Wait-Free CAS-Based Algorithms: The Burden of the Past

Denis Bédin
Université de Nantes, France

François Lépine
Université de Nantes, France

Achour Mostéfaoui ✉
LS2N, Université de Nantes, France

Damien Perez
Université de Nantes, France

Matthieu Perrin ✉
LS2N, Université de Nantes, France

Abstract

Herlihy proved that CAS is universal in the classical computing system model composed of an a priori known number of processes. This means that CAS can implement, together with reads and writes, any object with a sequential specification. For this, he proposed the first universal construction capable of emulating any data structure. It has recently been proved that CAS is still universal in the infinite arrival computing model, a model where any number of processes can be created on the fly (e.g. multi-threaded systems). In this paper, we prove that CAS does not allow to implement wait-free and linearizable visible objects in the infinite model with a space complexity bounded by the number of active processes (i.e. ones that have operations in progress on this object). This paper also shows that this lower bound is tight, in the sense that this dependency can be made as low as desired (e.g. logarithmic) by proposing a wait-free and linearizable universal construction, using the compare-and-swap operation, whose space complexity in the number of ever issued operations is defined by a parameter that can be linked to any unbounded function.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Process synchronization; Computer systems organization → Multicore architectures; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Compare-And-Swap, Concurrent Object, Infinite arrival model, Linearizability, Memory complexity, Multi-Threaded Systems, Shared-Memory, Universality, Wait-freedom

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.11

Funding This work was partially supported by the French ANR project 16-CE25-0005 O'Browser (`http://obrowser.univ-nantes.fr/`) devoted to the study of decentralized applications on Web browsers.

1 Introduction

Synchronization appeared with the concurrency brought by the first parallel programs in the early sixties. Concurrent accesses to shared data or any physical or logical resource by multiple processes can lead to inconsistencies. Dijkstra introduced the famous mutual-exclusion problem and proposed to solve it using locks [7]. Since then it is still one of the most popular mechanisms for inter-process synchronisation due to its supposed simplicity. The simplest way to implement mutual exclusion on uni-processor systems is by interruption disabling. Interestingly, it turns out that locks can also be implemented using read and write operations on shared variables [8]. However, these implementations have a space

complexity linear with the number of processes [4]. This drawback has been overcome with the introduction of hardware special instructions like compare-and-set, test-and-set, fetch-and-add, etc. These instructions, referred to as read-modify-write instructions, aim to avoid certain interleavings in the execution of the processes by making it possible to read and update a memory location in one atomic operation. They represent, in some sense, a seed of atomicity. The compare-and-set instruction (CAS) is certainly one of the most popular (a.k.a. `compare_exchange_strong` in C++ and `compareAndSet` in Java). It is supported by most modern multiprocessor and multi-core architectures. Informally, the CAS operation has three arguments: the address of a memory location and two values. The memory location is set to the second value if, and only if, the first value is equal to the one stored by the memory location, and a Boolean result (success or failure) is returned to the calling process.

However, locks don't compose and do not tolerate process crashes. If a process holding a lock fails, the whole computation will stuck. Prohibiting the use of locks led to several progress conditions, among which wait-freedom [10] and lock-freedom [12][1]. While wait-freedom guarantees that every operation invoked by a non crashed process terminates after a finite time, lock-freedom guarantees that, if a computation runs for long enough, at least one process makes progress (this may lead some other processes to starve). Wait-freedom is thus stronger than lock-freedom: while lock-freedom is a system-wide progress condition, wait-freedom is a per-process progress condition.

Coordination between processes that access shared resources can be captured as concurrent data structures [2, 6, 11]. The design of the most popular concurrent data structures such as counters, queues, stacks, logs, etc. has been very active these last three decades. Unfortunately, not all data structures admit linearizable wait-free implementations in an asynchronous crash-prone concurrent system that only offers read and write basic operations on variables. This is due to the impossibility to solve the Consensus problem deterministically in this model [13]. The formulation of the Consensus problem is particularly simple. Each process proposes a value and all non-faulty processes decide on the same value among those which are proposed. In contrast, consensus was proved universal in [10]. Namely, any object having a sequential specification has a wait-free linearizable implementation using only read/write basic operations and some number of consensus objects. Moreover, some – but not all – special instructions, such as compare-and-set (CAS) or load-link/store-conditional (LL/SC), are universal as well. Therefore, while special hardware instructions provide efficiency in lock-based computing, they are necessary for lock-free and wait-free computing.

In order to prove the universality of consensus, Herlihy introduced the notion of universal construction. It is a generic algorithm that can emulate any object from its sequential specification. Since then, several universal constructions have been proposed for different special hardware instructions such as CAS and LL/SC [16]. Those are usually designed by first introducing a lock-free universal construction, which is then made wait-free with the use of helping: when a process invokes an operation, it first announces it in a dedicated single-writer variable, and then helps all other announced operations to terminate. Valency-based Helping has recently been proved to be unavoidable for CAS-based implementations of several data structures [5]. Hence, similarly to the space complexity drawbacks described above for the implementations of locks using only read and write operations on shared variables, many wait-free universal constructions have a space complexity linear in the number of potential participating processes. This inefficiency concerns both lock-free and wait-free implementations and is related to the use of historyless objects such as registers, LL/SC, CAS,

[1] In [12] lock-freedom is called non-blocking.

and TAS for example. It has been proved in [9] that the minimal space complexity of the implementations that use only historyless objects is linear with the number of participating processes.

