Operation-Valency and the Cost of Coordination

[Extended Abstract]

Danny Hendler *
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel 69978
hendlerd@post.tau.ac.il

Nir Shavit
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel 69978
shanir@cs.tau.ac.il

ABSTRACT

This paper introduces *operation-valency*, a generalization of the valency proof technique originated by Fischer, Lynch, and Paterson. By focusing on critical events that influence the return values of individual operations rather then on critical events that influence a protocol's single return value, the new technique allows us to derive a collection of realistic lower bounds for lock-free implementations of concurrent objects such as linearizable queues, stacks, sets, hash tables, shared counters, approximate agreement, and more. By realistic we mean that they follow the real-world model introduced by Dwork, Herlihy, and Waarts, counting both memory-references and memory-stalls due to contention, and that they allow the combined use of read, write, and read-modify-write operations available on current machines.

By using the operation-valency technique, we derive an $\Omega(\sqrt{n})$ *non-cached shared memory accesses* lower bound on the worst-case time complexity of lock-free implementations of objects in *Influence*(n), a wide class of concurrent objects including all of those mentioned above, in which an individual operation can be influenced by all others.

We also prove the existence of a fundamental relationship between the space complexity, latency, contention, and "influence level" of any lock-free object implementation. Our results are broad in that they hold for implementations combining read/write memory and *any* collection of read-modify-write operations, and in that they apply even if shared memory words have unbounded size.

Categories and Subject Descriptors

C.1.4.1 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures,Distributed Architectures*

*This work was supported in part by a grant from Sun Microsystems.

PODC'03, July 13–16, 2003, Boston, Massachusetts, USA.

General Terms

Algorithms Theory

1. INTRODUCTION

In 1993, Dwork et. al [7] introduced a formal model to capture the real world phenomenon of memory contention on today's shared memory machines, machines that allow read, write, and read-modify-write (RMW) operations. Using FLP-style valency arguments, they proved that there are inherent tradeoffs between contention and latency[1] in concurrent data structure design. Their work was extended in several directions, most notably in the context of mutual exclusion [1, 2, 6] and counting networks [3, 4].

This paper presents *operation-valency*, a generalization of the valency proof technique of Fischer et. al (FLP) [9], and uses it to continue the above work in deriving real-world time complexity lower bounds for state of the art concurrent objects. As surveyed by Lynch [15] and by Fich and Rupert [8], there are numerous elegant extensions and reformulations of the FLP-style valency technique. The main difference between the operation-valency approach we present here and FLP-style arguments is that we focus on *temporary* changes in the anticipated results of *individual-operation solo executions* rather than on permanent changes in the valency of the protocol as a whole. In doing so, we are able to capture some of the complexity resulting from the influence among shared object operations that return distinct yet dependent values.

The time metric we use, which we call *memory steps*, counts only first-access shared memory events, and memory stalls due to contention in writing to shared locations. It is stricter than the time metric used by [7], as the later counts all shared memory references and also counts memory stalls due to contention in *reading*; it is similar to the *communication cost* metric used by Cypher [6], and to the *remote memory references* metric used by Anderson and Yang [1], and by Anderson and Kim [2], in that a single unit of both metrics corresponds to a shared memory reference that cannot be resolved by a local cache [2].

[1]In [7] Dwork et. al define a protocol's latency to be the maximal number of shared variable accesses, over all executions, a single high-level operation does.

[2]Note, however, that the communication cost metric and the remote memory references metric are stricter than our memory steps metric, since in distributed shared memory (DSM) systems, they do not count references to local memory, whereas such references *may* be counted as memory

We use the new operation-valency technique 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 individual operation can be influenced by $\Omega(n)$ other operations. *Influence*(n) includes data structures such as linearizable queues, stacks, shared counters, hash tables, sets, multi-sets, and approximate agreement objects. 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 briefly describe the operation-valency technique.

1.1 The Operation-Valency Technique

Valency arguments, introduced by Fischer et. al [9], have been used extensively [5, 7, 10, 14] to derive impossibility results and lower bounds for consensus and related problems. In problems such as consensus, a protocol is required to eventually output a *single* protocol value. Valency arguments classify system states according to whether they are *univalent* or *multivalent*. A state S is *univalent* if, in any two execution extensions starting from S, the protocol outputs the same value, and *multivalent* otherwise. Thus, essentially, a state S is unvialent iff the protocol's output value is already determined in S [3]. The valency technique looks at critical events that atomically change the system state from multivalent to univalent. Valency arguments are then applied w.r.t. to these critical events to derive impossibility results or lower bounds.

