Wait-free Algorithms: the Burden of the Past

Denis Bédin
University of Nantes
François Lépine
University of Nantes
Achour Mostéfaoui
University of Nantes
Damien Perez
University of Nantes
Matthieu Perrin
matthieu.perrin@univ-nantes.fr
University of Nantes

Research Article

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

Posted Date: March 22nd, 2024

DOI: https://doi.org/10.21203/rs.3.rs-4125819/v1

Additional Declarations: No competing interests reported.

Wait-free Algorithms: the Burden of the Past

Denis Bédin[1], François Lépine[1], Achour Mostéfaoui[1*], Damien Perez[1], Matthieu Perrin[1*]

[1]LS2N, Nantes Université, Nantes, France.

*Corresponding author(s). E-mail(s): achour.mostefaoui@etu.univ-nantes.fr; matthieu.perrin@univ-nantes.fr;
Contributing authors: denis.bedin@etu.univ-nantes.fr; francois.lepine@etu.univ-nantes.fr; damien.perez@univ-nantes.fr;

Abstract

Herlihy proved that compare-and-set (CAS) is universal in the classical computing system model composed of an a priori known number of processes. For this, he proposed the first universal construction capable of emulating any data structure with a sequential specification. 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.

This paper explores the complexity issues related to the wait-free CAS-based universal constructions. We first prove that CAS does not allow to implement wait-free and linearizable visible objects in the infinite arrival model with a space complexity bounded by the number of active processes. We then show that this lower bound is tight, in the sense that this dependency can be made as low as desired by proposing a wait-free and linearizable universal construction, using the CAS operation, whose space complexity dependancy on the number of ever issued operations is defined by a parameter that can be linked to any unbounded function.

This paper also proves that the lower bound obtained for CAS-based algorithms might be avoided by the use of other synchronization primitives. As an example, we explore algorithms based on the memory-to-memory swap special instruction, that exchanges the content of two shared registers. We propose a universal construction based on memory-to-memory swap whose complexity only depends on the contention, and we illustrate how compare-and-set and memory-to-memory swap can be used jointly within a wait-free queue algorithm.

Keywords: Compare-And-Set, Concurrent Object, Infinite arrival model, Linearizability, Memory complexity, Memory-To-Memory Swap, Multi-Threaded Systems, Shared-Memory, Universality, Wait-freedom.

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 [1]. 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 [2]. However, these implementations have a space complexity linear with the number of processes [3]. 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 [4] and lock-freedom [5][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 [6–8]. The design of the most popular concurrent data structures such as counters, queues, stacks, logs, etc. has been very active these last 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 [9]. The formulation of the Consensus problem is particularly simple. Each process proposes a value and all non-faulty processes eventually decide on the same value among those which are proposed. In contrast, consensus was proved universal in [4]. 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

[1] In [5] lock-freedom is called non-blocking.

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 an object from its sequential specification. Since then, several universal constructions have been proposed for different special hardware instructions such as CAS and LL/SC [10]. 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 [11]. 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, and TAS for example. It has been proved in [12] 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 [13] 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/threads 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 [14] 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 wait-free synchronization based on hardware special instructions. 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?*

Responding to this question, this paper has four main contributions.

Contribution 1: a lower bound.

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: a new CAS-based universal construction.

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.

Contribution 3: on the memory-to-memory swap side.

In a second stage, we explored another special instruction: the memory-to-memory swap. This hardware instruction swaps the contents of two memory locations whereas the classic swap instruction swaps a memory location with a processor register. It was first proposed in [4]. So far, it has not been studied since compare&swap and other special instructions access a unique shared location. We exhibit a universal construction based on memory-to-memory swap that enjoys a space complexity linear in the number of active threads, i.e. constant in the number of terminated operations.

On the other side, this proves that beyond computability, special instructions provide different performance capabilities, and that the lower bound identified for CAS-based algorithms is not a fatality, as other hardware special instructions may be more appropriate for wait-free algorithms.

Contribution 4: the queue use case.

Finally, we illustrate the joint use of memory-to-memory swap and compare-and-set in the implementation of a simple, wait-free linearizable queue, with a complexity comparable to Michael&Scott lock-free queue [15]. This allows us to discuss the usage differences between compare-and-set and memory-to-memory swap, and how, although both special instructions are universal, they tackle different and complementary aspects of synchronisation.

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. Section 5 presents a simple universal construction based on memory-to-memory swap that has a constant space complexity. An implementation of a concurrent queue is proposed in Section 6 and finally, Section 7 concludes the paper. A Java implementation of the algorithms presented in the paper is available at https://github.com/MatthieuPerrin/BurdenOfThePast.

2 Model

This paper considers the infinite arrival model [13] 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 a more complex data structure) and returns its reference, i.e. it allocates the memory locations needed to manage the object and initializes them by calling a constructor defined within the algorithm of the type T, alongside the operations on T. 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 special atomic instruction that are referred to as enriching special instructions. Reads, writes and enriching special instructions are atomic in the sense that the different executions of the calls to these operations are totally ordered. In this paper, we consider two enriching special instructions: compare-and-set and memory-to-memory swap.

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

[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.

they are equal, then *update* is written in the register and **true** is returned. Otherwise, the state is left unchanged and **false** is returned.

