A Paradox of Eventual Linearizability in Shared Memory

Rachid Guerraoui
EPFL, Switzerland

Eric Ruppert
York University, Canada

ABSTRACT

This paper compares, for the first time, the computational power of linearizable objects with that of eventually linearizable ones. We present the following paradox. We show that, unsurprisingly, no set of eventually linearizable objects can (1) implement any non-trivial linearizable object, nor (2) boost the consensus power of simple objects like linearizable registers. We also show, perhaps surprisingly, that any implementation of an eventually linearizable complex object like a fetch&increment counter (from linearizable base objects), can itself be viewed as a fully linearizable implementation of the same fetch&increment counter (using the exact same set of base objects).

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

Concurrent; asynchronous; linearizable; consistency; wait-free; obstruction-free; fetch-and-increment; consensus

1. INTRODUCTION

A central activity in shared-memory computing is that of raising the abstraction level of synchronization primitives by building, in software, higher-level inter-process communication objects than those provided in hardware. The goal is usually to ensure that, even if the constructed objects have richer semantics, they appear as if they were provided in hardware. In particular, every process should be able to access the shared object independently of contention, i.e., even when other processes may be accessing the same object. This property, called *wait-freedom* [9], says that if we build, for instance, a fetch&increment counter in software, then every process should be able to increment the counter and get its new value, independently of whether other processes that are accessing the counter concurrently have been swapped or paged out. This requirement rules out the usage of locks. However, in order to provide the illusion of a single shared object, processes must synchronize using underlying hardware primitives, such as compare&swap. This illusion, in turn, is typically captured by the property of being *linearizable* [11]. This property says that even if they are actually performed concurrently, the operations issued on that object appear as if they are executed sequentially.

Ensuring either linearizability or wait-freedom alone is simple. What is challenging, and sometimes considered too expensive, is to ensure both at the same time. Two major research directions have been considered to reduce the difficulty. The first is to weaken wait-freedom and devise algorithms that "simply" ensure lock-freedom, obstruction-freedom, or even use locks in specific portions of the code [10]. The rationale here is that the conditions under which the operating system swaps out a process for a long period of time do not happen very often. A lot of theoretical work has been devoted in the last decades to this avenue of research. The second approach is to weaken linearizability: for some applications, weaker consistency conditions suffice [19]. Consider a shared fetch&increment abstraction used to count references of objects in a concurrent setting. As pointed out, this would typically be implemented in software using the system's compare&swap objects. If several compare&swap tentatives fail due to unusually high contention, it may be acceptable to return a temporary value of the counter, as long as, eventually, all increments of concurrent processes are taken into account in some future value of the counter. In other words, in the implementation, if a process is taking too long to synchronize perfectly, because of contention, it could give up trying to get the most up-to-date value and (a) do its increment locally, making sure later that its increments are eventually counted, and (b) get a value of the counter to return that is lower than the true value, assuming that the process will get information about concurrent increments later.

The notion of *eventual consistency* [19] aims to capture precisely this notion of "intermittent inconsistency". The approach is analogous to the way *self-stabilization* was defined, namely stating that the system should eventually work correctly forever beyond some stabilization time, with the goal of modelling intermittent failures and the return of the system to normal after the failure. Obviously, if instead of consistency, implementations only need to implement an eventual form of it, they would offer weaker guarantees. In our example above, the value returned by the counter might be temporarily inconsis-

PODC'14, July 15–18, 2014, Paris, France.
Copyright 2014 ACM 978-1-4503-2944-6/14/07 ...$15.00.
http://dx.doi.org/10.1145/2611462.2611484 .

tent with the eventual value. But would these implementations be easier to implement? Maybe surprisingly, we show that the answer is no in some situations. More specifically, we highlight the following paradox.

- On the one hand, unsurprisingly, eventual linearizability is strictly weaker than linearizability in the following two senses. (1) No set of eventually linearizable objects can implement any non-trivial linearizable object. (A trivial object is one that can be implemented without inter-process communication.) This is true even if we only require the resulting linearizable object to be obstruction-free, whereas the eventually linearizable base objects are wait-free. (2) No set of eventually linearizable objects can boost the consensus power of simple linearizable objects like registers, i.e., help solve consensus among two processes in conjunction with linearizable registers. In short, eventually linearizable objects can neither implement linearizable ones (1) nor augment the power of shared linearizable registers (2).
- On the other hand, eventual linearizability can be as hard to implement as linearizability: any implementation of an eventually linearizable fetch&increment counter (from a set of linearizable base objects), yields an implementation of a fully linearizable fetch-&increment counter (from the same set of base objects). Roughly speaking, this is because a fetch&increment object continues to require synchronization forever. In contrast, such a result does not hold for objects that require synchronization only in the beginning of an execution: for example, consensus and test&set objects are trivial to implement in an eventually linearizable manner. Thus, an eventually linearizable implementation of consensus (respectively, test&set) will clearly *not* yield a linearizable implementation of consensus (respectively, test&set).

In fact, we show that any algorithm A that implements an eventually linearizable fetch&increment object, starting from a given initial state of the object, can itself be viewed as an algorithm A' that implements a fully linearizable fetch&increment object, simply starting from a different initial state of the counter than A. Basically, the variables of A' are those of A, initialized after a global "stabilization time" of eventual linearizability. Interestingly, this is in spite of the fact that the stabilization time is not *a priori* fixed: it could be different in different executions (and this is allowed by our definition of eventual linearizability). The main difficulty of the proof is to show that there must exist a stable configuration C of algorithm A where *every* possible run that passes through C has already stabilized *at* C. (Then, we initialize the variables of A as they are in C to get A'.)

2. RELATED WORK

Eventual consistency originated in the context of replicated systems where updates to the replicas are propagated using gossipping to improve latency and tolerate asynchrony and partitions [4, 6, 18]. Because gossipping is used, replicas are not always consistent, but eventually become so if the updates stop. More recently, researchers argued eventual consistency is a reasonable alternative to consistency in the cloud context. The argument is that consistency does not scale and is not necessary for many applications, e.g., [1, 17, 19]. So far, research on eventual consistency has focused on describing systems that implement some form of eventual consistency.