The basic idea behind our operation-valency technique is to generalize the approach of [9] by looking at the return values of individual operations instead of the single return value of the protocol as a whole. Similarly to FLP-style valency, we identify critical events that atomically affect return values, and we argue about the order and location of these events to obtain our results. We note, however, the significant differences between operation-valency and FLP-style valency:

- FLP-style valency looks at a single protocol output value; operation-valency looks at the output value of specific operations, in protocols where different operations are allowed to return different values;
- FLP-style valency looks at critical events that have *permanent* effect on the protocol output value: before the event, there exist two different execution extensions that yield two different output values; after the critical event is executed, *all* execution extensions yield the same protocol output value. Operation-valency looks at a different class of critical events, which we call *modifying events*. These events atomically modify the return value of a *solo execution* of a specific operation R: before such a modifying event e is executed, R's solo execution must return some value V; *right after* e is executed, R's solo execution must return some other value. If after e is executed, additional events are executed, a solo execution of R may, once more, have to return value V. This implies that the effect of a modifying event may be *temporary*.

steps by our metric.

[3]The output value may still be unknown to all participating processes, however, even in a univalent state.

As an example, consider an implementation of a "one time" n-process linearizable counter object allowing *fetch-and-increment* (FAI) operations. Before execution starts, any process may start a solo execution that returns 1. Let E be an execution where process p is idle, and assume some process q, $q \neq p$, completes its FAI operation in E, then a solo execution by p after E must return a value bigger than 1. We identify the critical modifying memory events, write, or RMW events, following which the return value of p's solo execution is modified. Our proof technique constructs executions in which such modifying events are pending, and shows that the operations whose return values are about to be atomically influenced by them must read the memory locations on which they are pending, otherwise we can construct indistinguishable executions that will lead to contradicting outcomes.

1.2 Our Results

To characterize the coordination requirements of shared objects, we introduce the *influence level* metric $\mathbb{I}$, informally defined as the maximal number of high level operations by other processes that can influence the outcome of another given process' high level operation. For example, in a linearizable shared counter, the outcome of a given operation can be influenced by $n-1$ others: if it runs alone it will return one value, but if any of the $n-1$ other processes precedes it, the value returned will be different.

1.2.1 New Fundamental Tradeoffs

We prove that the following fundamental relationships exist for all lock-free protocols. Let P be a lock-free protocol for a shared object with influence level $\mathbb{I}$; let $\mathbb{L}(P)$, $\mathbb{S}(P)$, and $\mathbb{C}(P)$ respectively denote the latency, space complexity, and write-contention of P, then:

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

For example, for linearizable counting this tradeoff strengthens a result of Herlihy et. al [12], which try to capture contention via a static measure of *capacity*: the maximal number of processes $c(P)$ that access any particular variable in any execution. They prove the existence of the tradeoff $\mathbb{L}(P) \geq (n-1)/c(P)$ between the capacity and latency of linearizable counters. However, they note that the capacity $c(P)$ is not necessarily correlated with contention. Our tradeoff captures a stronger relationship between latency and the *actual write-contention* for a broad class of problems, which for linearizable shared counting implies the desired relationship $\mathbb{L}(P) \geq (n-1)/\mathbb{C}(P)$. The above tradeoff also answers an open question posed by Dwork et. al [7] as to whether there exists a tradeoff between latency and contention for the approximate agreement problem.

1.2.2 New Time Complexity Lower Bounds

We identify *Influence*(n), the class of problems where a single operation has influence level $\mathbb{I} \in \Omega(n)$. This class includes lock-free implementations of key objects such as linearizable queues, stacks, hash-tables, sets, shared counters, approximate agreement, and consensus. We prove a lower bound of $\Omega(\sqrt{n})$ memory steps for any object belonging to *Influence*(n). This bound has an immediate real-world implication: any lock-free implementation for any of the above objects on any of today's architectures, using any combination of reads, writes, and RMWs, has a worst-case opera-

tion time-complexity of at least $\sqrt{n}$ shared memory accesses; moreover, we show that these accesses cannot be resolved by a local cache.