The memory-to-memory-swap instruction. concerns two registers x and y and can be invoked using the expression $\texttt{swap}(x, y)$, which does not return any value. In the execution, the value stored in x before the invocation is written into y, and the value stored in y before the invocation is written into x.

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 an enriching special instruction. 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.

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 that is executed when an object of this type is instanciated (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 `initState`, and whose transitions are defined by two functions: `newState` and `returned` that take as arguments a state of the object and an operation, and return respectively 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 and returns the first element from the queue is specified as $\texttt{newState}([x_0, x_1, \ldots, x_n], \texttt{dequeue}) = [x_1, \ldots, x_n]$ and $\texttt{returned}([x_0, x_1, \ldots, x_n], \texttt{dequeue}) = x_0$.

3 Lower Bound on Universal Constructions using Compare-And-Set

This section explores the limitations of wait-free linearizable universal constructions based on compare-and-set. More precisely, Theorem 6 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 invisible process and visible object. We call *invisible 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 [12], 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)". In other words, a visible object is, by definition, an object that has an operation that cannot be performed by an invisible process. Intuitively, any update operation on any visible object 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 invisible 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. For the sake of preciseness, Lemma 1 and Theorem 6 only consider a linearizable counter as an example of a visible object.

As explained in Section 2.2, the state of a distributed computation can be captured by the notion of configuration. Many impossibility proofs are based on the fact that, at some point of a distributed execution, some process has no way to distinguish two different configurations. As said in [16]: "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".

The notion of *invisible process* introduced below will help to build indistinguishable configurations.

Definition 4 (Invisibility)**.** *Let α be a finite execution, and let p be a process. We say that p is* invisible *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 to p'.*

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

Lemma 1 (Counters are visible)**.** *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 an invisible execution.*

Proof. Let A be a wait-free linearizable implementation of a counter. Suppose (by contradiction) that A has an invisible 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 5 calls them static), and other memory locations may be allocated by some operation, and later be made unreachable by the same or another operation (Definition 5 calls them dynamic). Lemma 3 builds an execution in which some process p remains invisible forever because, whenever p covers a static location x (i.e. p is about to write in x, see Definition 6), 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.

Definition 5 (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 invisible in α takes steps in β, and 3) all processes are either invisible 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 6 (Covering)**.** *Let α be a finite execution, p a process and x a shared register. 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 3, algorithms that follow a normal form, defined in Definition 7. This assumption is done without loss of generality, since the proof of Lemma 2 discusses how to normalize any algorithm.

Definition 7 (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 2 (Normalization)**.** *Any wait-free algorithm A 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 algorithm A' in normal form with a constant quiescent complexity and verifying the same safety properties as A.*

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 Algorithm A' verifies all liveness and safety properties of A. In particular, A' is

Algorithm 1: Normalisation of shared registers

```
   constructor (initial) is
1  |  u_0 ← {value ← initial, time ← 0, pid ← 0};
2  |  internal ← new Reg(u_0);
   operation read() is
3  |  return internal.read().value;
   operation write(v_i) is
4  |  cl_i ← cl_i + 1;
5  |  u_i ← {value ← v_i, time ← cl_i, pid ← i};
6  |  internal.write(u_i);
   operation CAS(e_i, u_i) is
7  |  e'_i ← internal.read();
8  |  if e'_i.value ≠ e_i then return false;
9  |  cl_i ← cl_i + 1;
10 |  u'_i ← {value ← u_i, time ← cl_i, pid ← i};
11 |  return internal.CAS(e'_i, u'_i);
```

also wait-free. 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 3 (Quiescent complexity and invisibility)**.** *All wait-free algorithms in normal form with constant quiescent complexity have an invisible execution.*

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 $i \in \{0, \ldots, k\}$.

Claim 4 ($P_1(i)$)**.** *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 invisible processes in $C(\alpha\beta)$ that didn't participate in α, and all non-invisible 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 $j \in \{0, \ldots, i+1\}$.

Claim 5 ($P_2(i, j)$)**.** *For all finite executions α, there exists an extension $\alpha\beta$ of α 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 invisible processes that did not participate in α, or (2) at least j static locations are write-covered in $C(\alpha\beta)$ by invisible processes that did not participate in α, and all non-invisible 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 invisible 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 invisible, which implies $P_2(i, j)$. 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 invisible 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 invisible 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 invisible 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 invisible 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 invisible 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 invisible 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 invisible 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 5, all k are static.

Therefore, all k locations are covered at least twice by invisible processes in $C(\alpha\beta)$. In particular, there are two invisible 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 invisible in $\alpha\beta\delta$ and q terminates its execution, so $\alpha\beta\delta$ is an invisible execution of A. □

Theorem 6 (Lower bound for CAS-based algorithms)**.** *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 2, we can suppose without loss of generality that it is in normal form. By Lemma 3, A has an invisible execution, and by Lemma 1, A does not have an invisible execution. This is a contradiction, so A does not exist. □

4 Upper Bound on Universal Constructions using Compare-And-Set

From Theorem 6, we can derive that the quiescent complexity of any wait-free linearizable universal construction is in $\omega(1)$. Differently, [17] presents such a construction with a quiescent complexity in $\mathcal{O}(n)$. The present section closes the gap thanks to Algorithm 2, [3] 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.