Some attempts have been made to formalize the semantics of eventual consistency. In [14, 15], for instance, the focus was on abstract properties, while in [2] the goal was to show how to verify replication schemes that ensure only eventual consistency. However, little effort has been devoted to theoretical studies that would precisely measure the power and limitations of eventual consistency (beyond artifacts related to specific implementations). Two notable exceptions are [16] and [5].

In [16], Serafini et al. introduced the concept of eventual linearizability and addressed the problem of the weakest failure detector to implement objects for which some operations are linearizable and some operations are not. In a sense, their work is orthogonal to ours. The main common feature is the definition of eventual linearizability. There are however two differences between our definition of an eventually linearizable implementation and the definition used by Serafini et al. The first is a minor difference. Serafini et al. used a timed model of computation, and talked about stabilizing after time t rather than after t events. This is due to their underlying message-passing model equipped with failure detectors where timing assumptions are used, while we consider a totally asynchronous shared-memory model, where time plays no role. The second, more important difference, has to do with quantifiers. Serafini et al. defined eventual linearizability of an implementation by saying that there is a *single* time t such that *all* executions stabilize by time t. We do not require a single t for all executions: the number of events before executions stabilize may vary, and even be unbounded. Serafini et al. motivate their choice of focusing only on *finite* executions by arguing that any finite history is trivially linearizable after some t (because t can be chosen large enough to satisfy the definition vacuously), and thus the notion of eventual linearizability makes sense only if there is a single t for all executions. We take a different approach. We consider *infinite* executions, which do not trivially satisfy the property of being t-linearizable for some t. We made this choice following an analogy with the literature on self-stabilization, which also deals with a property describing eventual good behaviour.

Dubois et al. [5] consider a message-passing environment and compare the weakest failure detector to implement total order broadcast with the weakest failure detector to implement eventual total order broadcast. Their results do not provide a comparison between linearizable and eventually linearizable implementations of specific objects. We show in this paper that some eventually linearizable objects are trivial to implement in an asynchronous system (and would thus not require any failure detector), whereas others are as hard to implement as their linearizable counterparts. Note that the fact that two abstractions have the same weakest failure detectors does not mean that any implementation of the first is also an implementation of the second (which is what we show for fetch&increment objects in this paper). For instance, [3] shows that two object types T_1 and T_2 can have the same weakest failure detectors even if the consensus number of T_1 is greater than that of T_2, meaning that T_2 cannot implement T_1.

To our knowledge, this paper is the first to compare the computational power of linearizable and eventually linearizable objects in a shared-memory context.

3. MODEL, DEFINITIONS AND PROPERTIES

A shared-memory system consists of a collection of processes that communicate by accessing shared objects of various types. Each type of object has a *sequential specification*, which consists of

- a set Q of possible states,
- a set of possible initial states $Q_0 \subseteq Q$,
- a set INV of possible operation invocations,
- a set RES of possible operation response values, and
- a transition relation $\delta \subseteq Q \times OP \times RES \times Q$.

Intuitively, if $(q, op, r, q') \in \delta$, it means that when the object is in state q and operation op is applied to it, the object can return the response r and move into state q'. We consider the name of an operation in OP to include all of the operation's arguments. A type T is *deterministic* if, for each state q and each operation op, there is a unique result r and state q' such that $(q, op, r, q') \in \delta$. A type T has *finite non-determinism* if, for each state q and each operation op, there are finitely many pairs (r, q') such that $(q, op, r, q') \in \delta$.

An *event* is a tuple $\langle p, o, x\rangle$ where p is a process name, o is the name of an object of some type T and x is either an invocation or response for type T. A computation of the distributed system is described by a sequence of such events, called a *history.* We use $H|o$ to denote the subsequence of history H consisting of events at object o and $H|p$ to denote the subsequence H consisting of events performed by process p. A history is *sequential* if it consists of operation invocation and response events, starting with an invocation, where each invocation $\langle p, o, inv\rangle$ (except possibly the last, if the history is finite) is immediately followed by a matching response $\langle p, o, res\rangle$. Any history H mentioned in this paper is assumed to be *well-formed*, i.e., the subsequence $H|p$ is sequential for each process p. An *operation* consists of an invocation event and its matching response event (if it exists).

A sequential history H is *legal* if, for each object o, the subsequence $H|o = \langle p_1, o, i_1\rangle, \langle p_1, o, r_1\rangle, \langle p_2, o, i_2\rangle, \langle p_2, o, r_2\rangle, \ldots$ satisfies the following with respect to o's sequential specification $(Q, Q_0, INV, RES, \delta)$: there is a sequence of states $q_0, q_1, \ldots$ in Q such that $q_0 \in Q_0$ and, for each $j \geq 0$, $(q_j, i_j, r_j, q_{j+1}) \in \delta$.

An *implementation* of an object of type $T = (Q, Q_0, INV, RES, \delta)$ provides, for each $q_0 \in Q_0$, a programme that each process can follow to perform each operation op in INV. When the programme terminates, it outputs a result in RES. Since we focus on a shared-memory model, the programme can make use of shared base objects: it can invoke an operation on a base object and await a response from it. In a concurrent *execution* of the implementation, each process repeatedly executes the programme to perform an operation until the programme generates a response to the operation, and the steps of different processes may be interleaved in an arbitrary way. Each execution defines a history, the sequence of invocations and responses on the object of type T. In Section 3.1, we define eventual linearizability, which is a correctness condition for an implementation, that constrains the set of possible histories that can arise from executions of the implementation.

An implementation is *wait-free* if each operation completes within a finite number of steps of the process performing it. An implementation is *non-blocking* if, whenever some operation is pending and processes continue to take steps, some operation is eventually completed. (Some authors call the non-blocking progress property lock-freedom.) An implementation is *obstruction-free* if, there is no execution where only one process takes steps in an infinite suffix without completing its operation.