Our lower bounds are the first real-world time bounds for such objects. Though they seem stronger than Jayanti's interesting $\Theta(\log n)$ time bounds [13] on similar objects, they are in a sense orthogonal. This is because Jayanti's time metric does not count contention when accessing shared locations. For example, according to Jayant's metric, a shared linearizable counter for n processes can be implemented in constant time. Unlike our results, Jayanti's bounds are restricted to a model with only load-locked/store-conditional, a specific type of RMW operation.

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 $\Theta(\sqrt{n})$ time complexity. However, we believe the tight bounds on many interesting problems in *Influence*(n) are higher, and that the operation-valency approach might be the key to deriving them.

2. PRELIMINARIES

2.1 Shared Memory System Model

Our model of an asynchronous shared memory system is based on the model described by Cypher in [6], which is based, in turn, on the model given by Merritt and Taubenfeld [16]. We will assume that shared objects are specified as in [11]. Our model allows three types of shared memory *events*: atomic Read, atomic Write, and Read-Modify-Write events. No bound is assumed on register size. An n-process shared memory protocol (**E**,**P**,**R**,**I**) consists of a non-empty set **E** of executions, a set **P** of n processes, a set **R** of memory registers, and a function **I** that assigns an initial value to each register in **R**. An execution is a sequence (either finite or infinite) of events, where an event is an atomic memory access performed by a single process. An event can have one of the following three forms:

- Atomic Read: *read(p,r,v)* indicates that process p reads the value v from register r;
- Atomic Write: *write(p,r,w)* indicates that process p writes the value w to register r;
- Atomic RMW: *RMW(p,r,v,w)* indicates that process p atomically does the following: it reads the value v from register r and computes w based on v. p then proceeds in one of the following two ways: if w is different from the special value *null*, p writes the value w to register r, otherwise p does not write to register r. [4]

Given any event e, read(e) is true if e is either a Read or a RMW event, and *write*(e) is true if e is a Write event, or if e is a $RMW(p, r, v, w)$ event with $w \neq null$. $mem(e)$ is the memory register accessed (read and/or written) by e; $proc(e)$ is the process that executed e. For any $e \in E$, $index(e, E)$ is the number of events that precede e in E; when the execution discussed is clear from the context, we just use $index(e)$.

[4]This definition of RMW events captures both conditional (such as: Compare-and-swap, Test-and-set, Load-linked/Store-conditional) and non conditional RMW operations.

For any finite execution E and any sequence of events E', $E \circ E'$ denotes the concatenation of E and E'. For any sequence of events E', we define $procs(E')$ to be the set of processes that perform some event in E'. Let $r \in R$ be a memory register, and $E \in \mathbf{E}$ a finite execution, then $value(r, E)$ (the value of r after E) is the value written by the last event in E that wrote to r, or $\mathbf{I}(r)$ if there was no such event. Given an execution E and any subset $P \subseteq \mathbf{P}$, we let $proj(E, P)$ denote the subsequence of E containing only the events in E that were issued by processes in P. If $P = \{p\}$, we also use the notation $proj(E, p)$ instead of $proj(E, \{p\})$. If $proj(E_1, p) = proj(E_2, p)$, we say that the executions E_1, E_2 are *indistinguishable* by p.

DEFINITION 2.1. *A shared memory protocol satisfies the following* shared memory axioms ***A1 - A3:***

- ***A1:*** *If $E \circ e \in \boldsymbol{E}$, then $E \in \boldsymbol{E}$.*
- ***A2:*** *Let $E \circ e \in \boldsymbol{E}$ be an execution, and let e be a Read or RMW event, then the value read by e is $value(mem(e), E)$.*
- ***A3:*** *Let $E \circ e \in \boldsymbol{E}$ be an execution, and assume $proj(E, p) = proj(E', p)$, then $E' \circ e \in \boldsymbol{E}$ holds.*

Intuitively, Axiom **A1** states that a prefix of any possible execution is also a possible execution; Axiom **A2** states that the value read by any Read or RMW event returns the value of the most recent write to the accessed register (or the register's initial value if there were no earlier writes to that register); finally, Axiom **A3** states that if there are two possible executions that are indistinguishable by process p, and it is possible for p to perform a memory-operation (either Read, Write or RMW) following one of the executions, then it is also possible for p to perform the same operation following the other execution.