[3] a Java implementation is available at https://github.com/MatthieuPerrin/BurdenOfThePast/blob/main/fr/univ_nantes/burdenOfthePast/universalImplementations/CasUniversal.java

Algorithm 2: Universal construction using compare-and-set

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

- 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.op is an operation of the state machine, o.res is a register storing either $\perp$ or a value that can be returned by o.op, and o.done is a Boolean register. An operation node o is created when an operation o.op is invoked by a process p_i on the state machine, initially with o.res $= \perp$ and o.done $=$ **false** (Line 6). After the operation has been linearized, o.done is set to **true** by some process p_j (possibly different to p_i) (Line 21), which serves as a signal to p_i that it can return o.res (Lines 13-13).
- 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

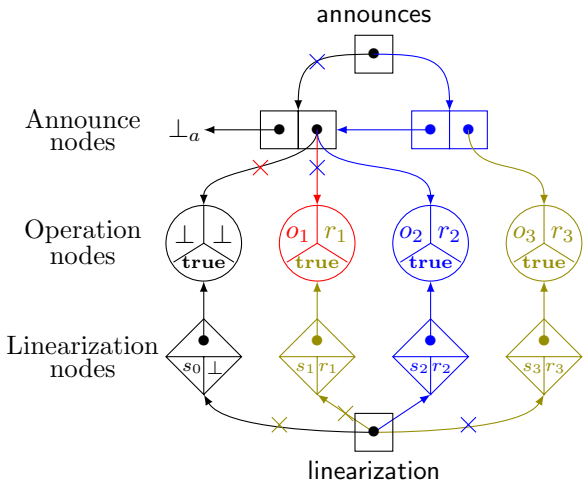

Fig. 1 An execution of Algorithm 2, with $f(1) = 1$. The initial state is represented in black. Processes p_1 (in red), p_2 (in blue) and p_3 (in green) attempt to concurrently execute o_1, o_2 and o_3, respectively. Initially, p_1 and p_2 read the same announce node a, and p_1 wins the first compare-and-set, 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.

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 $\texttt{rank}(a)$ of a node a as the length of the linked list, i.e. $\texttt{rank}(\perp_a) = 0$ and $\texttt{rank}(a) = \texttt{rank}(a.\mathsf{next}) + 1$ 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.res 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, containing a dummy operation node. 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 the same dummy operation node as the initial announce 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.

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 6), 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 14), after helping operation nodes already announced to be linearized by calling $\texttt{help}(a_i)$ (Line 11). If p_i fails to write o_i into $a_i.\mathsf{o}$ $f^{-1}(\texttt{rank}(a_i)+1)$ times, it tries to insert a new announce node at the head of the announces list (Lines 9 and 10) to prevent newly arrived processes to compete on $a_i.\mathsf{o}$, 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 $a_i.\mathsf{o}.\mathsf{op}$ of the operation node $a_i.\mathsf{o}$, it first helps recursively all announce nodes reachable from a_i (Line 16), and then tries to replace the linearization node in linearization using compare-and-set, until success (Lines 19 to 26). Remark that the new state of the state machine, as well as the value returned by an operation, are computed (Lines 23-24), before the linearization node referencing the operation is created, and the return value is later reported on the operation node (Line 20), possibly by still a different process.

Lemma 7 (Termination of help)**.** *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 17. By Line 22, $o_i.\mathsf{done}$ is always **false**, so no process ever wins the compare-and-set on Line 26 with a linearization node referencing o_i (otherwise, the next process that writes linearization on Line 26 would previously have set $o_i.\mathsf{done}$ to **true** on Line 21), and $a_i.\mathsf{o} = o_i$ 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 12 and 2) whenever a process wins a compare-and-set on Line 14, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 12) and then terminates on Line 13, so p_j can only be prevented to terminate K times. After that, $a_j.\mathsf{o}$ is never written, and $a_j.\mathsf{o}.\mathsf{done}$ remains **true** forever. In particular, no further invocation of $\texttt{help}(a_j)$ executes Line 26, as they terminate on Line 22.

After that point, all new invocations of $\texttt{help}(a_j)$ by some process p_j on Line 12 are such that $\texttt{rank}(a_j) \geq r_i$. Thanks to Line 16, and by what precedes, p_j's first execution of Line 26 is during its recursive call $\texttt{help}(a_i)$, in which p_j reads $a_i.\mathsf{o} = o_i$ on Line 17. As all executions of Line 26 try to write some l_j with $l_j.\mathsf{o} = o_i$, only one of them succeeds. After that point, some process (possibly p_i) reads l_j on Line 19 and writes **true** in $o_i.\mathsf{done}$ on Line 21, which is a contradiction. □

Lemma 8 (Wait-freedom)**.** *Algorithm 2 is wait-free.*

Proof. Let us consider an invocation of $\texttt{invoke}(op_i)$ by a process p_i. By Lemma 7, 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 10 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 7, so only a finite number of processes compete with p_i on Line 14. 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 12) and then terminates on Line 13, so an operation can only prevent p_i to win its compare-and-set once. Therefore, p_i eventually terminates its execution. □