Generally, the focus of distributed computing research is on the interprocess communication, and the computational power of the processes is not specified. For this work, we assume that the programmes are written in a programming language that has equivalent power to the Turing machine model (but includes a mechanism for accessing the shared base objects). Correspondingly, we assume that the transition relations of object type specifications are Turing-computable.

3.1 Eventual Linearizability

We now give several definitions that are used in defining eventual linearizability. These are based on the definitions in [16], but differ in the ways described in Section 2. The first property, weak consistency, ensures that operations cannot return responses that are "out of left field," even during the initial period when absolute consistency is not enforced. In particular, the response to each operation should take into account earlier operations done by the same process and should be a possible response if the process knows only about a subset of the operations performed by other processes.

DEFINITION 1. *A history H is* weakly consistent *if, for each operation op that has a response in H, there is a legal sequential history S that*

- *contains only operations that are invoked in H before op terminates,*
- *contains all operations that are performed by the same process that does op and precede op in H, and*
- *ends with the same response to op as in H.*

Now we define the notion of being linearizable "after the first t events have occurred."

DEFINITION 2. *Let H be a history and t be a natural number. Let H' be the suffix of H obtained by removing the first t events. Then, a legal sequential history S is called a* t-linearization *of H if*

- *each operation invoked in S is invoked in H,*
- *each operation completed in H is completed in S,*
- *if op_1's response is before op_2's invocation and both of these events are in H' and op_2 is in S, then op_1 precedes op_2 in S, and*
- *each operation that has a response in H' has the same response in S.*

A history is t-linearizable *if it has a t-linearization.*

DEFINITION 3. *A history is* eventually linearizable *if it is weakly consistent and t-linearizable for some t.*

DEFINITION 4. *An implementation of an object is* eventually linearizable *if every history is eventually linearizable. (Note that different histories can be t-linearizable for different values of t.)*

3.2 Basic Properties

We start with some easy properties of t-linearizability.

Lemma 5. *If a history H is t-linearizable, then it is also t'-linearizable for any $t' > t$.*

Proof. If S is a t-linearization of H, then S is also a t'-linearization of H. $\square$

Lemma 6. *If a history H is t-linearizable, then every prefix of H is also t-linearizable.*

Sketch of proof. Let S be a t-linearization of H. A prefix of H of length at most t is trivially t-linearizable. So, consider a prefix H_1H_2 of H where $|H_1| = t$. Let S_1 be the shortest prefix of S containing all operations that terminate in H_2. Let S_2 be an arbitrary permutation of the operations that terminate in H_1 but do not appear in S_1 (with the responses they would have if executed in that order after S_1). Then, it is easy to check that S_1S_2 is a t-linearization of H_1H_2. $\square$

Serafini et al. [16] prove the following proposition using their slightly different definition of t-linearizability. Note, however, that the proof holds only for histories involving a finite number of objects (an assumption that should have been included in [16]).

Lemma 7. *(Proved in [16]) A history H involving a finite number of objects is t-linearizable for some t if and only if, for each object o, there is a t_o such that $H|o$ is t_o-linearizable.*

The proof of Lemma 7 is based on the proof in [11] that linearizability is a local property, and it carries over to our definition. For the "only if" direction, we can take $t_o = t$. For the "if" direction, we choose t large enough so that the first t events of H include the first t_o events of $H|o$ for each o. This is possible because the set of objects o is finite.

The proof of the following lemma in [16] has a small error, so for completeness, we provide a detailed proof.

Lemma 8. *A history H is weakly consistent if and only if, for each object o, $H|o$ is weakly consistent.*

Proof. Suppose H is weakly consistent. Let o be any object and *op* be any operation with a response in $H|o$. Since H is weakly consistent, there is a sequential history S that satisfies Definition 1 for *op* in H. Then, $S|o$ satisfies Definition 1 for *op* in $H|o$. So, $H|o$ is weakly consistent.

Now, suppose $H|o$ is weakly consistent for each object o. Let *op* be any operation that has a response in H. Let p be the process that performs *op* and o be the object *op* is performed on. Let S be the legal sequential history that satisfies Definition 1 for *op* in $H|o$. Let S' be obtained from S by inserting at the beginning of S any operations performed by p before *op* in H on objects other than o, with responses as dictated by the objects' sequential specifications. Then, S' satisfies Definition 1 for *op* in H. Thus, H is weakly consistent. $\square$

The next proposition is immediate from Lemma 7 and 8.

Proposition 9. *A history H that involves a finite number of objects is eventually linearizable if and only if, for each object o, $H|o$ is eventually linearizable.*

Proposition 9 does not hold for some histories that use infinitely many objects. For example, consider the following sequential history H that uses read/write registers $R_1, R_2, \ldots$, all initialized to the value 0.
$\langle p, R_1, write(1)\rangle, \langle p, R_1, ack\rangle, \langle q, R_1, read\rangle, \langle q, R_1, 0\rangle,$
$\langle p, R_2, write(1)\rangle, \langle p, R_2, ack\rangle, \langle q, R_2, read\rangle, \langle q, R_2, 0\rangle,$
$\langle p, R_3, write(1)\rangle, \langle p, R_3, ack\rangle, \langle q, R_3, read\rangle, \langle q, R_3, 0\rangle, \ldots$
If t_i is chosen so that the response to the read of R_i occurs before the t_ith event of H, then $H|R_i$ is t_i-linearizable and it is easy to see that $H|R_i$ is weakly consistent, so $H|R_i$ is eventually linearizable for every i. However, H is not eventually linearizable: for any t, there is an i such that the write to R_i is invoked after the tth event of H, and $H|R_i$ is not t-linearizable, so H is not t-linearizable.

A *safety property* is a set of histories that is non-empty, prefix-closed and limit-closed. A *liveness property* is a set of histories such that every finite history is the prefix of some history in the set. The following lemma follows easily from the definition of weak consistency.

Lemma 10. *Weak consistency is a safety property.*

Proof. The empty execution (consisting of no events) vacuously satisfies Definition 1, so the set of weakly consistent histories is non-empty.