2.2 High Level Operations

Shared memory protocols support high-level operations that processes can execute. We consider protocols that support at least one operation-type that returns a value. High-level operations involve, in general, both private- and shared-memory events; in this paper we only deal with shared-memory events, and so we view a high-level operation Op as consisting of a sequence of one or more atomic shared memory events, each of which can be either *Read*, *Write*, or *RMW*. We only consider executions where each process performs at most a single high-level operation. [5] Let Op be an operation performed by some process in some execution E. We denote by $proc(Op, E)$ the process that executes Op in E; we denote by $events(Op, E)$ the sequence of memory-events performed by $proc(Op)$ while executing Op in E. Whenever E is clear from the context, we simply write $proc(Op)$ and $events(Op)$. We denote by $first(Op, E)$ and $last(Op, E)$ the first and last events, respectively, in $events(Op, E)$. (Note that $first(Op, E)$ and $last(Op, E)$ may be the same event.) As before, we omit E when it is clear from the context, and simply write $first(Op)$ and $last(Op)$. We say that an operation Op *starts* in E, and write $starts(Op, E)$, if $first(Op)$ appears in E. We say that an operation Op is contained within an execution E ($Op \subset E$) if all the events of $events(Op)$ appear in E. If

[5]Obviously, this just strengthens our lower bounds.

$Op \subset E$, we denote by $result(Op, E)$ the value returned by Op in E. Let Op_1, Op_2 be two operations contained in E; if $index(last(Op_1), E) < index(first(Op_2), E)$, we write $Op_1 \xrightarrow{E} Op_2$. Note that for any execution E, $\xrightarrow{E}$ is a partial order on all the operations contained in E. We say that an execution E is *quiescent*, if there is no operation that started in E but did not terminate in E. We say that a high-level operation Op is pending after an execution E, if Op can start at any time after E. A formal definition follows.

DEFINITION 2.2. *We say that a high-level operation Op is* pending *after an execution E, if Op did not start in E, and the following holds for every extension E' of E:*

$$E \circ E' \in \boldsymbol{E} \wedge (\neg starts(Op, E \circ E')) \Rightarrow E \circ E' \circ (first(Op)) \in \boldsymbol{E}$$

Slightly abusing notation, and for presentation simplicity, we sometimes refer to an underlying execution as a *state*. Thus, e.g., instead of saying that E is quiescent, we may say that the state after E is quiescent; instead of saying that a high-level operation Op is pending after E, we may say that after E the system is in a state where Op is pending.

2.3 The Time Complexity Metric

Our time complexity metric counts the worst-case number of *memory steps* that a single high-level operation may incur. Our metric counts both first-access shared memory events and *stalls* that are incurred when a few processes concurrently attempt to perform a Write or RMW operation to the same memory register. Formal definitions of *first-access events* and *stalls* follow.

DEFINITION 2.3. *Let $E \in \boldsymbol{E}$ be an execution; let e be an event in E; let $r = mem(e)$ be the memory register accessed by e, and let Op be a high-level operation such that $e \in events(Op, E)$, then we say that e is a* first-access event *of Op, if e is the first event in $events(Op, E)$ that reads and/or writes r.*

In other words, an event is a first-access event of a high-level operation Op, if it is the first event of Op to access some memory register.

DEFINITION 2.4. *Let $E \in \boldsymbol{E}$ be an execution, let $l > 0$ be an integer, and let $e_j, 0 \leq j \leq l$ be a sequence of consecutive events in E such that the following holds:*

1. $\forall j, 0 \leq j \leq l : e_j$ *is either a* Write *or a* RMW *event.*
2. $\forall j_1, j_2, 0 \leq j_1 \neq j_2 \leq l :$
   $(proc(e_{j_1}) \neq proc(e_{j_2})) \wedge (mem(e_{j_1}) = mem(e_{j_2}))$

Let Op be the high-level operation whose execution issued e_j, then we add j stalls to the memory step count of Op on account of e_j.

The above definition of stalls captures the fact that in shared memory systems, when a group of processes have pending Write or RMW events on the same memory location, then a scheduling adversary can release all these events simultaneously, thus causing the operation that issued the second event to incur a single stall, the operation that issued the third event to incur two stalls, and so on. [6]

[6]This definition of stalls does *not* assume that concurrent write events to the same memory-register are serviced in FIFO order, or in any other order.