In 2000, Merritt and Taubenfeld [14] introduced the infinite arrival model to deal with computing systems composed of an unbounded number of processes unlike the classical model composed of a fixed and a priori known number of processes. This model includes among others the multi-threaded model where any number of threads can be created and started at run-time and may leave or crash. So although the number of processes at each time instant is finite, it is not a priori known and there is no bound on the total number of threads that can participate in long-running executions. Recently, the universality of consensus and CAS has been extended to the infinite arrival model [15] by proposing a universal construction. In the proposed construction, helping is managed by an announcement data structure in which newly arrived processes can safely insert their operation. Unfortunately, terminated operations cannot be removed, resulting in an ever-growing data structure whose size depends on the total number of ever issued operations.

Problem Statement. This paper explores the performance aspect of the synchronization based on the CAS hardware special instruction. More precisely, we ask the following question: *Is it possible to design a wait-free universal construction whose space complexity only depends on the number of operations in progress?*.

Contribution 1: We prove that the answer is negative when only read, write and compare-and-set operations are available. This means that the space complexity depends on the total number of processes that ever issued operations, and that complex data structures must be maintained, and traversed, to implement helping mechanisms.

Contribution 2: Conversely, we show that our lower bound is tight, in the sense that this dependency can be made as low as desired (e.g. logarithmic), as long as it remains unbounded. We present a wait-free and linearizable universal construction, using the compare-and-set operation, whose space complexity in the number of ever issued operations is defined by a parameter that can be linked to any unbounded function. Obviously, this low spatial complexity is obtained to the detriment of the time complexity.

Let us note that lock-free linearizable implementations can trivially have a constant space complexity when no operation is in progress by simply having a CAS in a loop.

Organization of the paper. The remainder of this paper is organized as follows. In Section 2, we present the computing model that we consider and we define some notions that will be used afterwards. Then, Sections 3 and 4 respectively present the lower and upper bounds on the space complexity of wait-free and linearizable CAS-based algorithms. Finally, Section 5 concludes the paper.

2 Model

This paper considers the infinite arrival model [14] composed of a countable set Π of asynchronous sequential processes $p_0, p_1, \ldots$ that have access to a shared memory. The set Π is the set of potential processes that may join, get started and crash or leave during a given execution. At any time, the number of processes that have already joined the system is finite, but can be infinitely growing in long-running executions. Each process p_i has a unique identifier i that may appear in its code.

2.1 Communication between processes

Processes have access to local memory for local computations and have also access to a shared memory to communicate and synchronize. The shared memory is composed of an infinite number of unbounded locations, called registers [2]. Processes have access to a dynamic memory allocation mechanism accessible through the syntax **new** T, that instantiates an object of type T (T may be `Reg` to allocate a single register, as well as a record datatype or an array) and returns its reference, i.e. it allocates the memory locations needed to manage the object and initializes them by calling a constructor. Processes are not limited in the number of locations they can access, nor by the number of times they can use the allocation mechanism, during an execution. However, they can only access memory locations that either 1) have been allocated at the system set up, or 2) are returned by the allocation mechanism, or 3) are accessible by following references stored (as integer values) in some accessible memory location. In other words, when a process p_i allocates memory locations at runtime, they can initially only be accessed by p_i until it manages to share a reference pointing to these new memory locations. We say that a memory location is *reachable* when it can be accessed by a newly arrived process. When a memory location becomes inaccessible by any process in the system, it is automatically de-allocated by a garbage collector mechanism.

Processes can read the value of a shared register x by invoking x.`read`(), and can write a value v in x by invoking x.`write`(v). Moreover, as some objects cannot be implemented using only read/write operations, the system is enriched with the special atomic instruction `CAS`. Reads, writes and `CAS` are atomic in the sense that the different executions of the calls to these operations are totally ordered.

The compare-and-set instruction can be invoked on a register x with the expression x.`CAS`(*expect*, *update*), which returns a Boolean value. In the execution, the value stored in x is first compared to *expect*. If they are equal, then *update* is written in the register and **true** is returned. Otherwise, the state is left unchanged and **false** is returned.

2.2 Concurrent executions

An execution α is a (finite or infinite) sequence of steps, each taken by a process of Π. A step of a process corresponds to the execution of a read, a write, or `CAS`. Processes are asynchronous, in the sense that there is no constraint on which process takes each step: a process may take an unbounded number of consecutive steps, or wait an unbounded but finite number of other processes' steps between two of its own steps. This makes it possible to abstract from the difference in load of the different processors (cores) and from the fact that access to a processor is controlled by a scheduler. Moreover, it is possible that a process stops taking steps at some point in the execution, in which case we say this process has *crashed*, or even that a process takes no step during a whole execution (Π is only a set of potential participants). We say that a process p_i *arrives* in an execution at the time of its first step during this execution. Remark that, although the number of processes in an execution may be infinite in the infinite arrival model, the number of processes that have already arrived into the system at any step is finite.

[2] The assumption of an infinite memory, also made in the definition of Turing Machines, abstracts the fact that modern memories are large enough for all applications we consider in this paper and allows for simpler reasoning. The assumption of unbounded memory locations is then necessary to store references as memory addresses of an infinite memory are unbounded. The reader can check that we do not use these assumptions in any unpractical way.

A configuration C is composed of the local state of each process in Π and the value of each location in the shared memory. For a finite execution α, we denote by $C(\alpha)$ the configuration obtained at the end of α. An empty execution is denoted ε. An execution β is an extension of α if α is a prefix of β.

2.3 Implementation of shared objects

An implementation of a shared object is an algorithm divided into a set of sub-algorithms, one for the initialization (a.k.a. the constructor of the object), and one for each operation of the object, that produces wait-free and linearizable executions.

