory steps lower bound obtained in Section 4 for the *Influence(n)* class translates to a time lower bound. We show that if a multiprocessor does not support multiple outstanding read/read-modify-write events per processor, our $\Omega(\sqrt{n})$ lower bound implies that, for objects in *Influence(n),* the worst-case time complexity of performing a single operation is

$\Omega(\sqrt{n})$ times the time it takes to make a non-local access. If the multiprocessor *does* support multiple outstanding read/read-modify-write events per processor, then our lower bound implies a worst-case time complexity of $\Omega(\sqrt{n})$ local accesses.

In the proof of Theorem 4.4 we have shown that either an operation incurs at least $\lfloor\sqrt{\mathbb{I}}\rfloor - 1$ *consecutive* stalls or a process reads at least $\lfloor\sqrt{\mathbb{I}}\rfloor$ distinct base objects as it performs a single operation, where $\mathbb{I}$ is the influence level of the object to which the operation is applied. We now analyze both cases.

- In any shared-memory multiprocessor, when multiple processors attempt to apply write or read-modify-write events to the same memory location simultaneously, these events are being serialized and are serviced by the memory controller one after the other. Even on cache-coherent multiprocessors, x consecutive stalls imply a delay of $x - 1$ non-local accesses: if the cache scheme is *write-through*, then every write or read-modify-write access generates a cache miss. If the cache scheme is *write-back* then, as the events are issued by different processors, none of them (except for possibly the first) can be handled by just updating the local cache: they have to either invalidate or update the local caches of other processors.

- In all shared-memory multiprocessors, the *first* read or read-modify-write access of a memory location made by a process cannot be resolved from the local cache of the processor that makes the access. Consequently, if the multiprocessor does not support non-blocking reads or read-modify-writes, then the delay incurred by an operation that accesses x distinct base objects is at least the time it takes to make a non-local access, multiplied by x. If the multiprocessor *does* support multiple outstanding read or read-modify-write events per processor, then, theoretically, the time to access x distinct base objects may equal the time it takes to make a local access, multiplied by x.

5 Discussion and Further Research

This paper introduces solo-valency, a variation of the FLP valency arguments that can be applied to implementations of concurrent objects such as counters, stacks and queues. It also introduces the influence level metric for quantifying the extent to which the response of a high-level operation by one process can be influenced by high-level operations performed by other processes. By using the influence level metric and applying solo-valency arguments we obtain $\Omega(\sqrt{n})$ time lower bounds for n-process lock-free implementations of a broad class of objects that we call *Influence*(n). The time metric we use, which we name *memory steps,* is contention-aware: it counts the number of stalls caused by contention in writing, in addition to the number of distinct base objects accessed by a high-level operation.

We prove an $\Omega(\sqrt{n})$ lower bound on the memory-steps complexity of the one-time versions of all the objects in the *Influence*(n) class. We also show that the bound is tight for at least one object in it.

In a recent paper by Fich, Hendler, and Shavit [16], a lower bound of $n - 1$ stalls is proved for a class of objects that includes counter and snapshot objects. They also prove a lower bound of $n - 1$ memory steps for stacks and queues. These results do not hold, however, for the one-time versions of these objects. Finding the tight time complexity for the one-time versions of these objects remains an interesting open problem. Another interesting open question is that of finding a non-trivial lower bound on the *average* number of memory steps incurred by high-level operations applied to these objects.

6 Acknowledgements

We would like to thank Faith Ellen Fich and Ori Shalev for helpful comments. Comments of the anonymous referees were also very helpful.

References

[1] Afek, Y., and Stupp, G. Optimal time-space tradeoff for shared memory leader election. *J. Algorithms 25*, 1 (1997), 95–117.

[2] Alur, R., Attiya, H., and Taubenfeld, G. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing 26*, 2 (1997), 539–556.

[3] Anderson, and Yang. Time/contention trade-offs for multiprocessor synchronization. *INFCTRL: Information and Computation (formerly Information and Control) 124* (1996).

[4] Anderson, J., and Kim, Y. An improved lower bound for the time complexity of mutual exclusion, 2001.

[5] Anderson, T. E. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *ICPP (2)* (1989), pp. 170–174.

[6] Aspnes, J., Herlihy, M., and Shavit, N. Counting networks. *J. ACM 41*, 5 (1994), 1020–1048.