2.4 The *Influence* Metric for Coordination Level

In this section we define a quantitative metric which is a measure of the coordination level of distributed protocols. More specifically, the *influence level* metric is a measure of the extent to which concurrently executing operations (which we call *influencing* operations or just *influencers*) can influence the result of another pending operation (which we call the *influenced* operation). To get a feel for this metric, consider an n-process protocol that implements a linearizable stack, with *push* and *pop* high-level operations. Consider a quiescent state S, where the stack contains a single item - the number 1. Assume that process p_1 has a pending *pop* operation, and each of the processes $p_i, 2 \leq i \leq n$ has a pending $push(i)$, for $2 \leq i \leq n$, respectively. Clearly, the value returned by the *pop* operation can be influenced by the *push* operations: if the *pop* operation is allowed a solo-execution while the *push* operations have not yet begun, then, from linearizability, it has to return 1; on the other hand, if we allow any interleaved execution of the *push* operations where at least one of them terminates, and only then start a solo-execution of the *pop* - then (again from linearizability) the *pop* must return a different value. Dependencies of this type are what we capture in the following definitions:

DEFINITION 2.5. *We say that a state S has* influence level K *(and write $\mathbb{I}(S) = K$) if the following holds:*

- *S is quiescent;*
- *There is a process that has a pending operation R so that a solo-execution of R starting from S returns some value V;*
- *There are K other processes, each having a pending operation $W_i, 1 \leq i \leq K$, such that the following holds: after any execution E, consisting of events issued by the operations W_i, where at least one operation terminates - a solo-execution of R returns a value other than V.* [7]
- *K is maximal.*

We say that R is K-influenced in S *and that the operations W_i are the* influencers *of R in S. We call V the* distinguished value *of R in S. If a state S is not quiescent, we define $\mathbb{I}(S)$ to be 0.*

We next extend the above definition of influence level to executions, protocols, and concurrent objects.

DEFINITION 2.6. *The* influence level *of an execution E, denoted by $\mathbb{I}(E)$, is the maximum influence level over all the states E reaches; the influence level of a protocol P, denoted by $\mathbb{I}(P)$, is the maximum influence level over all its executions.*

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

DEFINITION 2.7. *A concurrent object O has* influence level $\mathbb{I}$, *if the influence level of every lock-free protocol implementing it is at least $\mathbb{I}$.*

[7]It is not assumed that the operations W_i return a value.

Based on the above definitions, we can now determine a lower bound on the influence level of linearizable implementations of a concurrent object, based on the object's sequential specification.

LEMMA 2.1. *Let O be an object whose sequential specification S includes a history H, such that:*

- *H can be extended by a value-returning operation R, and $result(R, H \circ R) = V$ for some V;*
- *There are K operations, $W_i, 1 \leq i \leq K$, such that the following holds: for any non-empty subset of indices $T \subseteq \{1, \cdots, K\}$, and for every permutation σ_T of T, the following extension of H exists: $H_{\sigma_T} = H \circ W_{\sigma_T(1)} \circ \cdots \circ W_{\sigma_T(|T|)} \circ R$ and $result(R, H_{\sigma_T}) \neq V$.*

Then any linearizable implementation of O has influence level at least K.

PROOF. Immediate from the definition of influence level and from the linearizability of the implementation. □

3. TRADEOFFS AND LOWER BOUNDS

We consider a lock-free n-process protocol, P, for $n \geq 2$, that has influence level K. We define P's *latency* as the maximal number of shared memory events issued by a single high-level operation, over all executions, and denote it by $\mathbb{L}(P)$; we define P's *read latency* as the maximal number of read/RMW shared memory events issued by a single high-level operation, over all executions, and denote it by $\mathbb{L}_{\mathbf{R}}(P)$; we define P's *space complexity* as the total number of shared memory registers read/written by P, over all executions, and denote it by $\mathbb{S}(P)$; we define P's *write-contention* as the maximal number of consecutive write/RMW events to the same memory register, over all executions, and denote it by $\mathbb{C}(P)$. Recall that the definition of a protocol's influence level states that P can be brought to a state with influence-level K. For presentation-simplicity, and w.l.o.g., the proofs in this section assume that the execution starts from such a state S, where process p has a pending influenced operation R with distinguished value V, and each process $q_i \in Q, 1 \leq i \leq K$ has a pending influencer W_i. Our proofs need only consider executions where each process executes at most a single operation[8].

The following lemma defines what *modifying events* are and proves their existence.

LEMMA 3.1. *Let E be an execution such that $procs(E) \subseteq Q$, and assume some influencer W_i completes in E, then there is at least one event $e \in E$, which we call a* modifying event, *such that the following holds:*

- *Until e is executed, a solo execution of R must return V;*
- *If a solo execution of R starts immediately after e, then it returns a value other than V.*

PROOF. Since S has influence level K w.r.t. R, with V as its distinguished value, then before E starts, a solo execution of R must return V. On the other hand, since some influencer completes in E, then, again from the definition of influence, a solo execution of R cannot return V after E, hence the result follows. □

[8]Obviously this just strengthens our lower bounds.