▶ **Definition 1** (Linearizability). *An execution α is* linearizable *if all operations have the same effect and return the same value as if they occurred instantly at some point of the timeline, called the* linearization point*, between their invocation and their response, possibly after removing some non-terminated operations.*

▶ **Definition 2** (Wait-freedom). *An execution α is* wait-free *if no operation takes an infinite number of steps in α.*

In this paper, we are interested in the space complexity of implementations. We distinguish the space complexity necessary to processes during the execution of their operations (e.g. their local memory and the memory locations that will be garbage-collected at the end of their execution), and the long-lasting space requirements of the data structures necessary to store the metadata of the algorithm, and that remains allocated even after all processes have terminated their operations. More precisely, we aim at minimizing the *quiescent complexity*, that measures the memory space required to store the state of a shared object when no process is executing an operation on it. This is to make sure we do not count the local storage of processes, which is not meaningful since the number of processes is unbounded.

▶ **Definition 3** (Quiescent complexity). *Let A be an algorithm. A finite execution α of A is said to be n-*quiescent *if exactly n operations of A were invoked, and all of them are completed, in $C(\alpha)$.*

The quiescent *complexity of A is the function $QC : \mathbb{N} \mapsto \mathbb{N} \cup \{\infty\}$, where $QC(n)$ is the maximal number of memory locations reachable in some configuration $C(\alpha)$ obtained at the end of any n-quiescent execution α, if this maximum exists, and ∞ otherwise.*

As explained in the Introduction, a universal construction is a generic algorithm, parametrized by the specification of a shared object, called a *state machine*, and that emulates a wait-free, linearizable shared version of the state machine. In this paper, the sequential specification of a state machine is defined as a transition system, whose initial state is the constant `initialState`, and whose transitions are defined by a function `execute` that takes as arguments a state of the object and an operation, and returns a pair formed by the resulting state and the return value obtained when the input operation is executed sequentially on the input state. For example, the fact that the `dequeue` operation on a non-empty queue deletes the first element from the queue and returns it is specified as $\texttt{execute}([x_0, x_1, \ldots, x_n], \texttt{dequeue}) = \langle [x_1, \ldots, x_n], x_0 \rangle$.

3 Lower Bound on Universal constructions using Compare-And-Swap

This section explores the limitations of wait-free linearizable universal constructions based on compare-and-set. More precisely, Theorem 11 proves that there is no such construction with constant quiescent complexity, i.e. any such construction must maintain a data structure that

may grow over time. Let us first introduce the notions of mute process and visible object. We call *mute process* (Definition 4) a process that lost all its attempts at compare-and-set, and all the values it wrote in shared variables were overwritten before they could be read by another process. The class of *visible objects*, as defined in [9], is "a class that includes all objects that support some operation that must perform a visible write before it terminates. This class includes many useful objects (counters, stacks, queues, swap, fetch-and-add, and single-writer snapshot objects)". Intuitively, any update operation on any visible object (Lemma 5 considers a linearizable counter for this matter) must modify the global state of the system, in order to have an impact on the value returned by subsequent reads. For CAS-based algorithms, it implies that the presence of mute processes cannot be known by any other process, so they cannot complete their update operations on their own, nor can they be helped by others.

▶ **Definition 4** (mutism). *Let α be a finite execution, and let p be a process. We say that p is* mute *in α if there exists an execution α' such that p did not participate in α' and, for all processes $p' \neq p$, all shared variables and the local state of p' are the same in $C(\alpha)$ and $C(\alpha')$, i.e. $C(\alpha)$ and $C(\alpha')$ are indistinguishable*[3] *to p'.*

A finite execution α is said to be mute *if there exists a mute process p that terminated its execution in $C(\alpha)$.*

▶ **Lemma 5.** *Let A be a wait-free linearizable implementation of a counter (i.e. containing one operation,* `increment`*, that returns the number of previous invocations to* `increment`*). Then A does not have a mute execution.*

Proof. Let A be a wait-free linearizable implementation of a counter. Suppose (by contradiction) that A has a mute execution α, and let n be the number of processes that participate in α.

By definition, there exists a process p and an execution α' such that p terminated its execution in $C(\alpha)$, p did not participate in α' and, for all processes $p' \neq p$, $C(\alpha)$ and $C(\alpha')$ are indistinguishable to p'.

Let us consider the extension $\alpha\beta\gamma$ of α such that in β, all processes p' that took steps in α terminate their invocation, and in γ, some process q that did not participate in α joined and completed an invocation of `increment` in isolation, getting n as a result (β and γ exist because A is wait-free). As α and α' are indistinguishable to all processes p' and to q, $\alpha'\beta\gamma$ is also a valid execution, in which q also gets n as a result. However, only $n-1$ invocations of `increment` were started before q terminated, so A is not linearizable. ◀

In an algorithm A with a constant quiescent complexity, a bounded number of memory locations may remain reachable forever after a certain point in time (Definition 6 calls them static), and other memory locations may be allocated by some operations, and later be made unreachable by the same or another operation (Definition 6 calls them dynamic). Lemma 10 builds an execution in which some process p remains mute forever because, whenever p covers a static location x (i.e. p is about to write in x, see Definition 7), some other process also covers x and wins the competition, and whenever p covers a dynamic location, this location is made unreachable, so p_i's write remains unnoticed.

[3] This means that process p has no way to distinguish the two configurations. As said in [1] "Lack of knowledge about other components can formally be captured through the concept of indistinguishability, namely inability to tell apart different behaviors or states of the environment. Indistinguishability is therefore a consequence of the fact that computer systems are built of individual components, each with its own perspective of the system".