Next, we prove that the set of weakly consistent histories is prefix-closed. Suppose H is weakly consistent. Let H' be a prefix of H. Let *op* be an operation that terminates in H'. Let S be the sequential history that satisfies definition 1 for *op* in H. Then S also satisfies Definition 1 for *op* in H'. Thus, H' is weakly consistent.

To show the set of weakly consistent histories is limit-closed, let $H_1, H_2, \ldots$ be an infinite sequence of histories such that, for each i, H_i is weakly consistent and H_i is a prefix of H_{i+1}. We show that $H = \lim_{i\to\infty} H_i$ is weakly consistent. Let *op* be an operation that terminates in H. Then there exists an i such that *op* terminates in H_i. Let S be a sequential history that satisfies Definition 1 for *op* in H_i. Then, S also satisfies Definition 1 for *op* in H. $\square$

0-linearizability is equivalent to linearizability [11]. For deterministic objects, Lynch [13] proved that linearizability is a safety property. The proof extends easily to objects with finite non-determinism [8]. However, linearizability is *not* a safety property for objects with infinite non-determinism [8].

For $t > 0$, t-linearizability is *not* a safety property, even for deterministic types. Consider a fetch&increment object, which stores a natural number and provides a single operation, $fetch\&inc$, which adds one to the value stored and returns the old value. Consider the infinite sequential history which uses a single fetch&increment object X initialized to the value 0.
$\langle p, X, fetch\&inc\rangle, \langle p, X, 0\rangle, \langle q, X, fetch\&inc\rangle, \langle q, X, 0\rangle,$
$\langle q, X, fetch\&inc\rangle, \langle q, X, 1\rangle, \langle q, X, fetch\&inc\rangle, \langle q, X, 2\rangle, \ldots$
Every finite prefix is t-linearizable if event t is the response to the first operation: the t-linearization moves the first operation to the end. However, the whole infinite history is not t-linearizable.

Thus, t-linearizability is neither a safety nor a liveness property. However, the property of being t-linearizable for *some* t is trivially a liveness property: for any finite history H, just take t to be equal to the number of events in H, and then any permutation of the operations in H yields a legal, sequential history that is a t-linearization of H. Thus, eventual linearizability is the intersection of a safety property (weak consistency) and a liveness property (being t-linearizable for some t).

3.3 Using Registers to Obtain Weak Consistency

Our definition of eventual linearizability combines a safety property (weak consistency) and a liveness property (being t-linearizable for some t). If our underlying system includes linearizable read/write registers, then we can easily modify any non-blocking implementation that satisfies the liveness property to also satisfy the safety property.

Proposition 11. *Consider a system that includes linearizable registers as base objects. Let T be an object type with finite nondeterminism. There is an eventually linearizable non-blocking implementation of T if and only if there is a non-blocking implementation of T such that for every history there is a t such that the history is t-linearizable.*

This proposition is proved in detail in the appendix. The basic idea is to announce all operations by writing them to shared memory. Then, when a response to an operation has been computed, a process reads all announced operations and returns the response only if it can verify that the response does not violate weak consistency. If a violation of weak consistency is detected, the process is free to return any response that is consistent with just its own prior operations.

4. EVENTUAL LINEARIZABILITY IS WEAK

It is fairly easy to prove that eventually linearizable objects by themselves are useless for implementing linearizable objects. The intuition is that a linearizable implementation must eventually produce the correct output for some operations, but if the eventually linearizable base objects used by the implementation are still behaving badly when that output is produced, that output can be incorrect. The following theorem says that there is no linearizable implementation of any non-trivial object that is useful for inter-process communication from any collection of eventually linearizable objects.

Theorem 12. *For $n \geq 2$, the following are equivalent for any type T with finite non-determinism.*

1. *There is an n-process linearizable wait-free implementation of T using no shared objects.*
2. *There is an n-process linearizable obstruction-free implementation of T using no shared objects.*
3. *There is an n-process linearizable wait-free implementation of T from some collection of eventually linearizable objects.*
4. *There is an n-process linearizable obstruction-free implementation of T from some collection of eventually linearizable objects.*

Proof. It is trivial to see that $1 \Rightarrow 3 \Rightarrow 4$ and $1 \Rightarrow 2 \Rightarrow 4$. So, it remains to prove that $4 \Rightarrow 1$.

Suppose there is an obstruction-free linearizable implementation I of an object of type T from some collection of eventually linearizable objects for n processes $p_1, \ldots p_n$. We construct an n-process wait-free linearizable implementation I' of an object of type T simply by replacing each shared object o by n local copies $o_1, \ldots, o_n$. (Since we assume that transition functions are Turing-computable, a process can simulate the object in its local memory.) Whenever process p_i must perform an operation *op* on shared object o according to I, p_i instead performs *op* on its local copy o_i.

Then, any finite history H of I' is also a possible finite history of I since the eventually linearizable objects used by I can return arbitrary answers (that satisfy weak consistency) in any finite prefix of an execution. (Note that using a local copy of each object ensures the responses satisfy weak consistency.) Thus, every finite history H of I' must be linearizable, since I guarantees linearizability. Since linearizability is a safety property for T (which is assumed to have finite non-determinism), this means that every history of I' is linearizable.

It remains to show that I' is wait-free. Consider any execution α of I' and any process p_i that takes infinitely many steps in α. Since there is no communication between processes in α, p_i cannot distinguish α from $\alpha|p_i$, where p_i runs solo. But $\alpha|p_i$ is also a solo execution of I and I is obstruction-free, so every operation that p_i invokes in this execution must terminate. □

Theorem 12 allows us to characterize the trivial deterministic types that have linearizable implementations from eventually linearizable objects as follows.

Definition 13. *A deterministic type T is called* trivial *if and only if there is a computable function r that maps each initial state q_0 and operation op to a response $r(q_0, op)$ that is the correct response to op for every state reachable from q_0.*

Proposition 14. *A deterministic type T has a linearizable obstruction-free implementation for 2 processes from some collection of eventually linearizable objects if and only if T is trivial.*

Proof. ($\Leftarrow$): Suppose T is trivial. For any initial state q_0, an object of type T initialized to state q_0 can be implemented by having *op* simply return $r(q_0, op)$.