Lemma 9 (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.op $= op_i$ is such that an invocation of linearization.CAS(l', l_i) returns **true** on Line 26. Indeed, first remark that all linearization nodes written in linearization are unique, because they are created immediately before they are written, and that only one operation node o_i, built on Line 6 by p_i, is such that o_i.op $= op_i$. Suppose (by contradiction) that two linearization nodes l_j and l_k, with l_j.o $= l_k$.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_j i.e. such that the invocation linearization.CAS(l_j, l_m) by some process p_m returned **true**. Process p_k read l in linearization on Line 19 before **false** in o_i.done on Line 22, before p_m wrote **true** in o_i.done on Line 21, before p_m invoked linearization.CAS(l_j, l_m) on Line 26. Therefore, l is at least as old as l_k. It is impossible that $l = l_k$ because it would mean $p_k = p_m$ would have executed Line 22 before Line 21, and it is impossible that l is older than l_k because it would have been overwritten by l_k 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.op $= op_i$.

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 13, so some process p_j wrote **true** in l_i.o.done $= o_i$.done on Line 22, after having read l_i on Line 19, which can only happen after some process p_k wrote l_i on Line 26. 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 6.

Finally, let us remark that, thanks to Lines 23-24, 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 26. If Process p_i returns r_i at the end of the execution of `invoke`(op_i), it read it in o_i.res.`read`() on Line 13, after reading **true** in o_i.done on Line 13, which can only happen if some process wrote **true** in o_i.done on Line 21 after writing l_i.res in o_i.res on Line 20, with l_i.o $= o_i$ and l_i read in linearization on Line 19. Therefore, p_i returns the same value as in E.

In conclusion, α is linearizable. □

Lemma 10 (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 the last time announces was updated in α, on Line 10, by a process p_i. Remark that whenever a process wins a compare-and-set on Line 14, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 12) and then terminates on Line 13, 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 Universal Construction using Memory-to-memory Swap

Sections 3 and 4 depict a fairly negative picture for wait-free CAS-based data structures in the infinite arrival model: any CAS-based wait-free linearizable universal construction must maintain, and therefore traverse, a data structure whose size can only grow as more and more operations are performed on the object. A natural question that arises from these results is whether this complexity issue is inherent to wait-free synchronization, or rather a weakness of CAS-based algorithms. In other words, do there exist other enriching special instructions that do not present the same complexity issue? This section answers the question positively by presenting Algorithm 3, a wait-free linearizable universal constructions with a constant quiescent complexity, based on the memory-to-memory swap special instruction, that simply exchanges the content of two registers passed in arguments. [4]

In Algorithm 3, processes share a register last that references nodes of type `Node`. A node n is composed of four fields: n.op is an operation of the state machine, n.state is a register storing either $\perp$ or a reference to a state of the state machine, n.res is a register storing either $\perp$ or a value that can be returned by n.op, and n.prev is a register that may reference another node of the same type. The last register is initialized with a node n_0, such that n_0.state is initialized to `initState`, the initial state of the state machine.

When a process p_i needs to apply an operation op_i on the state machine, it invokes `invoke`(op_i) on the universal construction. Process p_i first creates an node n_i containing its operation, such that n_i.prev references n_i (Line 4 and 5), and then swaps the values of last and n_i.prev (Line 6). Immediately after the swap, last references n_i and n_i.prev references the node that was previously referenced by last. In other words, Line 6 alone creates a total order on all the operations, encoded by a linked list. Then, p_i recursively traverses the list, computing the resulting state and return value of each node it encounters, and then detaching the end of the list by writing $\perp$ in the prev fields.

[4] a Java implementation is available at https://github.com/MatthieuPerrin/BurdenOfThePast/blob/main/fr/univ_nantes/burdenOfthePast/universalImplementations/SwapUniversal.java

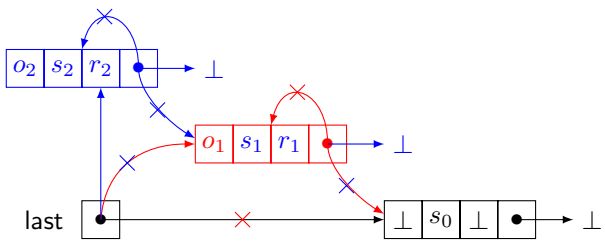

Fig. 2 Processes p_1 (in red) and p_2 (in blue) attempt to concurrently execute o_1 and o_2, respectively. Here, p_1's swap happens before p_2's. A linked list is formed, and o_1 will be linearized before o_2 by p_2, that also detaches the nodes of the list.

Lemma 11 (Wait-freedom)**.** *Algorithm 3 is wait-free.*

Proof. Let us suppose (by contradiction), that some call to `help`(n_i) by a process p_i takes an infinite number of steps. Without loss of generality, we can suppose that, amongst all the nodes n_j for which a call to `help`(n_j) does not terminate, the process that created n_i was the first to execute Line 6. In particular, p_i read n'_i on Line 9, that was written in last on Line 6 before n_i, so if there was a recursive call to `help`(n'_i) on Line 11, it terminated. Therefore, `help`(n_i) terminates as well.