Intuitively, we consider executions that start from a state S, with influence level K, that only include events executed by the influencers. A modifying event e is an event within such an execution. Before e occurs, a solo execution of R must return V; *right after* e is executed, a solo execution of R must return a different value.

We now prove that modifying events are either write or RMW events.

LEMMA 3.2. *Let $e \in E$ be a modifying event, then $write(e)$ holds.*

PROOF. Assume by way of contradiction that the claim does not hold, then e does not write any value. Let $E = E' \circ e \circ E''$, then from the definition of modifying events, right after E' (i.e. just before e) p can execute a solo execution of R that returns V, and so there is an extension E_1 of E' such that $E_1 = events(R)$, $proc(R) = p$, and $result(R, E' \circ E_1) = V$. Since the modifying event e does not involve a write, and since e is not an event of p, the executions E' and $E' \circ e$ are indistinguishable by p, and therefore, by Axiom $A3$ of the shared memory model, $E' \circ e \circ E_1$ is also an execution of P, which implies that a solo execution of R returns V also right after e is performed. This obviously implies that e is not a modifying event, a contradiction. □

The following lemma proves that starting from state S, the processes in Q can be brought to a state where they all have pending modifying events.

LEMMA 3.3. *There is a finite execution $E \in \boldsymbol{E}$, such that following E every $q \in Q$ has a pending modifying event.*

PROOF. From the definition of influence level, no influencer can terminate before some modifying event is performed. We construct an execution E where every process $q \in Q$ has a pending modifying event, by letting the processes $q_i, 1 \leq i \leq k$, initiate simultaneously their influencers, and by letting each q_i execute its influencer, until it is about to perform a modifying event. Note that this is an iterative construction, as bringing one process to be on the verge of performing a modifying event can take another process out of this state, but the constructed execution is finite because P is lock-free. □

As a corollary we can prove the following tradeoff between the space-complexity and write-contention of any lock-free object implementation.

THEOREM 3.4. *Let O be an object with influence level $\mathbb{I}$ and let P be a lock-free implementation of O, then the following holds:*

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

PROOF. Since P implements O, $\mathbb{I}(P) \geq \mathbb{I}$. Consequently, from Lemmas 3.2 and 3.3, P can be brought to a state where at least $\mathbb{I}$ write or RMW events are pending. Since at most $\mathbb{C}(P)$ such events can be pending on any single register, the result follows. □

We next prove a similar tradeoff between the latency and write-contention of lock-free implementations. We actually prove a stronger result, by showing that the tradeoff holds even if we exclude write events from the latency count.

THEOREM 3.5. *Let O be an object with influence level $\mathbb{I}$ and let P be a lock-free implementation of O, then the following holds:*

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

PROOF. As all the events counted by $\mathbb{L}_{\mathbf{R}}$(P) are also counted by $\mathbb{L}$(P), the first inequality is obvious. As for the second inequality, let $\mathbb{I}(P) = K$. Since P implements O, we have $K \geq \mathbb{I}$. From Lemma 3.3, P has an execution E, such that following E each of $q_1, \cdots q_K$ has a pending modifying event. Let B be the set of registers to which these outstanding events are about to write. Note that $|B| \geq \lceil K/\mathbb{C}(\mathrm{P}) \rceil$. Since no modifying event has been performed yet, then from the definition of modifying events E can now be extended by a sequence E', which are the events of the solo execution of operation R by process p, so that $result(R) = V$. We prove the theorem by showing that R must read all the registers in B. Assume to the contrary, then R returns a value without reading some register $r \in B$. Let e be some modifying event which is outstanding on register r, then clearly $proj(E \circ e \circ E', p) = proj(E \circ E' \circ e, p)$ and so R must return the same value whether it executes before the event e or after it, which is a contradiction to our assumption that e is a modifying event. □

Based on Theorem 3.5, we can now establish a lower bound on the memory steps complexity of any lock-free object implementation.