($\Rightarrow$): Suppose there is an obstruction-free linearizable implementation from eventually linearizable objects. By Theorem 12, there is also an obstruction-free linearizable implementation using no shared objects at all for two processes p_1 and p_2. To compute $r(q_0, op)$, we simply run the programme for p_1 that implements *op* for an object initialized to state q_0 until it produces a response. Let α be this solo execution of the implementation. Let q be any state that can be reached from q_0 via some sequence of operations. We must show that $r(q_0, op)$ is the correct response for *op* when an object of type T is in state q. Consider an execution β of the implementation in which process p_2 runs solo, performing a sequence of operations that takes the implemented object from the initial state q_0 to state q. (This uses the assumption that T is deterministic, so that we can force the implemented object into any reachable state.) Since the implementation uses no shared objects, $\beta\alpha$ is a also possible execution of the implementation. Since this execution is linearizable, and p_1 returns $r(q_0, op)$ as the response for *op* in α, there must be a transition $(q, op, r(q_0, op), q')$ in the transition relation that specifies type T. □

In particular, Proposition 14 says that even weak objects like read/write registers do not have linearizable implementations from any collection of eventually linearizable objects.

Eventual Linearizability is Still Weak Even With Linearizable Registers

Theorem 12 tells us that if T is a non-trivial type (i.e., cannot be implemented linearizably without any communication) then a linearizable implementation of T cannot be built from eventually linearizable objects. But what if we also have linearizable registers available? Then, could we build a linearizable object of type T? We use a valency argument [7] to show the answer is still no if T is strong enough to solve two-process consensus.

Proposition 15. *Let T be any object type that can solve wait-free two-process consensus. There is no wait-free linearizable implementation of T from linearizable registers and any collection of eventually linearizable objects (of any types).*

Proof. Suppose there were such an implementation to derive a contradiction. Then we could build a wait-free consensus algorithm for two processes p_0 and p_1 from that same collection of eventually linearizable objects and linearizable registers. Consider the tree of all possible histories of this consensus algorithm in which process p_i has input i for $i = 0, 1$. (In this tree, nodes represent configurations of the algorithm and the edge connecting a parent to a child are labelled by an invocation or response on one of the base objects.) A node C is 0-valent (or 1-valent) if all decisions made in the subtree rooted at C are 0 (or 1, respectively). A node is multivalent if it is neither 0- nor 1-valent. The root of this tree is multivalent, since a solo run by p_i must produce output i. So there must be a critical configuration C (i.e., C is multivalent, but each child of C in the tree is 0- or 1-valent).

Thus, there must be one event s_0 at p_0 and one event s_1 at p_1 that take the system from C to configurations C_0 and C_1 with opposite valencies. If s_0 and s_1 do not involve the same shared object o, then the events commute: the configurations reached from C by doing s_0s_1 or s_1s_0 are indistinguishable to all processes. So, a solo execution by p_0 from either one leads to the same outcome, contradicting the fact that C_0 and C_1 have opposite valencies. We consider three cases.

Case 1 (s_0 is an invocation event on o): Let α be any solo run by p_1 continuing from C_1 until p_1 decides. Then $s_1\alpha$ is also a legal continuation from the configuration C_0, contradicting the fact that C_0 and C_1 have opposite valencies.

Case 2 (s_1 is an invocation event on o): symmetric to Case 1.

Case 3 (s_0 and s_1 are both response events from o): We show s_0s_1 and s_1s_0 could both occur after C.

First, suppose o is a linearizable register. Then, whatever response p_0 gets in event s_0 from configuration C is still a valid response to p_0's pending operation after s_1 occurs. In particular, if s_0 returns a null response to a write by p_0, the write would receive exactly the same response after s_1. Likewise, if s_0 returns a value v as the response to a read by p_0, that value is still a valid response to the read if p_1 gets the response to its operation first. So, s_1s_0 is a valid sequence of events following C. Similarly, s_0s_1 is a valid sequence of events following C.

Now suppose o is an eventually linearizable object, then there is no constraint that the responses from o in any finite prefix of the execution must satisfy (beyond weak consistency). So s_0s_1 and s_1s_0 are both possible sequences of events following C.

Let $s_0s_1\alpha$ be an execution onward from C where α is a solo execution by p_0 until it decides. Then, $s_1s_0\alpha$ is a possible execution from C, contradicting the fact that C_0 and C_1 have opposite valencies. □

Implementing Eventually Linearizable Objects Using Registers

We next consider whether eventually linearizable objects are easier to implement than their linearizable counterparts. Here, we focus on implementations from registers: we show that it is possible to obtain eventually linearizable implementations of some objects that have no linearizable implementations.

Any one-shot type has a trivial eventually linearizable implementation using no shared objects since the implementation may behave badly during the finite prefix when all operations are performed. Similarly, some long-lived types whose behaviour is "interesting" only in a finite prefix of each execution have eventually linearizable implementations using no shared memory. For example, a *test&set object* has an eventually linearizable implementation where each process simply returns 0 for its first invocation of test&set and 1 for all subsequent invocations.

A *consensus object* provides one operation *propose*(v), where v is a value drawn from some domain D. Each *propose* operation returns the value used as the argument of the first *propose* operation to be linearized. This object is essentially the hardest object to implement in a linearizable way (since it can be used to build linearizable implementations of every other type [9]), but it is trivial to implement it in an eventually linearizable way using only linearizable registers. In fact, the following proposition states that such an implementation can be built even from eventually linearizable registers.

Proposition 16. *A consensus object has a wait-free, eventually linearizable implementation from eventually linearizable registers.*

Proof. The implementation for n processes uses an array *Proposals*$[1..n]$ of single-writer multi-reader registers, each initially holding the value $\perp \notin D$. When process p_i invokes Propose(v), it runs the following programme.

```
1 PROPOSE($v$)
2     if $Proposal[i] = \perp$ then $Proposal[i] := v$
3     read $Proposal[1..n]$ and return leftmost non-$\perp$ value
4 end PROPOSE
```

The weak consistency property of the base registers ensures that p_i's first read of *Proposal*$[i]$ returns $\perp$ and all of p_i's subsequent reads of *Proposal*$[i]$ returns the argument of p_i's first Propose operation. Thus, p_i always sees a non-$\perp$ value on line 3, so the leftmost non-$\perp$ value is well-defined. Moreover, if p_i performs a write, then it only performs one write (during its first execution of Propose); all subsequent executions of line 2 must see a non-$\perp$ value in *Proposal*$[i]$.