[7] Attiya, H., Lynch, N., and Shavit, N. Are wait-free algorithms fast? *Journal of the ACM 41*, 4 (July 1994), 725–763.

[8] Busch, C., Hardavellas, N., and Mavronicolas, M. Contention in counting networks. In *Symposium on Principles of Distributed Computing* (1994), p. 404.

[9] Busch, C., and Mavronicolas, M. An efficient counting network. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)* (1998), pp. 380–385.

[10] Chor, B., Israeli, A., and Li, M. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)* (New York, NY, 1987), ACM Press, pp. 86–97.

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

[12] Detlefs, D., Flood, C. H., Garthwaite, A., Martin, P., Shavit, N., and Jr., G. L. S. Even better DCAS-based concurrent deques. In *International Symposium on Distributed Computing* (2000), pp. 59–73.

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

[14] Fatourou, P., Fich, F., and Ruppert, E. Space-optimal multi-writer snapshot objects are slow. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing* (New York, NY, USA, 2002), ACM Press, pp. 13–20.

[15] FICH, F., HENDLER, D., AND SHAVIT, N. On the inherent weakness of conditional synchronization primitives. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2004), ACM Press, pp. 80–87.

[16] FICH, F. E., HENDLER, D., AND SHAVIT, N. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science* (October 2005), pp. 165–173.

[17] FICH, F. E., AND RUPPERT, E. Hundreds of impossibility results for distributed computing. *Distributed Computing 16*, 2-3 (2003), 121–163.

[18] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (April 1985), 374–382.

[19] GAWLICK, D. Processing ”hot spots” in high performance systems. In *CompCon'85* (1985), pp. 249–251.

[20] GOODMAN, J., VERNON, M., AND WOEST, P. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *3rd Internation Conference on Architectural Support for Programming Languages and Operating Systems* (1989), pp. 64–75.

[21] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The nyu ultracomputer — designing an mimd shared memory parallel computer. *IEEE Transactions on Computers 32*, 2 (Feb. 1983), 725–763.

[22] GOTTLIEB, A., LUBACHEVSKY, B. D., AND RUDOLPH, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems 5*, 2 (Apr. 1983), 164–189.

[23] GREENWALD, M. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing* (2002), pp. 260–269.

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

[25] HERLIHY, M. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems 15*, 5 (Nov. 1993), 745–770.

[26] HERLIHY, M., LIM, B.-H., AND SHAVIT, N. Scalable concurrent counting. *ACM Transactions on Computer Systems 13*, 4 (1995), 343–364.

[27] HERLIHY, M., SHAVIT, N., AND WAARTS., O. Linearizable counting networks. *Distributed Computing 9*, 4 (April 1996), 193–203.

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

[29] JAYANTI, P. A time complexity lower bound for randomized implementations of some shared objects. In *Symposium on Principles of Distributed Computing* (1998), pp. 201–210.

[30] KLUGERMAN, M. R. Small-depth counting networks and related topics. Tech. Rep. MIT/LCS/TR-643, 1994.

[31] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems 10*, 4 (1988), 579–601.

[32] LANIN, V., AND SHASHA, D. Concurrent set manipulation without locking. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (1988), pp. 211–220.

[33] LOUI, M. C., AND ABU-AMARA., H. H. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research 4* (1987), 163–183.

[34] LYNCH, N. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)* (New York, NY, 1989), ACM Press, pp. 1–28.

[35] MELLOR-CRUMMEY, J., AND SCOTT, M. Synchronization without contention. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1991), ACM Press, pp. 269–278.

[36] MELLOR-CRUMMEY, J. M. Concurrent queues: Practical fetch-and-f algorithms. Tech. Rep. TR 229, Computer Science Department, University of Rochester, 1987.

[37] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing* (1996), pp. 267–275.

[38] MORAN, S., AND TAUBENFELD, G. A lower bound on wait-free counting. *J. Algorithms 24,* 1 (1997), 1–19.

[39] PRAKASH, S., LEE, Y. H., AND JOHNSON, T. A nonblocking algorithm for shared queues using compare-and-swap. . *IEEE Transactions on Computers 43,* 5 (May 1994), 548–559.

[40] SHALEV, O., AND SHAVIT, N. Split-ordered lists: lock-free extensible hash tables. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing* (2003), pp. 102–111.

[41] TREIBER, R. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM T.J. Watson Research Center, 1986.

[42] VALOIS, J. D. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems* (Las Vegas, NV, 1994), pp. 64–69.