The algorithm for `invoke`(op_i) contains no loop and `help`(n'_i) terminates, so Algorithm 3 is wait-free. □

Lemma 12 (Linearizability)**.** *Algorithm 3 is linearizable.*

Proof. Let us consider an execution α admitted by Algorithm 3.

Let us define the linearization point of any operation `invoke`(op_i) as, if it exists, the execution of `swap` on Line 6. Clearly, any terminating operation has a unique linearization point between its invocation and termination point.

Let us consider a process p_i executing `help`(n_i). If p_i enters in the conditional (Line 10), as $n'_i \neq \perp$, its value for n'_i was written on Line 6, i.e. n'_i.o immediately precedes n_i.o in the linearization order, or n'_i is the node created at initialization if n_i.o is the first operation. By Lines 13-14, the values written in n_i.state and n_i.res are the states and return values reached in the sequential execution E in which the same operations are executed in the order of their linearization point.

As n_i.res is always read (Line 8) when n_i.prev $= \perp$ (Line 10 or 15) and $\perp$ is only written in n_i.prev (Line 15) after n_i.res is updated (Line 14), the value returned by p_i is the same as the return value of n_i.o in E. In conclusion, α is linearizable. □

A link with help-free computing.

In [11], Censor-Hillel, Petrank and Timnat introduced the notion of (valency-based) helping, and proved that many wait-free shared objects, and therefore wait-free universal constructions, required helping to be implemented in M[CAS]. Algorithm 2 in the previous section, illustrates how this requirement for helping leads to a non-constant quiescent complexity. In contrast, [11] proposed the *fetch-and-cons object*, a list of items providing an operation that atomically adds an item at the beginning of the list and returning the items following it. They then proved that fetch-and-cons objects

Algorithm 3: Universal construction using memory-to-memory-swap

```
   constructor (initState) is
1      state_0 ← new Reg(initState);
2      n ← new Node { op ← ⊥, state ← state_0, res ← new Reg(⊥), prev ← new Reg(⊥) };
3      last ← new Reg(n);
   operation invoke(op_i) is
4      n_i ← new Node { op ← op_i, state ← new Reg(⊥), res ← new Reg(⊥), prev ← new Reg(⊥) };
5      n_i.prev.write(n_i);
6      swap(last, n_i.prev);
7      help(n_i);
8      return n_i.res.read();
   function help(n_i) is
9      n'_i ← n_i.prev.read();
10     if n'_i ≠ ⊥ then
11         help(n'_i);
12         s_i ← n'_i.state.read();
13         n_i.state.write(newState(l_i.state, o_i.op));
14         n_i.res.write(returned(l_i.state, o_i.op));
15         n_i.prev.write(⊥);
```

could be used in help-free universal constructions. Unfortunately, any practical implementation of fetch-and-cons objects must use at least as many registers as the size of the list, resulting in a quiescent complexity in $\mathcal{O}(n)$.

In Algorithm 3, although the work done by p_i on Lines 12 to 15 does benefit to other processes, this is not valency based helping because the linearization order is fixed on Line 6. Indeed, Algorithm 3 is a help-free algorithm. Hence, memory-to-memory swap is, to our knowledge, the first special instruction that operates on a fixed and finite number of registers, and that allows help-free universal constructions.

6 A wait-free linearizable queue

Universal constructions, like the ones presented in Sections 4 and 5, are usually less efficient than dedicated constructions, although they can serve to illustrate synchronization techniques at the heart of more evolved dedicated constructions, or simply adapted to the specificities of operations on particular shared objects. For example, a simple stack can be derived from the simplest lock-free universal construction using

compare-and-set [18]. Similarly, a wait-free stack can be easily derived from Algorithm 3. A drawback of this approach is that it tends to force an ordering of all operations, unnecessarily increasing contention. For example, enqueue and dequeue operations on non-empty queues commute, so they do not need to be ordered. Efficient lock-free queues have been proposed on this principle, called *disjoint-access parallelism*, including the celebrated Michael-Scott queue [15]. In this section, we show how to implement an efficient wait-free linearizable queue when both `swap` and `CAS` are available.

Algorithm 4 [5] maintains two lists, each in a similar fashion as Algorithm 3: one containing enqueued values, and one for managing dequeue operations. The head of each list is accessible to all processes *via* a shared register. The first register, enqueues references an enqueue node of type `ENode`. An enqueue node e is made of three fields: e.prev and e.next are registers that reference other enqueue nodes, and e.value is a value that can be stored in the queue. The second register, dequeues references a dequeue node of type `DNode`. A dequeue node d is composed of three fields: d.prev is a register referencing another dequeue node, d.match is a register referencing an enqueue node, and d.value is a register storing a value that can be enqueued. Both enqueues and dequeues are initialized with dummy nodes e_0 and d_0, such that d_0.match $= e_0$. At the end of quiescent executions, the dequeues list contains exactly one dequeue node d, and the enqueues list contains one enqueue node per value stored in the queue, and one more additional node referenced by d.match at the tail of the list.

When Process p_i enqueues a value v, it first creates an enqueue node e_i such that e_i.prev $= e_i$, and then swaps enqueues and e_i.prev, which inserts e_i in the list accessible from enqueues, i.e. least recent enqueue nodes are accessible from most recent enqueue nodes. In order to make the values in the list accessible to dequeues operations in a first-in-first-out order, links are reversed by `helpEnqueue`(e_i) in a similar way as in Algorithm 3.