The implementation is clearly wait-free. Consider any concurrent execution of the implementation. Let H be the history of invocations and responses on the consensus object during this execution and let H' be the history of invocations and responses on the base objects.

We first show that H is weakly consistent. Let *op* be a Propose operation by p_i that terminates in H and returns the value v it read from *Proposal*$[j]$. Then, the

PROPOSE(v) operation op' by process p_j that wrote v must have been invoked before op terminated. Thus, a sequential history that starts with op' can be used to show that the result returned by op satisfies Definition 1 in H.

Our next goal is to define a value of t such that H is t-linearizable. By Lemma 7, there is a t' such that H' is t'-linearizable (since each of the finitely many base registers are eventually linearizable). Let P be the set of PROPOSE operations, all of whose reads terminate after event t' in H' and appear after all writes in the t'-linearization of H'. (Note that there are only finitely many PROPOSE operations that are *not* in P because there are at most n writes during the execution.) Then, all PROPOSE operations in P read exactly the same set of values on line 3 and therefore return the same result v_0.

Choose t such that a PROPOSE(v_0) operation op_0 begins before event t of H and the finitely many terminating PROPOSE operations that are not in P all terminate before event t of H. To create a t-linearization of H, we put op_0 first, followed by all other complete operations, in the order that they terminate. In this sequential history, all operations return v_0, as do all operations in the concurrent history that terminate after step t. □

5. EVENTUAL LINEARIZABILITY CAN BE HARD TO IMPLEMENT

We prove in Proposition 18, below, that if we have an eventually linearizable *implementation* of a fetch&increment object from linearizable base objects, we can "fix" the implementation so that it is linearizable. We use the following technical lemma.

LEMMA 17. *Suppose we have an eventually linearizable implementation of a fetch&increment object. Let $t > 0$ and let α be an infinite history of this implementation. If every finite prefix of α is t-linearizable, then α is t-linearizable.*

PROOF. Since the implementation is eventually linearizable, there is some t' such that α has a t'-linearization S'. If $t' \leq t$, then the claim follows from Lemma 5. So, for the remainder of the proof, we assume $t' > t$.

We partition the operations of α into four sets according to their response events as follows.

- Let A_1 contain the operations whose response is among the first t events of α.
- Let A_2 contain the operations whose response is among events $t+1, \ldots, t'$.
- Let A_3 contain the operations whose response occurs after event t'.
- Let A_4 contain the operations that do not terminate.

Our goal is to construct a t-linearization S of α. We describe how to do this by assigning operations to positions in S. We assume the positions are numbered starting from 0. If an operation in A_2 or A_3 returns a value v, we assign it to position v in S. (Note that this cannot assign two operations to the same position, since if two operations op_1 and op_2 return the same result v, the prefix of α that includes the responses of both op_1 and op_2 would not be t-linearizable.) Let $E = \{v :$ no operation in A_2 or A_3 returns $v\}$. This is the set of slots that have not yet been assigned an operation.

We first show that $|A_1| \leq |E| \leq |A_1| + |A_4|$. Let $E' = \{v :$ no operation in A_3 returns $v\}$. Then, S' fills all positions of E' with operations in A_1 and A_2 and some subset $A'_4 \subseteq A_4$. Thus, $|A_1|+|A_2| \leq |E'| \leq |A_1|+|A_2|+|A_4|$ (and all of these quantities are finite). Each operation of A_3 is assigned the same slot by S and S' (since both are t'-linearizations of α). So, $|E| = |E'| - |A_2|$. Thus, $|A_1| \leq |E| \leq |A_1| + |A_4|$.

For our linearization S, we fill in the first $|A_1|$ slots of E with operations in A_1, in the order they are invoked in α. Then, we fill in the remaining slots with the first $|E| - |A_1| \leq |A_4|$ operations of A_4 in the order of their invocations in α. It remains to prove that S is a t-linearization of α.

- Each operation invoked in S is invoked in α: This follows from the definition of S.
- Each operation that terminates in α appears in S: By definition of S, all operations in $A_1 \cup A_2 \cup A_3$ are assigned slots in S.
- If op_1 terminates after event t and op_2 begins after op_1 terminates and op_2 appears in S, then op_1 precedes op_2 in S: Then, $op_1 \in A_2 \cup A_3$ and $op_2 \in A_2 \cup A_3 \cup A_4$. We consider several cases.

  First, suppose op_2 is in $A_2 \cup A_3$. Then, op_1 must return a smaller value than op_2, since the prefix containing the responses of both op_1 and op_2 is t-linearizable. Thus, op_1 is assigned an earlier slot than op_2 in S.

  Now, suppose op_2 is in A_4. Let v be the response returned by op_1. Consider a prefix of α that contains all response events of α that return values less than or equal to v. By the hypothesis of the lemma, there is a t-linearization of this prefix. In that t-linearization, all the slots of E prior to slot v are filled using operations of A_1 and operations of A_4 that begin before op_1's response (and hence before op_2 is invoked). This means that the number of operations in A_1 and the number of operations in A_4 that begin before op_2 begins is at least $|E \cap \{0, 1, \ldots, v\}|$. In S, all of these operations are linearized before op_2, so op_2 is assigned a slot greater than v. Thus, op_1 precedes op_2 in S.
- Each operation that has a response after event t of α has the same response in S: Each operation of $A_2 \cup A_3$ is assigned to the slot that would cause it to return the same response in S as it does in α.

□

For Lemma 17, the hypothesis that the implementation is eventually linearizable is necessary since t-linearizability is not a safety property of fetch&increment implementations (see Section 3.2).

