Noname manuscript No.
(will be inserted by the editor)

Long-Lived Counters with Polylogarithmic Amortized Step Complexity

Mirza Ahad Baig · Danny Hendler · Alessia Milani · Corentin Travers

Received: date / Accepted: date

Abstract A shared-memory counter is a widely-used and well-studied concurrent object. It supports two operations: An `Inc` operation that increases its value by 1 and a `Read` operation that returns its current value. In [JTT00], Jayanti, Tan and Toueg proved a linear lower bound on the *worst-case* step complexity of obstruction-free implementations, from read-write registers, of a large class of shared objects that includes counters. The lower bound leaves open the question of finding counter implementations with sub-linear *amortized* step complexity.

In this work, we address this gap. We show that n-process, wait-free and linearizable counters can be implemented from read-write registers with $O(\log^2 n)$ amortized step complexity. This is the first counter algorithm from read-write registers that provides sub-linear amortized step complexity in *executions of arbitrary length*. Since a logarithmic lower bound on the amortized step complexity of obstruction-free counter implementations exists, our upper bound is within a logarithmic factor of the optimal. The worst-case step complexity of the construction remains linear, which is optimal.

This is obtained thanks to a new *max register* construction with $O(\log n)$ amortized step complexity in executions of arbitrary length in which the value stored in the register does not grow too quickly. We then leverage an existing counter algorithm by Aspnes, Attiya and Censor-Hillel [AAC12] in which we "plug" our max register implementation to show that it remains linearizable while achieving $O(\log^2 n)$ amortized step complexity.

Keywords Shared Memory · Wait-freedom · Counter · Amortized Complexity · Concurrent Objects

A preliminary version of this work appeared in DISC'19. Mirza Ahad Baig, Alessia Milani and Corentin Travers are supported by ANR projects Descartes and FREDDA. Mirza Ahad Baig is supported by UMI Relax. Danny Hendler is supported by the Israel Science Foundation (grants 380/18 and 1425/22).

Ahad Mirza Baig
Institute of Science and Technology, Austria
E-mail: ahad.baig@ist.ac.at

Danny Hendler
Ben-Gurion University of the Negev, Israel
E-mail: hendlerd@bgu.ac.il

Alessia Milani
LIS, Aix-Marseille University, France
E-mail: alessia.milani@lis-lab.fr

Corentin Travers
LIS, Aix-Marseille University, France
E-mail: corentin.travers@lis-lab.fr

1 Introduction

A shared-memory *counter* [MTY96] is a well-studied [AC10, AH90, AKK$^+$14, BG11, MT97] and widely-used concurrent object. A counter stores a non-negative integer and supports two operations: An `Inc` operation that increases its value by 1 and a `Read` operation that returns its current value.

A wait-free counter can be constructed easily by using a *single-writer atomic snapshot* [And93, AAD$^+$93, AH90] object. Such an object allows each process to update its own component (by invoking an `Update` operation) and to obtain an atomic view of all components (by invoking a `Scan` operation). To increment the counter, a process p simply increments its component. To read the counter's value, p invokes `Scan` and returns the sum of all components in the view it obtains. Since wait-free atomic snapshots can be implemented from read-write registers with step complexity linear in the number of processes n [AF01,IC94], so can counters.

A well-known result [JTT00] by Jayanti, Tan and Toueg showed this is tight. They prove a linear lower bound on the worst-case step complexity of obstruction-free implementations of a large class of shared objects, including counters, from operations in a set that includes (among some other operations) read and write. In [AAC12], Aspnes, Attiya and Censor-Hillel observed that the lower bound holds only when numerous operations are applied to the object and does not rule out the existence of algorithms whose step complexity is sub-linear when the number of operations is bounded. Leveraging this observation, they presented constructions of several data structures for which operations' step complexity is polylogarithmic in n as long as the object's value is polynomial in n, where n is the number of processes. More precisely, they presented a wait-free counter for which the step complexity of `Inc` operation is $O\big(min(\log n \log v, n)\big)$ and $O\big(min(\log v, n)\big)$ for `Read` operations, where v is the object's current value. However, the worst-case and amortized step complexities of the counter algorithm of [AAC12] deteriorate as the number of `Inc` operations increases. For executions in which the number of `Inc` operations is exponential in n, both the worst-case and the amortized step complexities become the same as those of the snapshot-based algorithm, that is, linear in n.