Similarly, when Process p_i invokes `dequeue`(), it inserts a dequeue node d_i in the list accessible from dequeues, and helps the previous dequeue operations to obtain their values. At this point, d_i.prev refers to the dequeue operation linearized immediately before d_i, and d_i.prev.match references the last enqueue node whose value was dequeued. If the queue is not empty, d_i.prev must be set to d_i.prev.match.next, and d_i.value must be set to d_i.prev.match.next.value. Otherwise, d_i.prev must be set to d_i.prev.match, and d_i.value must be left unchanged to $\perp$. When `dequeue`() is called on an empty queue, concurrently with an enqueue operation that creates an enqueue node e, some process p_i may read d_i.prev.match.next $= \perp$ and some other process may read d_i.prev.match.next $= e$. To remove the ambiguity, p_i and p_j compete on a binary consensus over d_i.match

In Algorithm 4, this consensus is done using compare-and-set (Lines 25 and 26), but it could be easily replaced by load-link and store-conditional, or even by swap as a simple consensus algorithm can be derived from the universal construction. Remark, however, that no known consensus algorithm from memory-to-memory swap is as efficient as what we can obtain from compare-and-set. This observation illustrates

[5] a Java implementation is available at https://github.com/MatthieuPerrin/BurdenOfThePast/blob/main/fr/univ_nantes/burdenOfthePast/examples/SwapQueue.java

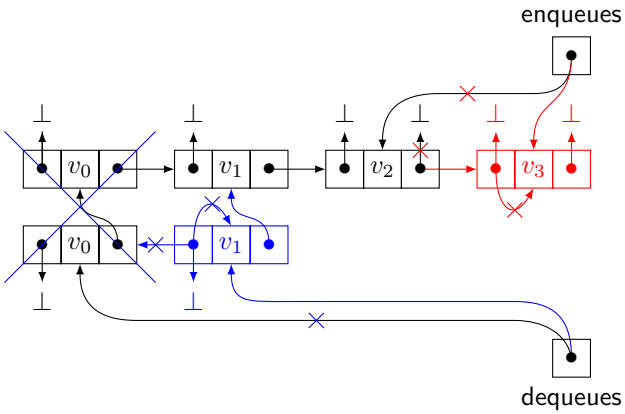

Fig. 3 The data structure in black encodes the queue $[v_1, v2]$. An enqueue node and a dequeue node reference the last dequeued value, v_0, and serve as an entry point for the tail of the queue. Process p_1 (in red) enqueues v_3, and inserts an enqueue node at the head of the list. Process p_2 (in blue) concurrently dequeues from the queue. It first inserts a dequeue node at the tail of the list, then links it to v_1, the oldest value still in the queue, and finally detaches the nodes containing v_0 that are no longer necessary and can be garbage-collected.

that, although compare-and-set and memory-to-memory swap are computationally equivalent, both have their own purpose when designing concurrent data structures: compare-and-set seems to be more easily used to solve safety issues, and memory-to-memory swap helps to solve liveness issues.

Lemma 13 (Wait-freedom)**.** *Algorithm 4 is wait-free.*

Proof. Let us suppose (by contradiction), that some call to `helpEnqueue`(e_i) by a process p_i takes an infinite number of steps. Without loss of generality, we can suppose that, amongst all the enqueue nodes e_j for which a call to `helpEnqueue`(e_j) does not terminate, the process that created e_i was the first to execute Line 7. In particular, p_i read e'_i on Line 9, that was either 1) written in `enqueues` on Line 7 before e_i, in which case any potential recursive call to `helpEnqueue`(e'_i) on Line 11 terminated, or 2) created at initialization, in which case e'_i.prev $= \perp$. In both cases, `helpEnqueue`(e_i) terminates as well.

Similarly, the prev fields of the dequeue nodes form a linked list that ultimately points to $\perp$, so recursive calls to `helpDequeue` terminate.

The algorithms for `enqueue`(v) and `dequeue`$()$ contain no loop and `helpEnqueue` and `helpDequeue` terminate, so Algorithm 4 is wait-free. □

Lemma 14 (Match agreement)**.** *If two processes p_i and p_j execute Line 27 of* `helpDequeue`(d)*, then $e_i = e_j \neq \perp$ and $e'_i = e'_j \neq \perp$.*

Proof. Let us prove the property by induction on the sequence $d_0, d_1, ...$ of the dequeue nodes, such that d_0 is the node created at initialization, and d_k is the k^{th} dequeue node for which Line 16 was executed. For the first node, d_0.prev $= \perp$, so no process executes Line 27 of `helpDequeue`(d_0). Suppose the property is true for some dequeue node d_k, and suppose Two processes p_i and p_j execute Line 27 of `helpDequeue`(d_k).