THEOREM 3.6. *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. Let $\mathbb{I}(P) = K$. Since P implements O, we have $K \geq \mathbb{I}$. By using Lemma 3.3, there is an execution E, such that following E each of $q_1, \cdots q_k$ has a pending modifying event. According to Lemma 3.2 all of the modifying events are either Write or RMW events. Let B be the set of registers to which these outstanding events are about to write. Assume that $|B| < \lfloor \sqrt{K} \rfloor$, then there is at least one register $r \in B$ that has at least $\lfloor \sqrt{K} \rfloor$ outstanding events about to write to it. Let W_j be the influencer whose outstanding event is executed last, then W_j is charged by $\lfloor \sqrt{k} \rfloor - 1$ memory steps because of the stalls it incurs when accessing r, plus an additional memory step on account of the first access of r, which proves the theorem. Otherwise, $|B| \geq \lfloor \sqrt{K} \rfloor$, and based on Theorem 3.5 R can be made to access all the registers in B. Consequently we can charge R by $|B|$ memory steps for the first-access events of all the registers in B. □

3.1 Memory Steps and Time

We now discuss how the memory steps lower bound translates to a time lower bound. For this, we need the following definition:

DEFINITION 3.1. *Let M be a shared memory multiprocessor; we denote by* nlcr-time(M) *the minimal time a non-local-cache-reference takes in M, i.e. the minimal time in M of a memory reference that is not resolved by the local cache (if any) of the processor that issued it.*

If M is a distributed shared memory system without caches, *nlcr-time(M)* is simply the minimal time it takes a processor in M to access its local memory (the minimum taken over all processors); if M is a cache-coherent multiprocessor, then the minimum in the above definition is taken over all memory references that cannot be resolved by the local cache and generate interconnect traffic, such as: references to a memory location that is not in the local cache; writes that generates cache-invalidate transactions; writes that generate cache-update transactions, or any other shared memory references not resolved in the local cache.

In the proof of Theorem 3.6 we have shown that either some operation incurs at least $\lfloor \sqrt{\mathbb{I}} \rfloor$ consecutive stalls, or some other operation performs at least $\lfloor \sqrt{\mathbb{I}} \rfloor$ first-access events. We now analyse both cases.

- In any shared-memory multiprocessor M, when multiple processors attempt to write to the same memory register simultaneously, the writes are being serialized and are serviced one after the other. Moreover, even if M is a cache-coherent system, x consecutive stalls take at least $x \cdot$ *nlcr-time(M)* time: if the cache scheme is *write-through*, then every write generates a cache miss; and even if the cache scheme is *write-back*, then since the writes are by different processors, none of them (except, maybe, the first) can be accomplished by just updating the local cache: they have to either invalidate or update other caches.
- Clearly in no shared-memory multiprocessor can a first-time shared memory read be resolved from the local cache; as for first-time shared memory writes, in write-through cache schemes every write generates a cache-miss, and in write-back cache schemes, the first write to a shared location must either invalidate or update other caches, which implies interconnect traffic. Consequently, if we assume M does not support non-blocking reads and writes, then x first-time access events by an operation take at least $x \cdot$ *nlcr-time(M)* time. If M *does* support multiple outstanding references per processor, then, theoretically, x first-access references may be resolved in a time equivalent to x cache references.

3.2 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 key distributed objects belong to this class, and thus have an inherent operation complexity of $\Omega(\sqrt{n})$ memory steps.

DEFINITION 3.2. *A* generic object *O is an object that is specified for any number of processes n. 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 3.3. Influence(n) *is the objects class that contains all generic objects O such that $\mathbb{I}_O$ is in $\Omega(n)$.*

It is easily shown that the following objects are in *Influence(n)*: linearizable counters, stacks, queues, hash-tables, sets, approximate agreement. As two examples, we show that approximate agreement and linearizable counting belong to *Influence(n)*.

An approximate agreement object supports a single *decide* operation. Each participating process calls *decide* with the

process' real-number input-value. The values returned by the *decide* operation to different processes are required to be within a given distance ϵ of each other, and are also required to be within the range of the inputs.

THEOREM 3.7. *Approximate agreement is in* Influence(n).