Our contribution. The lower bound of [JTT00] leaves open the question of whether there exists a counter algorithm with sub-linear *amortized* step complexity. In this paper, we answer this question in the affirmative, by showing that linearizable and wait-free counters for n processes can be implemented from read-write registers with polylogarithmic amortized step complexity. Intuitively, an implementation is wait-free if every process can complete its operation in finitely many steps, regardless of the behavior of the other processes. This is the first wait-free counter from read-write registers that provides sub-linear amortized step complexity in *executions of arbitrary length.* We reuse the counter algorithm presented in [AAC12]. Their counter algorithm uses max registers, an object type they introduced and implemented. A *max register* r supports a `WriteMax`(r,v) operation that writes a non-negative integer v to r and a `ReadMax`(r) operation that returns the maximum value previously written to r.

We present a novel wait-free deterministic implementation of an unbounded max register and "plug it" into the counter algorithm of [AAC12]. We show that the resulting counter remains linearizable, is wait-free and has $O(\log^2 n)$ amortized step complexity. The worst-case step complexity is $O(n)$, which is optimal [JTT00]. Aspnes et al. also presented an unbounded max register, however the step complexities of both `ReadMax` and `WriteMax` operations in their algorithm are $O\big(min(\log v, n)\big)$, where v is the object's current value. Thus, executions of arbitrary length can have linear amortized step complexity. Aspnes and Censor-Hiller [AC13] presented an unbounded max register implementation for which every operation terminates in a constant number of steps with high probability, under the assumption that the max register's value does not grow too quickly. Our implementation of unbounded max register makes a similar assumption. The max register algorithm of [AC13] is randomized, whereas ours is deterministic. The space complexity of our implementation is unbounded.

Using information-theoretic arguments, Jayanti established a logarithmic lower bound on the *worst-case* operation step complexity for obstruction-free implementations of a set of one-time objects that includes a `fetch&increment` object, from operations such as load-linked/store-condition, move and swap [Jay98]. Attiya and Hendler [AH10] presented lower bounds on the time and space complexities of obstruction-free implementations of several objects from k-word compare-and-swap operations. Using as well an information-theoretic argument as well, they proved [AH10, Theorem 9] a logarithmic lower bound on the *amortized* step complexity of implementing a one-time `fetch&increment` object in an obstruction-free manner. Their proof can be modified in a straightforward manner to establish the same result for counters, implying that the amortized step complexity of our algorithm is at most within a logarithmic factor of the optimal.

Related work. Counting networks [AHS94], presented by Aspnes, Herlihy and Shavit, allow processes to assign themselves successive values in a given range. They are similar to Batcher's sorting networks [Bat68], except that instead of comparators, they are constructed by interconnecting simple objects called *balancers.* A balancer has several input and output wires and balances tokens received on its input wires to its output wires. Balancers are typically implemented from registers supporting read-modify-write operations. To obtain a number, processes shepherd tokens from an input wire of the network to an output wire. The step complexity thus depends on the number of traversed balancers, which can be as low as $O(\log n)$ [DS00] for n processes. Counting networks are however, in general, not linearizable. Herlihy, Shavit and Waarts have shown a lower bound of $\Omega(n)$ [HSW96] on the depth of n-processes counting networks that are linearizable.

A `fetch&increment` object stores a non negative integer and supports a single operation that increments the value stored into the object and returns the previous value. It cannot be implemented deterministically in a wait-free manner from read-write registers since its consensus number is 2 [Her91]. Optimal implementations for n processes from load-link/store conditional objects are presented in [EW13] by Ellen and Woelfel, in which each `fetch&increment` operation has $O(\log n)$ step complexity. A fast implementation of a counter from compare-and-swap is presented by Khanchandani and Wattenhofer in [KW17]. The step complexity of `Inc` is $O(\log n)$ and constant for `Read`. Our counter implementation requires only read/write registers but has $O(\log^2 n)$ *amortized* step complexity for each operation. The algorithms presented in [EW13, KW17] have bounded space complexity: The number of base objects they use is bounded by a polynomial function of n. Unlike theirs, our algorithm has unbounded space complexity.

Randomized is another approach to beat the lower bound of [JTT00]. A randomized approximate counter from read-write registers is presented by Aspnes and Censor [AC10] with step complexity $O((\frac{1}{\delta}\log n)^{O(\frac{1}{\epsilon})})$ for `Inc` and $O(n^{4/5+\epsilon}((\frac{1}{\delta}\log n)^{O(\frac{1}{\epsilon})}))$ for `Read`, where $\epsilon > 0$ is a small constant, n is the number of increments and δ is the approximation ratio the counter achieves with high probability.

As mentioned previously, a deterministic implementation of a counter for n processes from read-write registers is presented in [AAC12]. The step complexities are $O(\min(\log n \log v, n))$ and $O(\min(\log v, n))$ and for `Inc` and `Read` operations and for operations respectively. Here, v is the current value of the counter. The algorithm of [AAC12] uses max registers as building blocks. A linearizable implementation of a max register from read-write registers with $O(\min(\log v, n))$ step complexity for reading or writing a value v is also given in [AAC12]. In Section 3, we present a novel implementation of a max register from read-write registers and show that it achieves for `ReadMax` and `WriteMax` operations $O(\log n)$ *amortized* step complexity in executions in which the value stored in the register does not grow too quickly.