Algorithm 4: Wait-free linearizable queue

```
   constructor is
   |   e_0 ← new ENode { prev ← new Reg(⊥), value ← ⊥, next ← new Reg(⊥) };
1  |   d_0 ← new DNode { prev ← new Reg(⊥), value ← new Reg(⊥), match ← new Reg(e_0) };
2  |   enqueues ← new Reg (e_0);
3  |   dequeues ← new Reg (d_0);
4  |
   operation enqueue(v) is
   |   e_i ← new ENode { prev ← new Reg(⊥), value ← v, next ← new Reg(⊥) };
5  |   e_i.prev.write(e_i);
6  |   swap(enqueues, e_i.prev);
7  |   helpEnqueue(e_i);
8  |
   function helpEnqueue(e_i) is
   |   e'_i ← e_i.prev.read();
9  |   if e'_i ≠ ⊥ then
10 |   |   helpEnqueue(e'_i);
11 |   |   e'_i.next.write(e_i);
12 |   |   e_i.prev.write(⊥);
13 |   |
   operation dequeue() is
   |   d_i ← new DNode { prev ← new Reg(⊥), value ← new Reg(⊥), match ← new Reg(⊥) };
14 |   d_i.prev.write(d_i);
15 |   swap(dequeues, d_i.prev);
16 |   helpDequeue(d_i);
17 |   return d_i.value.read();
18 |
   function helpDequeue(d_i) is
   |   d'_i ← d_i.prev.read();
19 |   if d'_i ≠ ⊥ then
20 |   |   helpDequeue(d'_i);
21 |   |   e'_i ← d'_i.match.read();
22 |   |   e_i ← e'_i.next.read();
23 |   |   if e_i = ⊥ then e_i ← e'_i;
24 |   |   d_i.match.CAS(⊥, e_i);
25 |   |   e_i ← d_i.match.read();
26 |   |   if e_i ≠ e'_i then d_i.value.write(e_i.value);
27 |   |   d_i.prev.write(⊥);
28 |   |
```

As the condition of Line 20 was true, they read a non-$\perp$ value on Line 20, that can only have been written on Line 16, i.e. $d'_i = d'_j = d_{k-1}$, and $e'_i = e'_j \neq \perp$ by induction.

By Line 24, no process writes $\perp$ on Line 25, so e_i and e_j are the same non-$\perp$ value written by the first process that executed Line 25 `helpDequeue`(d), i.e. $e_i = e_j \neq \perp$. □

Lemma 15 (Linearizability)**.** *All executions admitted by Algorithm 4 are linearizable.*

Proof. Let α be an execution admissible by Algorithm 4.

Let us define the linearization point of an operation `enqueue`(v) in α, in which an enqueue node e is created on Line 5, as the first write on Line 12 by a process executing `helpEnqueue`(e), if it exists, and the linearization point of an operation `dequeue`() in α, in which a dequeue node d is process that executed the compare-and-set on Line 25, while executing `helpDequeue`(d), if it exists. Clearly, the linearization point of an operation, if it exists, is unique and happens after the node was created, so after the starting point of the operation.

Let us consider a terminated `enqueue`(v) (resp. `dequeue`()) operation by p_i, in which a node n is created. Process p_i invokes `helpEnqueue`(n) on Line 8 (resp. `helpDequeue`(n) on Line 17) and either executes Line 12 (resp. Line 23 and 25) or reads $\perp$ in n.prev on Line 9 (resp. Line 19). As n.prev $\neq \perp$ after Line 7 (resp. Line 16), some process previously executed Line 13 (resp. Line 28) and Line 12 (resp. Line 23) earlier. In the case of a dequeue operation, that process either won a compare-and-set on Line 25, or n.match was updated earlier by a process that won a compare-and-set on Line 25 after executing Line 23. In all cases, the operation has a linearization point, before its termination point.

Let us define, for all finite prefixes β of α, the active nodes of $C(\beta)$ as the set of enqueue nodes e_i such that some process wrote e_i on Line 12 in β, and if some process p_i wrote e_i on Line 25 in α, then its read on Line 23 already occurred in β. Remark that the active nodes form a linked list that can be followed using their next fields. Indeed, whenever a process p_i writes e_i on Line 12 in some register e'_i.next, e'_i was previously written in e_i.prev (Line 9), which could only happen if e'_i is the node written in enqueues immediately before e_i, either on Line 7 or at initialization. Moreover, a process can only read e_i on Line 23 if its predecessor e'_i was written in d'_i.match by some process on Line 27, which can only happen if e'_i was at some point read on Line 23 or created at initialization, i.e. if e'_i is not an active node. We also define the virtual state of $C(\beta)$ as the sequence of values stored in e_i.value, in their order of appearance in the list.

Let us now consider a finite prefix $\beta\gamma$ of α, such that γ is only composed of one step. Let S be the virtual state of $C(\beta)$ and S' be the virtual state of $C(\beta\gamma)$. Remark that, if γ is not a linearization point, then the virtual state is left unchanged as the next field of enqueue nodes are only edited on Line 12, and only the first write actually changes the state, which occurs during a linearization point.

Let us suppose that γ is the linearization point of an operation `enqueue`(v). By definition, it is the first write on Line 12 by a process executing `helpEnqueue`(e), with e.value $= v$, so e is an active node in $C(\beta\gamma)$ but not in $C(\beta)$, and $S' = Sv$.