PROOF. We prove that any lock-free approximate agreement protocol for n processes has influence level $n-1$. Consider the problem instance where process $p_i, 1 \leq i \leq n$, starts with value $2(i-1)\cdot\epsilon$. We denote the *decide* operation executed by process i as $decide_i$. We prove that the initial state, S, has influence level $n-1$, by showing that $decide_1$ is $(n-1)$-influenced in S, with $decide_i, 2 \leq i \leq n$ as its influencers: clearly, if $decide_1$ runs alone, it must return 0. On the other hand, in any execution E which does not involve p_1, in which some other process decides - the decision value must be in the range $[2\epsilon \cdots 2(n-1)\epsilon]$, and so if $decide_1$ starts a solo execution after E, $decide_1$ must return a value no less than ϵ. □

A shared counter object supports a single fetch-and-increment (FAI) operation. The counter-values returned by FAI operations are required to be unique natural numbers. It is also required that in quiescent states the values distributed by the counter constitute a contiguous range of numbers. *Linearizable shared counters* are also required to be linearizable, i.e. if FAI_i and FAI_j are two activations of the FAI operation, and $FAI_i \xrightarrow{E} FAI_j$ in an execution E, then $result(FAI_i, E) < result(FAI_j, E)$ must hold.

THEOREM 3.8. *Linearizable counting is in* Influence(n).

PROOF. We prove that any lock-free linearizable-counting protocol for n processes has influence level $n-1$. Consider the sequential specification of a shared counter object LC for n processes, and let H be a history after which the counter value is V. We denote an FAI operation performed by process i by FAI_i. H can be extended by FAI_1, that must return V. On the other hand, if the execution of FAI_1 is preceded by any $FAI_i, 2 \leq i \leq n$, operations, in any order, then clearly FAI_1 must return a value greater than V. Consequently, by using Lemma 2.1 the result follows □

The proofs that linearizable stacks, queues, sets and hash-tables are in *Influence(n)* are very similar to the proof of Theorem 3.8, and are consequently omited.

We next present the *First-Generation* problem. We show that it belongs to the *Influence(n)* class and that it can be implemented in $O(\sqrt{n})$ memory steps; thus we prove, that there are problems in *Influence(n)* for which our bound is tight. Let E be an execution; we say that an operation Op belongs to the first-generation of E, and write $Op \in FG(E)$, if it has no predecessor in the partial-order induced by $\xrightarrow{E}$.

DEFINITION 3.4. *A* First-Generation *object supports a single operation* - First, *which every process can call once. The operation returns a boolean value. Any correct implementation must meet the following requirements for every execution E:*

- *An operation which is not in $FG(E)$ cannot return* true*;*
- *If all the operations in $FG(E)$ terminate, then at least one of them returns* true.

LEMMA 3.9. First-Generation *is in* Influence(n).

PROOF. We denote by $First_i$ the *First* operation performed by process $i, 1 \leq i \leq n$. It is immediate from the problem-definition that the initial state has influence level $n-1$, with (e.g.) $First_1$ an $(n-1)$ influenced-operation and $First_i, 2 \leq i \leq n$ its influencers. □

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

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

Figure 1: First Operation Code

The proof of the following lemma is straightforward and is therefore omitted.

LEMMA 3.10. *The code shown in Figure 1 correctly implements a* First-Generation *object, and has memory steps complexity of $\Theta(\sqrt{n})$.*

4. DISCUSSION AND FURTHER RESEARCH

This paper introduces the operation-valency technique and the influence metric for reasoning about multi-valued protocols, and uses them to obtain $\sqrt{(n)}$ time lower bounds for a broad class of objects. The time metric we use - *memory steps* - is similar to the communication-cost metric and the remote-memory-references metric used by [1, 2, 6] in that it counts only memory references that cannot be resolved by a local cache.

We have proven an $\Omega(\sqrt{n})$ time lower bound for all objects in the *Influence*(n) class, and we have also shown that the bound is tight for some objects in it. For most of the interesting objects in *Influence*(n), however, including linearizable counters, stacks and queues, all known lock-free implementations require $\Omega(n)$ time. Note that for linearizable objects such as these, differently from the *First-Generation* object, there's generally a requirement of *distinctness* - i.e. there are scenarios in which all n high-level operations are required to return distinct values. Finding the tight time complexity for this class of objects remains an interesting open problem.

It would also be interesting to see whether our $\Omega(\sqrt{n})$ lower bound for a single operation holds also for the protocol's amortized complexity, possibly by showing that it holds for $\Omega(n)$ different operations.

5. ACKNOWLEDGEMENTS

We would like to thank Ori Shalev for helpful comments. Comments of the anonymous PODC referees were also very helpful.

6. REFERENCES

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

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

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

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

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

[6] 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.

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

[8] F. E. Fich and E. Ruppert. Lower bounds in distributed computing. In *International Symposium on Distributed Computing*, pages 1–28, 2000.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

[13] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[14] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

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

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