PROPOSITION 18. *If there is an n-process eventually linearizable, non-blocking implementation of a fetch&increment object from a set $\mathcal{O}$ of linearizable objects, then there is an n-process linearizable, non-blocking implementation of a fetch&increment object from $\mathcal{O}$.*

PROOF. Let A be an n-process eventually linearizable implementation of a fetch&increment object from $\mathcal{O}$. Consider the execution tree of all possible executions of this implementation in which n processes $p_1, \ldots, p_n$ each repeatedly perform fetch&inc operations forever. Each edge in the tree represents either a local event or an atomic action on an object in $\mathcal{O}$.

Consider a node C in this tree. Let α_C be the finite execution represented by the path from the root to C. We say that C is *stable* if *every* execution with prefix α_C is $|\alpha_C|$-linearizable. (In other words, the behaviour of the implemented fetch&increment object has "stabilized" by C.)

It follows from the definition of stable and Lemma 5 that if C is stable, then so are C's descendants.

Claim 1: There is a stable node in the tree.

Proof of Claim 1: To derive a contradiction, suppose no node in the tree is stable. We inductively construct a sequence of finite paths $p_0, p_1, \ldots$ that have the following properties, where $\ell_i = |p_0 p_1 \ldots p_i|$:

1. For $i \geq 0$, $p_0 p_1 \cdots p_i$ is a path starting at the root of the tree.
2. For $i \geq 1$, p_i is non-empty.
3. For $i \geq 1$, $p_0 p_1 \ldots p_i$ is not ℓ_{i-1}-linearizable.

Let p_0 be the empty execution. For the inductive step of the construction, let $i \geq 1$. Given $p_0, \ldots, p_{i-1}$ with the properties listed above, we construct p_i as follows.

Let C be the endpoint of the path $p_0 p_1 \ldots p_{i-1}$. By the assumption, C is not stable. So, there is some execution α with prefix $p_0 p_1 \ldots p_{i-1}$ that is not ℓ_{i-1}-linearizable. By Lemma 17, some finite prefix α' of α is not ℓ_{i-1}-linearizable. That prefix has length greater than ℓ_{i-1}, so let p_i be the non-empty path such that $\alpha' = p_0 p_1 \ldots p_{i-1} p_i$.

Now, consider the infinite execution $\pi = p_0 p_1 \ldots$. Since the implementation is eventually linearizable, there is a t such that this execution is t-linearizable. Choose i such that $\ell_{i-1} > t$. (Such a choice is possible because all of the p_j's are non-empty, so we have $\ell_0 < \ell_1 < \cdots$.) Since π is t-linearizable, it is also ℓ_{i-1} linearizable, by Lemma 5. By Lemma 6, the prefix $p_0 p_1 \ldots p_i$ of π is also ℓ_{i-1} linearizable, which contradicts the third property of the construction. This completes the proof of Claim 1.

Let C be a stable node. Let t be the number of events in the execution α_C. Consider the configuration C_{idle} reached from C by allowing each process to run solo until it completes its current fetch&inc operation. Then, we let one process p run fetch&inc operations repeatedly. We argue that, eventually, some operation op_0 that p performs must return a value that is equal to the number of fetch&inc operations that were invoked before op_0. (If not, there would be no way to t-linearize the infinite execution where p continues forever.) Let C_0 be the configuration at the end of op_0. Let v_0 be the number of fetch&inc operations invoked in the path from the root to C_0.

Now, we construct a linearizable implementation A' of a fetch&increment object using $\mathcal{O}$. Initialize each object in $\mathcal{O}$ to the state it has in C_0. Similarly, each process uses the same local variables as in A, and they are initialized in A' to the values they have in C_0. In A', to perform a fetch&inc operation, a process executes the algorithm A for fetch&inc until it obtains a result v, and then it returns $v - v_0$.

If β is an execution of A', there is a corresponding execution $\alpha\beta'$ of A, where α is the execution described above that takes the system to configuration C_0, and β' is the same as β except that v_0 is not subtracted from the output values. Since C is stable, $\alpha\beta'$ is t-linearizable, and the t-linearization must linearize before op_0 all v_0 operations invoked before op_0. Moreover, the t-linearization respects the real-time order of operations in β' since they are invoked after event t of $\alpha\beta'$. The suffix of the t-linearization obtained by removing the first v_0 operations is a linearization of all operations in β. □

Remark: The preceding proposition can also be proved for other progress conditions: if the eventually linearizable implementation is obstruction-free or wait-free, then it can be used to create an obstruction-free or wait-free linearizable implementation.

Corollary 19. *There is no non-blocking eventually linearizable implementation of a fetch&increment object for two processes from linearizable registers.*

Proof. If there were such an implementation, we could build a linearizable implementation of a fetch&increment object for two processes from linearizable registers, by Proposition 18. This is impossible since a fetch&increment object and registers can solve two-process consensus and registers alone cannot [9, 12]. □

6. OPEN QUESTIONS

We have shown that eventually linearizable implementations can be much easier to build (and weaker) than linearizable ones for some types (e.g., consensus objects). For other types (such as fetch&increment), an eventually linearizable implementation appears to be just as hard to build as a linearizable one. It would be interesting to characterize the exact situations where an eventually linearizable implementation is easier to attain than a linearizable one.

One of the fundamental results about linearizable objects is Herlihy's wait-free universal construction [9]. It is natural to ask whether there is a lock-free universal construction of eventually linearizable objects from some natural eventually linearizable primitive objects (possibly in conjunction with linearizable registers). The results of [5] may provide some ideas for doing this.

APPENDIX: PROOF OF PROPOSITION 11

Proof. The "only if" direction is trivial by the definition of eventually linearizable. It remains to prove the "if" direction. Consider a system of n processes $p_1, \ldots, p_n$. Let A be an implementation of T with the property that for every history there is a t such that the history is t-linearizable. We define a new implementation A' that is eventually linearizable.