▶ **Definition 6** (static vs dynamic locations). *Let α be a finite execution, and x be a shared memory location that is reachable in $C(\alpha)$. We say that x is* dynamic *in α if there exists an extension $\alpha\beta$ of α such that 1) x is unreachable in $C(\alpha\beta)$, 2) no process that is mute in α takes steps in β, and 3) all processes are either mute in $C(\alpha)$ or have terminated their execution in $C(\alpha\beta)$. We say that x is* static *in α if it is not dynamic in α.*

▶ **Definition 7** (covering). *Let α be a finite execution, p a process and x a shared variable. We say that p* write-covers *(resp.* CAS-covers*) x in $C(\alpha)$ if the next step of p in $C(\alpha)$ is a write (resp an invocation of $\texttt{CAS}(e, u)$ with $e \neq u$) on x. We say that p* covers *x in $C(\alpha)$ if p write-covers or CAS-covers x in $C(\alpha)$.*

In order to simplify the proofs, we only consider, in Lemma 10, algorithms that follow a normal form, defined in Definition 8. This assumption is done without loss of generality, since the proof of Lemma 9 discusses how to normalize any algorithm.

▶ **Definition 8** (Normal form). *An algorithm A is said to be in* normal form *if it satisfies the following properties.*

1. *There exists a location* last *such that the last step of any process is a write in* last*, that is never accessed otherwise.*
2. *Each time a process p invokes $x.\texttt{CAS}(e, u)$, its previous step is a read of x that returned e.*
3. *All values written, or proposed as the second argument of compare-and-set, are different.*

▶ **Lemma 9.** *Any wait-free linearizable implementation A of a counter in the infinite arrival model, with a constant quiescent complexity and that only uses read, write and compare-and-set operations can be converted into a wait-free linearizable implementation A' of a counter in normal form with a constant quiescent complexity.*

Proof. We transform A into an algorithm A' in normal form as follows. First, we add an integer shared variable last initialized to 0 (if A already contains a variable named last, this variable is renamed in A'). We also add a concluding step in which all processes write their identifier in last.

Then, we replace all shared registers by a modified register whose type is defined by Algorithm 1, keeping the same invocations to read, write and compare-and-set. To comply with the third property of the definition of a normal form, Algorithm 1 associates a unique timestamp with each value proposed to write and `CAS` operations, consisting of a sequence number `time` and a process identifier `pid`. For that, each process locally numbers its different write and `CAS` operations using a local variable cl_i and, since the different processes have unique identifiers, no confusion can occur between the timestamps forged by different processes. During a read operation, the timestamp is removed and only the value relevant to the object is returned by the read. Hence, Algorithm 1 uses one shared register, storing values from a structured type containing three fields: a field value storing the value relevant to A, an integer field `time` and an integer field `pid`.

Remark that Algorithm 1 is itself a wait-free and linearizable implementation of a shared register (using each time the last operation on internal as linearization point), so the algorithm A' verifies all liveness and safety properties proved on A. In particular, A' is also a wait-free linearizable implementation of a counter. Moreover, A' also has a constant quiescent complexity since we only added one static memory location last, and Algorithm 1 multiplies the size of all used memory locations by a constant factor. ◀

▶ **Lemma 10.** *All wait-free algorithms in normal form with constant quiescent complexity have a mute execution.*

Algorithm 1 Normalisation of shared registers : code for p_i.

```
1  constructor (initial) is
2      internal ← new Reg({value ← initial, time ← 0, pid ← 0});
3  operation read() is
4      return internal.read().value;
5  operation write(v) is
6      v' ← {value ← v, time ← cl_i, pid ← i};
7      cl_i ← cl_i + 1;
8      internal.write(v');
9  operation CAS(e, u) is
10     e' ← internal.read();
11     if e'.value ≠ e then return false;
12     u' ← {value ← u, time ← cl_i, pid ← i};
13     cl_i ← cl_i + 1;
14     return internal.CAS(e', u');
```

Proof. Let A be a wait-free algorithm in normal form, and let us suppose there is a tight bound k on the quiescent complexity of A. We prove, by induction on i, the following claim for all $i \in \{0, \ldots, k\}$.

▶ **Predicate 1** ($P_1(i)$)**.** *For all finite executions α, there exists an extension $\alpha\beta$ of α in which no process participates in both α and β, at least i static locations are covered by mute processes in $C(\alpha\beta)$ that didn't participate in α, and all non-mute processes that took part in β have terminated their execution.*

Proof. The empty execution works for $P_1(0)$, as it concerns no location and no process. Suppose we have proved $P_1(i)$ for some $i < k$. We now prove, by induction on j, the following claim for all $j \in \{0, \ldots, i+1\}$.

▶ **Predicate 2** ($P_2(i, j)$)**.** *For all finite executions α, there exists an extension $\alpha\beta$ of α in which in which no process participates in both α and β, and either (1) at least $i+1$ static locations are covered in $C(\alpha\beta)$ by mute processes that did not participate in α, or (2) at least j static locations are write-covered in $C(\alpha\beta)$ by mute processes that did not participate in α, and all non-mute processes that took part in β have terminated their execution.*

Proof. Predicate $P_2(i, 0)$ is implied by $P_1(i)$. Suppose we have proved $P_2(i, j)$ for some $j \leq i$. We suppose (by contradiction), that $P_2(i, j+1)$ does not hold.