Let us suppose that γ is the linearization point of an operation `dequeue`() that returns $\perp$, performed by a process p_i that created a dequeue node d. Let p_j be the first process to call compare-and-set on d.match on Line 25. As d.value $= \perp$ when p_i returns, the condition of Line 27 was wrong when p_i executed `helpDequeue`(d), i.e. $e_i = e'_i$, and by Lemma 14, $e_j = e_i = e'_i = e'_j$. Moreover, by definition of p_j, e_j is the value p_j wrote on Line 25. As $e_j = e'_j$, the condition of Line 24 is true, so p_j read $\perp$ in γ, and $S = S'$. Moreover, in β, enqueue nodes that precede e_i have already been read, and, as e'_i.next is not yet written in β, the others have not yet been fully inserted, so $S = S'$ is the empty sequence.

Let us suppose that γ is the linearization point of an operation `dequeue`() that returns v, performed by a process p_i that created a dequeue node d. Let p_j be the first process to call compare-and-set on d.match. As d.value $= v$ when p_i returns, some process reads an enqueue node $e_i \neq \perp$ on Line 26, with e_i.value $= v$, so p_j executed d.match.`CAS`$(\perp, e_i)$ (by Lemma 14), on Line 25. Therefore, p_j read e_i on Line 23, which made it an inactive node. Therefore, $S = vS'$.

In conclusion, the sequence of linearization points define a sequential execution E such that the state of the queue in E is always the same as the virtual state in α, and all returned values are the same in α and in E. In other words α is linearizable. □

7 Conclusion

This paper investigated the performance of concurrent data structure implementations (counters, queues, stacks, journals, etc.) in the infinite arrival model where enriching special instructions are available.

When only the universal compare-and-set hardware instruction is available, we proved 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).

We also proved that this complexity issue is not inherent to wait-free synchronisation as we identified a special hardware instruction, memory-to-memory swap, and a universal construction based on memory-to-memory swap with a constant space complexity. Finally, we discussed the relevance of having both compare-and-set and memory-to-memory swap. For that, we proposed the first wait-free implementation of

a queue, to our knowledge, that has a space and time complexities comparable to the celebrated lock-free implementation by Michael and Scott. This example illustrates what advantage can be attributed to each of the two instructions, and how we can benefit from a joint use of both.

A drawback of memory-to-memory swap is that it is not currently implemented on available hardware architectures. One reason for that is that it operates on two memory locations, which makes it harder to implement. An interesting open question, therefore, is whether this complexity issue can be avoided using a combination of existing special hardware instructions, or by new universal special hardware instructions that operate on a single memory location.

Declarations

Funding.

This work was partially supported by the French ANR project ByBloS (ANR-20-CE25-0002-01).

Competing interests.

The authors declare they have no conflicts of interest or competing interests related to this work.

Data availability.

As a theoretic work, all the data required to reproduce and assess the validity of this work has been included in this paper.

References

[1] Dijkstra, E.: 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)

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

[3] Burns, J.E., Jackson, P., Lynch, N.A., Fischer, M.J., Peterson, G.L.: Data requirements for implementation of n-process mutual exclusion using a single shared variable. J. ACM **29**(1), 183–205 (1982)

[4] Herlihy, M.: Wait-free synchronization. ACM TOPLAS **13**(1), 124–149 (1991)

[5] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)

[6] Attiya, H., Welch, J.L.: Distributed Computing - Fundamentals, Simulations, and Advanced Topics (2. Ed.). Wiley series on parallel and distributed computing, Wiley (2004)

[7] Dice, D., Hendler, D., Mirsky, I.: 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. Lecture Notes in Computer Science, vol. 8097, pp. 595–606. Springer (2013)

[8] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Morgan Kaufmann (2008)

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

[10] Raynal, M.: Distributed universal constructions: a guided tour. Bulletin of the EATCS **121** (2017)

[11] Censor-Hillel, K., Petrank, E., Timnat, S.: Help! In: Proc. of the ACM Symposium on Principles of Distributed Computing, pp. 241–250 (2015)

[12] Fich, F.E., Hendler, D., Shavit, N.: On the inherent weakness of conditional synchronization primitives. In: Chaudhuri, S., Kutten, S. (eds.) Proc. of the 23rd Symposium on Principles of Distributed Computing, PODC 2004, Canada, ACM, pp. 80–87 (2004)

[13] Merritt, M., Taubenfeld, G.: Computing with infinitely many processes. In: Proc. of International Symposium on Distributed Computing, pp. 164–178 (2000). Springer

[14] Perrin, M., Mostéfaoui, A., Bonin, G.: Extending the wait-free hierarchy to multi-threaded systems. In: Proceedings of the 39th Symposium on Principles of Distributed Computing, pp. 21–30 (2020)

[15] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275 (1996)

[16] Attiya, H., Rajsbaum, S.: Indistinguishability. Commun. ACM **63**(5), 90–99 (2020)

[17] Bonin, G., Mostéfaoui, A., Perrin, M.: Wait-free universality of consensus in the infinite arrival model. In: 33rd International Symposium on Distributed Computing, DISC, Hungary. LIPIcs, vol. 146, pp. 38–1383 (2019)

[18] Treiber, R.K.: Systems Programming: Coping with Parallelism, International

Business Machines Incorporated, Thomas J. Watson Research (1986)