In addition to whatever shared objects are used by implementation A, A' uses a sequence of single-writer registers $R_i[0, 1, 2, ...]$ for each process p_i. All of these additional registers are initialized to the value $\perp$. Each process p_i also stores a local counter c_i, initially 0, that counts how many operations p_i has performed, and a local variable q_i (initialized to the initial state of the implemented object) that stores the state of the implemented object that would result if only p_i's own operations were performed on it. In the implementation A', p_i performs an operation op by executing the algorithm in Figure 1.

First, we prove A' is non-blocking. The call to algorithm A on line 5 is non-blocking, by the hypothesis. The

```
1  EXECUTE($op$)
2     write $op$ in $R_i[c_i]$    $\triangleright$ announce $op$
3     $c_i := c_i + 1$
4     $\langle q_i, r_{private} \rangle :=$ a possible new state and result if
             $op$ is applied to object in state $q_i$
             (according to sequential specification)
5     $r_{shared} :=$ result of running $op$ in implementation $A$
6     for $j = 1..n$         $\triangleright$ read all announced operations
7        $k = 0$
8        do until $R_j[k] = \bot$
9           read $R_j[k]$
10          $k := k + 1$
11       end do
12    end for
13    if a permutation of a subset of the operations read
             in the loop (including all read in $R_i$) yields a
             legal sequential execution where $op$ returns
             $r_{shared}$ then return $r_{shared}$
14    else return $r_{private}$
15 end EXECUTE
```

Figure 1: Algorithm to guarantee weak consistency.

only way the loop (line 8–11) can run forever without terminating is if process p_j increments its counter c_j infinitely many times, meaning that p_j completes infinitely many operations. Also, there are only finitely many permutations to try in line 13, since T has finite nondeterminism.

We now show that any history H generated by this implementation is eventually linearizable.

Consider an operation op that terminates in H and is performed by some process p. If op returns on line 14, let $S_{private}$ be the sequential execution defined by the sequence of operations performed by p up to and including op, with the responses chosen on line 4. Then $S_{private}$ satisfies Definition 1 for op in H. If op returns on line 13, then the permutation described on line 13 defines a sequential execution S_{shared} that satisfies Definition 1 for op in H. (Note that all operations included in the permutation begin before op terminates.) Thus, H is weakly consistent.

Next, we prove that H is t-linearizable for some t. Consider the execution that consists of all steps inside the calls to implementation A on line 5. These define a history H_A. By the hypothesis about implementation A, H_A is t_A-linearizable for some t_A. Let S be a t_A-linearization of H_A.

Let O be the set of operations op such that op terminates in H and op's call to A on line 5 terminates within the first t_A events in H_A. Note that O is finite, because only finitely many operations terminate before event t_A in H_A. Choose t large enough that that all operations in O terminate before event t of H. (This is possible because O is a finite set of operations, all of which terminate in H.) We prove that the legal sequential execution S is a t-linearization of H by showing it satisfies the four properties in Definition 2.

Since S is a t_A-linearization of H_A, each operation invoked in S is also invoked in H_A, and is therefore invoked in H.

Each operation that terminates in H also must terminate in H_A, and therefore terminates in S (since S is a t_A-linearization of H_A).

Consider any two operations op_1 and op_2 such that op_1 terminates before op_2 is invoked in H and both of these events occur after event t of H, and op_2 is in S. Let op'_1 and op'_2 be the operations in H_A called at line 5 of op_1 and op_2, respectively. Note that op'_1 exists because op_1 terminates, and op'_2 exists because the operation appears in S, which is a t_A-linearization of H_A. Since op_1 terminates after event t of H, op_1 does not belong to O, so op'_1 terminates after the first t_A events of H_A. Moreover, op_2's call to A on line 5 is after op_2 begins, which is after op_1 terminates, which is after op_1's call to A terminates. Thus, op'_1 terminates before op'_2 begins in H_A. Since S is a t_A-linearization of H_A, op_1 precedes op_2 in S.

Suppose op is an operation that terminates in H after the first t events. We first argue that the test performed by op at line 13 evaluates to true, so that op returns r_{shared} at line 13. Let op_p be any operation that precedes op in S. Let op' and op'_p be the operations in H_A called at line 5 of op and op_p, respectively. Note that op' exists because op terminates, and op'_p exists because it appears in S, which is a t_A-linearization of H_A. Since op terminates after event t of H, op does not belong to O, so op' terminates after the first t_A events of H_A. So, op'_p must begin before op' terminates in H_A (otherwise, op would have to precede op_p in S). This means that op_p begins executing line 5 before op finishes executing line 5. So, op_p executes line 2 before op reaches line 6. Since this is true for every operation op_p that precedes op in S, op will see the announcement of every operation that precedes op in S when it reads the registers. Thus, one possible permutation that op considers in the test on line 13 will be the prefix of S up to op, which yields the response r_{shared} for op, so the test will evaluate to true, and op will return r_{shared}, which is the same value that op returns in S. $\square$

7. REFERENCES

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions and beyond. *Commun. ACM*, 56(5):55–63, May 2013.

[2] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. 41st ACM Symposium on Principles of Programming Languages*, pages 285–296, 2014.

[3] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4), Apr. 2010.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[5] S. Dubois, R. Guerraoui, P. Kouznetsov, F. Petit, and P. Sens. The weakest failure detector for eventual consistency. Unpublished manuscript, 2014.

[6] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Comput. Sci.*, 220(1):113–156, June 1999.

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

[8] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Proc. 2nd International Conference on Networked Systems*, 2014. To appear.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[10] M. Herlihy and N. Shavit. On the nature of progress. In *Proc. 15th International Conference on Principles of Distributed Systems*, pages 313–328, 2011.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[12] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.

[13] N. Lynch. *Distributed Algorithms*, chapter 13. Morgan Kaufmann, 1996.

[14] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, Mar. 2005.

[15] A. G. Sebastien Burckhard and H. Yang. Understanding eventual consistency. Technical report, Microsoft, 2013.

[16] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 95–104, 2010. Full version available from http://labs.yahoo.com/files/aurora.pdf.

[17] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proc. 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 169–184, 2009.

[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 172–182, 1995.

[19] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.