Let $p \in \Pi$ be a process that did not take any step in α. We build, inductively, a sequence $(\alpha_m)_{m\in\mathbb{N}}$ of executions such that $\alpha_0 = \alpha$, for all $m \in \mathbb{N}$, p is mute in α_m, $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$ is an extension of α_m, and if $m > 0$, p takes at least one step in $\beta_m\gamma_m\delta_m$, a different set of processes participate in each extension $\beta_m\gamma_m\delta_m$, and all processes $q \neq p$ that take a step in $\beta_m\gamma_m\delta_m$ are terminated immediately after taking their step. Suppose we have built α_m for some $m \in \mathbb{N}$. In β_m, p takes steps in isolation until it covers some reachable location x. As A is wait-free, either such a situation is bound to happen or p is mute, which concludes the proof of $P_2(i, j)$. Three cases are to be distinguished.

- Suppose x is a dynamic location in $\alpha_m\beta_m$. There exists an extension $\alpha_m\beta_m\gamma_m$ of $\alpha_m\beta_m$ in which x is not reachable, and only mute processes or processes that have terminated their execution know the existence of x. Let $\alpha_m\beta_m\gamma_m\delta_m$ be the extension of $\alpha_m\beta_m\gamma_m$ in which p takes one step.

- Otherwise, x is a static location. Suppose p write-covers x in $C(\alpha_m\beta_m)$. Let $\alpha_m\beta_m\gamma_m$ be the extension of $\alpha_m\beta_m$ provided by Predicate $P_2(i,j)$. As we supposed that $P_2(i,j+1)$ does not hold, the only possibility is that at most j static locations are write-covered in $C(\alpha_m\beta_m\gamma_m)$, by mute processes that did not participate in $\alpha_m\beta_m$, including x, that is write-covered by some process q. In δ_m, p first takes one step, writing in x, and then q completes its execution, overwriting p's write. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.
- Otherwise, x is a static location and p's next step in $C(\alpha_m\beta_m)$ is $x.\texttt{CAS}(e,u)$, with $e \neq u$ since A is in normal form. Let $\alpha_m\beta_m\gamma_m$ be the extension of $\alpha_m\beta_m$ provided by Predicate $P_1(i)$. As we supposed that $P_2(i,j+1)$ does not hold, the only possibility is that at least i static locations are covered in $C(\alpha_m\beta_m\gamma_m)$, by mute processes that did not participate in $\alpha_m\beta_m$, including x, that is covered by some process q. If q writes-covers x, we build δ_m as previously. Otherwise, q's next step in $C(\alpha_m\beta_m\gamma_m)$ is $x.\texttt{CAS}(e',u')$, with $e' \neq u' \neq e$, by property (3) of the normal form, since A is in normal form.
  Let $\bar{x}$ be the value stored in x in Configuration $C(\alpha_m\beta_m\gamma_m)$. If $\bar{x} = e'$, in δ_m, q first takes one step, writing in u' in x, then p takes one step, that does not change the value of x and returns **false** ($u' \neq e$), and finally q terminates its execution. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.
  Otherwise, as A is in normal form, $\bar{x}$ was written in x after q read e' during γ_m, which occurred after p read e during β_m, so $e \neq \bar{x}$. In δ_m, then p takes one step, that does not change the value of x and returns **false**. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.

Finally, p takes an infinite number of steps in $\alpha\beta_1\gamma_1\delta_1\beta_2\gamma_2\delta_2\ldots$ without terminating, which contradicts the fact that A is wait-free. This terminates the proof of $P_2(i,j)$ for all $j \in \{0,\ldots,i+1\}$. ◀

Let us come back to the proof of Predicate $P_1(i+1)$. By $P_2(i,i+1)$, there exists an extension $\alpha\beta$ of α in which at least $i+1$ static locations are covered in $C(\alpha\beta)$, i.e. $P_1(i+1)$ is true. This terminates the proof of $P_1(i)$ for all $i \in \{0,k\}$. ◀

Finally, by invoking $P_1(k)$ twice, there exists an extension α of the empty execution ε in which k static locations are covered in $C(\alpha)$, and an extension $\alpha\beta$ of α in which k static locations are covered in $C(\alpha\beta)$ by processes that did not participate in α.

Let p be a process that did not participate in $\alpha\beta$. As all processes that participate in $\alpha\beta$ are either mute or have terminated their execution in $C(\alpha\beta)$, there exists an execution γ such that $C(\alpha\beta)$ and $C(\gamma)$ are indistinguishable to p, and no process is executing A in $C(\gamma)$. By definition of k, at most k locations are reachable in $C(\gamma)$, so at most k locations are reachable in $C(\alpha\beta)$ as well. By items items 2 and 3 of Definition 6, all k are static.

Therefore, all k locations are covered at least twice by mute processes in $C(\alpha\beta)$. In particular, there are two mute processes q and r that are about to write in last. Let us pose δ the sequence of steps in which q writes in last and then completes its invocation, and then p writes in last. Process q is mute in $\alpha\beta\delta$ and q terminates its execution, so $\alpha\beta\delta$ is a mute execution of A. ◀

▶ **Theorem 11.** *There is no wait-free linearizable implementation of a counter with a constant quiescent complexity in the infinite arrival model, that only uses read, write and compare-and-set operations.*

Proof. Suppose there is a wait-free linearizable implementation A of a counter with a constant quiescent complexity. In particular, by Lemma 9, we can suppose without loss of generality that it is in normal form. By Lemma 10, A has a mute execution, and by Lemma 5, A does not have a mute execution. This is a contradiction, so A does not exist. ◀

4 Upper Bound on Universal constructions using Compare-And-Swap

From Theorem 11, we can derive that the quiescent complexity of any wait-free linearizable universal construction is in $\omega(1)$. Differently, [3] presents such a construction with a quiescent complexity in $\mathcal{O}(n)$. The present section closes the gap thanks to Algorithm 2, a wait-free and linearizable universal construction that is parametrized by any unbounded and monotonically increasing function $f : \mathbb{N} \mapsto \mathbb{R}$ (e.g. log or $\log^\star$), and whose quiescent consistency is $QC(n) = \mathcal{O}(f(n))$. In the remainder of this section, let us fix an unbounded and monotonically increasing function f, and let us define its inverse f^{-1} as follows: for all $x \in \mathbb{N}$, $f^{-1}(x)$ is the smallest $y \in \mathbb{N}$ such that $f(y) \geq x$.

In Algorithm 2, a new operation is linearized each time a compare-and-set is won on a shared register linearization. In order to require the help from other processes, each operation starts by installing itself into a memory location that was at the head of a linked list announces when it started the algorithm. Once a process has failed too many times (depending on f) to install its operation, it changes the head of the linked list, which guarantees it not to lose again against any new operation.

Algorithm 2 maintains a data structure depicted on Figure 1, and composed of three kinds of nodes, described thereafter as structured data types.

- The first kind of nodes, called *operation node* and of type `ONode`, represents an ongoing operation. An operation node o is composed of three fields: o.oper is an operation of the state machine, o.result is a register storing either $\perp$ or a value that can be returned by o.oper, and o.done is a Boolean register. An operation node o is created when an operation o.oper is invoked by a process p_i on the state machine, initially with o.result $= \perp$ and o.done $=$ **false** (Line 8). After the operation has been linearized, o.done is set to **true** by some process p_j (possibly different to p_i) (Line 27), which serves as a signal to p_i that it can return o.result (Lines 16-17).
- The role of an *announce node* of type `ANode` is to expose a memory location in which a process can install an operation node, so that other processes can help completing the operation. An announce node a is either the empty node $\perp_a$ or a structure of two fields: a.next references another announce node and a.o is a register that references an operation node. In other words, an announce node is part of a linked list ending with $\perp_a$. We define the rank `rank`(a) of a node a as the length of the linked list, i.e. `rank`$(\perp_a) = 0$ and `rank`$(a) =$ `rank`$(a$.next$) + 1$ if $a \neq \perp_a$. The number of announce nodes accessible at the end of quiescent executions can only grow, which determines the quiescent complexity of Algorithm 2.
- Finally, a *linearization node* of type `LNode` represents a possible state of the state machine, as well as some information concerning the last operation leading to this state. A linearization node l is composed of three fields: l.state is a state of the state machine, l.result is a value returned by an operation of the state machine and s.o references an operation node. The sequence of states visited during an execution corresponds to a sequence of successful compare-and-set operations on linearization nodes.

Processes share two variables. The first one, announces, is a register that references announce nodes and is initialized to an announce node of rank one. The linked list of announce nodes accessible through announces provides a set of memory locations in which operation nodes can be placed to allow communication between processes that need helping and processes willing to help. The second variable, linearization, is a register that references a linearization node and is initialized to a new linearization node referencing the initial state of the state machine and a reference to a new dummy operation node. Later, linearization is composed of the current state of the state machine, as well as the operation node of the last linearized operation and its return value.

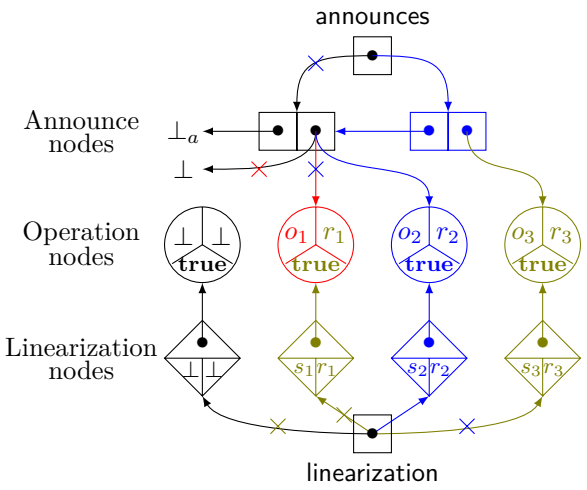

Figure 1 An execution of Algorithm 2, with $f(1) = 1$. The initial state is represented in black. Processes p_1 (in red), p_2 (in blue) and p_3 (in green) attempt to concurrently execute o_1, o_2 and o_3, respectively. Initially, p_1 and p_2 read the same announce node a, and p_1 wins the first compare-and-set, so p_2 creates a new announce node a' to prevent concurrency of newly arrived processes. Indeed, p_3 reads a' and writes its own operation node in it, then linearizes o_1 and o_3 and terminates. Finally, p_2 wins the compare-and-set on a and linearizes o_2.

When a process p_i needs to apply an operation op_i on the state machine, it invokes $\texttt{invoke}(op_i)$ on the universal construction. Process p_i first creates an operation node o_i containing its operation (Line 8), and then strives at installing o_i at the head a_i of the list of announce nodes referenced by announces, using compare-and-set (Line 18), after helping operation nodes already announced to be linearized by calling $\texttt{help}(a_i)$ (Line 14). If p_i fails to write o_i into $a_i.\texttt{o}$ $f^{-1}(\texttt{rank}(a_i) + 1)$ times, it tries to insert a new announce node at the head of the announces list (Lines 11 and 13) to prevent newly arrived processes to compete on $a_i.\texttt{o}$, and ensure its own termination. Remark that p_i can only fail if some other process succeeded in inserting another announce node, providing the same benefits.

When p_i executes $\texttt{help}(a_i)$ to linearize the operation $a_i.\texttt{o.oper}$ of the operation node $a_i.\texttt{o}$, it first helps recursively all announce nodes reachable from a_i (Line 21), and then tries to replace the linearization node in linearization using compare-and-set, until success (Lines 25 to 31). Remark that the new state of the state machine, as well as the value returned by an operation, are computed (Line 29), before the linearization node referencing the operation is created, and the return value is later reported on the operation node (Line 26), possibly by still a different process.

▶ **Lemma 12.** *No call to* $\texttt{help}(a_i)$ *in Algorithm 2 takes an infinite number of steps.*

Proof. Suppose, by contradiction, that some call to $\texttt{help}(a_i)$ by a process p_i takes an infinite number of steps. Without loss of generality, we can suppose that $r_i = \texttt{rank}(a_i)$ is minimal. Let o_i be the operation node read by p_i on Line 22. By Line 28, $o_i.\texttt{done}$ is always **false**, so no process ever wins the compare-and-set on Line 31 with a linearization node referencing o_i (otherwise, the next process that writes linearization on Line 31 would previously have set $o_i.\texttt{done}$ to **true** on Line 27), and $a_i.\texttt{o} = o_i$ at all time after some point.

In only a finite number K of invocations of $\texttt{invoke}(op_j)$ by some process p_j, all done before the invocation of $\texttt{help}(a_i)$ by p_i, p_j reads an announce node a_j with $\texttt{rank}(a_j) < r_i$. All of them terminate because 1) by minimality of r_i, $\texttt{help}(a_k)$ terminates on Line 15 and 2)

Algorithm 2 Universal construction using compare-and-set.

```
1  constructor (initialState) is
2      a_0 ← new ANode {next ← ⊥_a, o ← new Reg(⊥)};
3      o_0 ← new ONode {oper ← ⊥, result ← new Reg(⊥), done ← new Reg(true)};
4      l_0 ← new LNode {state ← initialState, result ← ⊥, o ← o_0};
5      announces ← new Reg(a_0) ;
6      linearization ← new Reg(l_0) ;
7  operation invoke(op_i) is
8      o_i ← new ONode {oper ← op_i, result ← new Reg(⊥), done ← new Reg(false)};
9      a_i ← announces.read();
10     for k ← 0, 1, 2, ... do
11         if k = f^{-1}(rank(a_i) + 1) then
12             a'_i ← new ANode {next ← a_i, o ← new Reg(⊥)};
13             announces.CAS(a_i, a'_i);
14         o'_i ← a_i.o.read();
15         help(a_i);
16         if o_i.done.read() then
17             return o_i.result.read();
18         a_i.o.CAS(o'_i, o_i);
19 function help(a_i) is
20     if a_i = ⊥_a then return;
21     help(a_i.next);
22     o_i ← a_i.o.read();
23     if o_i = ⊥ then return;
24     while true do
25         l_i ← linearization.read();
26         l_i.o.result.write(l_i.result);
27         l_i.o.done.write(true);
28         if o_i.done.read() then return;
29         ⟨s_i, r_i⟩ ← execute(l_i.state, o_i.oper);
30         l'_i ← new LNode {state ← s_i, result ← r_i, o ← o_i};
31         linearization.CAS(l_i, l'_i);
```

whenever a process wins a compare-and-set on Line 18, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so p_j can only be prevented to terminate K times. After that, a_j.o is never writen, and a_j.o.done remains **true** forever. In particular, no further invocation of help(a_j) executes Line 31, as they terminate on Line 28.

After that point, all new invocations of help(a_j) by some process p_j on Line 15 are such that rank(a_j) $\geq r_i$. Thanks to Line 21, and by what preceeds, p_j's first execution of Line 31 is during its recursive call help(a_i), in which p_j reads a_i.o $= o_i$ on Line 22. As all executions of Line 31 try to write $l_j = \langle s, r, o_i \rangle$ for some s and r, only one of them succeeds. After that point, some process (possibly p_i) reads l_j on Line 25 and writes **true** in o_i.done on Line 27, which is a contradition. ◀

▶ **Lemma 13** (Wait-freedom). *Algorithm 2 is wait-free.*

Proof. Let us consider an invocation of `invoke`(op_i) by a process p_i. By Lemma 12, the function `help` terminates, so all iterations of the loop by p_i terminate as well. Let $K = f^{-1}(\texttt{rank}(a_i) + 1)$. As f is unbounded, K is well defined.

If p_i iterates less than K times, then it terminates its execution. Otherwise, it executes Line 13 when $k = K$, and whether the compare-and-set is successful or not, announces $\neq a_i$ after that. All processes that arrive later read a different value on Line 9, so only a finite number of processes compete with p_i on Line 18. Each time one of them succeeds, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so an operation can only prevent p_i to win its compare-and-set once. Therefore, p_i eventually terminates its execution. ◀

▶ **Lemma 14** (Linearizability). *All executions admitted by Algorithm 2 are linearizable.*

Proof. Let α be an execution admissible by Algorithm 2.

Let us first remark that, for any operation `invoke`(op_i) invoked by process p_i, at most one linearization node l_i such that l_i.o.oper $= op_i$ is such that an invocation of linearization.`CAS`(l', l_i) returns **true** on Line 31. Indeed, first remark that all linearization nodes written in linearization are unique, because they are created on the same line as they are written, and that only one operation node o_i, built on Line 8 by p_i, is such that o_i.oper $= op_i$. Suppose (by contradiction) that two linearization nodes l_j and l_k, with l_i.o $= l'_i$.o $= o_i$, were successfully written in linearization by p_j and p_k respectively. Let us consider, without loss of generality, the first two such linearization nodes, and let us consider the linearization node l_m that overwrote l_i i.e. such that the invocation linearization.`CAS`(l_i, l_m) by some process p_m returned **true**. Process p_j read l in linearization on Line 25 before **false** in o_i.done on Line 28, before p_m wrote **true** in o_i.done on Line 27, before p_m invoked linearization.`CAS`(l_i, l_m) on Line 31. Therefore, l is at least as old as l_i. It is impossible that $l = l_i$ because it would mean $p_j = p_m$ would have executed Line 28 before Line 27, and it is impossible that l is older than l_i because it would have been overwritten by l_i or before.

Let us define the linearization point of any operation `invoke`(op_i) as, if it exists, the unique successful invocation of linearization.`CAS`(l', l_i) such that l_i.o.oper $= op_i$.

We now prove that any operation `invoke`(op_i) done by a terminating process p_i has a linearization point, between its invocation and termination point. As p_i terminated, it read **true** in o_i.done on Line 16, so some process p_j wrote **true** in l_i.o.done $= o_i$.done on Line 28, after having read l_i on Line 25, which can only happen after some process p_k wrote l_i on Line 31. This is a linearization point for `invoke`(op_i). As we have seen, the linearization point happened before the p_i's termination. It also happened after p_i's invocation, as o_i can only be created by p_i on Line 8.

Finally, let us remark that, thanks to Line 29, the states and result values reached in a sequential execution E defined by the linearization order are the same as the ones written in the linearization nodes on Line 31. If Process p_i returns r_i at the end of the execution of `invoke`(op_i), it read it in o_i.result.`read`() on Line 17, after reading **true** in o_i.done on Line 16, which can only happen if some process wrote **true** in o_i.done on Line 27 after writing l_i.result in o_i.result on Line 26, with l_i.o $= o_i$ and l_i read in linearization on Line 25. Therefore, the p_i returns the same value as in H.

In conclusion, α is linearizable. ◀

▶ **Lemma 15** (Complexity). *The quiescent complexity of Algorithm 2 is $QC(n) = \mathcal{O}(f(n))$.*

Proof. Let α be a finite execution of Algorithm 2 such that n invocations of `invoke`(op_i) happened in α and all of them are completed in $C(\alpha)$.

Let r be the rank of the announcement node referenced by announces in $C(\alpha)$, and let us suppose that $r \geq 2$. Let us consider last time announces was updated in α, on Line 13, by a process p_i. Remark that whenever a process wins a compare-and-set on Line 18, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so an operation can only prevent p_i to win its compare-and-set once. Therefore, $n \leq k = f^{-1}(r-1+1) = f^{-1}(r)$. By definition of f^{-1}, we have $r \leq f(n)$.

One linearization node and at most $f(n)$ announce nodes, referencing at most $f(n)$ operation nodes, are reachable in $C(\alpha)$. Therefore, at most $O(f(n))$ shared memory locations dedicated to Algorithm 2 are reachable. ◀

5 Conclusion

This paper investigated the performance of concurrent data structure implementations (counters, queues, stacks, journals, etc.) in the infinite arrival model where the universal compare-and-set hardware instruction is available. It proves that the space complexity of a universal construction cannot be constant in the number of operations ever issued, although it can be super-constant.

This separation result may seem weak to separate only between constant and super-constant space, however, note that a low space complexity is obtained to the detriment of time complexity. This is captured by the function f. This function relates space complexity to the worst-case step/time complexity (f^{-1}); there is a kind of trade-off. This function can be seen as a continuum between wait-freedom and lock-freedom. While wait-freedom offers a finite time complexity and an ever-increasing space complexity, lock-freedom offers a constant quiescent space complexity and an infinite worst case time complexity (in a real setting and in the average, lock-free implementations are time efficient). The faster f grows, the closer we get to wait-freedom, and conversely, the slower the closer we get to lock-freedom. When parameterized with a slowly growing function, the proposed data structure can be as efficient as a lock-free data structure while benefiting from wait-freedom (the guarantee of a finite step complexity). An interesting open question, therefore, is whether other universal special hardware instructions can avoid this complexity issue.

References

1. Hagit Attiya and Sergio Rajsbaum. Indistinguishability. *Commun. ACM*, 63(5):90–99, 2020.
2. Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.
3. Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin. Wait-free universality of consensus in the infinite arrival model. In *33rd International Symposium on Distributed Computing, DISC, Hungary*, volume 146 of *LIPIcs*, pages 38:1–38:3, 2019.
4. James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *J. ACM*, 29(1):183–205, 1982.
5. Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 241–250, 2015.
6. David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 595–606. Springer, 2013.

7 Edsger Dijkstra. Over de sequentialiteit van procesbeschrijvingen (on the nature of sequential processes). *EW Dijkstra Archive (EWD-35), Center for American History, University of Texas at Austin (Translation by Martien van der Burgt and Heather Lawrence)*, 1962.

8 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

9 Faith E. Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In Soma Chaudhuri and Shay Kutten, editors, *Proc. of the 23rd Symposium on Principles of Distributed Computing, PODC 2004, Canada*, pages 80–87. ACM, 2004.

10 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

11 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

12 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

13 Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Parallel and Distributed Computing*, 4(4):163–183, 1987.

14 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proc. of International Symposium on Distributed Computing*, pages 164–178. Springer, 2000.

15 Matthieu Perrin, Achour Mostéfaoui, and Grégoire Bonin. Extending the wait-free hierarchy to multi-threaded systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 21–30, 2020.

16 Michel Raynal. Distributed universal constructions: a guided tour. *Bulletin of the EATCS*, 121, 2017.