Our max register construction shares some similarities with the one of [AAC12]. Both constructions might be seen as binary trees in which internal nodes are switches (a bit stored in a read-write register) and leaves are bounded max register. Specifically, leaves are trivial 0-bounded max-registers in the case of [AAC12] and m-bounded in our case, with m a function of n. Unlike the implementation of [AAC12], our algorithm handles an unbounded number of operations in a wait-free manner without resorting to a snapshot-based implementation. The linearizability proof in [AAC12] is recursive. Such a proof cannot be applied to our algorithm, because, unlike [AAC12], our construction is non-recursive. Also, differently from [AAC12], in our algorithm a process writing a value to the max register accesses only a constant number of nodes. Another difference is that our algorithm employs a helping mechanism so that the max register can be read in a wait-free manner with linear worst-case step complexity.

Unlike [AAC12], correctness and logarithmic amortized step complexity are only guaranteed in executions in which there is a bound on the increments of the value of the max register. We establish in Section 4 that the way values change in the max registers used in the counter construction of [AAC12] satisfies this restriction. Hence, by plugging our max register implementation into the construction of [AAC12], we obtain a counter supporting an unbounded number of operations and whose amortized step complexity is polylogarithmic in the number of processes.

Moran, Taubenfeld and Yadin [MTY96] defined the notion of a concurrent counter that may assume values from $\{0, \ldots, m-1\}$, for some positive integer m, for which `increment` operations are modulo m. They define and investigate two notions of counters. A *static counter* guarantees the correctness of `increment` operations but allows a `read` operation[1] that is concurrent to an `increment` operation to return an arbitrary value. The second, stronger, notion is a *dynamic counter*, for which `read` operations must be linearizable regardless of whether or not they are concurrent to `increment` operations. He processes are anonymous in their model. They investigate the space complexity of implementing both static and dynamic counters from binary registers that support either reads and writes only, or stronger read-modify-write operations. Among other results, a wait-free static counter algorithm that uses $\log m$ bits and a wait-free dynamic counter algorithm that uses m bits are presented. For both these algorithms, the number of processes that may invoke the `increment` operation is unbounded. They also present a lower bound on the space complexity of dynamic counters. The counters investigated by our work (as well as by the works we previously described) are dynamic and may assume unbounded values. Unlike [MTY96], our model also assumes that processes have unique identifiers.

Organisation. The rest of this article is organized as follows. We present the system model we assume and additional required definitions in Section 2. In Section

[1] They use the name `look` rather than `read`.

3, we present our key technical contribution – an unbounded max register algorithm that guarantees linearizability and logarithmic amortized step complexity when its value is not increased "too quickly". In Section 4, we prove that by "plugging" our unbounded max register into the counter algorithm of [AAC12] (instead of using the max register algorithm of [AAC12]) we obtain a linearizable counter with polylogarithmic amortized step complexity. The article is concluded with a discussion in Section 5.

2 Model and Preliminaries

Shared memory. We consider a standard *asynchronous shared memory system* in which a set $\mathcal{P}$ of n processes that communicate by accessing shared *registers*. A register `r` stores a value from some set and supports two operations: `Read(r)` which returns the value of the register and `Write(r,`v`)` which writes the value v to `r`.

An *implementation* of a concurrent object specifies the object's state representation and which algorithms processes follow when they perform operations supported by the object. An *execution* is a sequence of *steps* performed by the processes as they follow their algorithms, in each of which a process invokes an operation, returns an operation response, or applies at most a single `Read` or `Write` operation to a register (possibly in addition to some local computation). An execution is *well-behaved* if no process invokes an operation on an object before having received a response from its previous invocation on the same object. In what follows, we consider only well-behaved executions.

The *execution interval* of an operation starts with the operation's invocation and ends when a response is returned. An operation *op precedes* another operation *op′* if the execution interval of *op* ends before the execution interval of *op′* starts. *op* and *op′* are *concurrent* if their execution interval intersect. An operation is *complete* in an execution if it returned a response in the execution. An execution e is *linearizable* [HW90] if, for all completed operations in e and some of the uncompleted operations in e, there is a point in the execution interval of the operation, called its *linearization point*, such that the responses returned are the same as the responses returned if all these operations were executed sequentially following the order of their linearization points. An implementation is linearizable if all its executions are linearizable; it is *wait-free* [Her91] if, in every execution, each process completes its operation within a finite number of steps; it is *lock-free* if, in every execution, at least one process completes its operation after performing a finite number of steps. An implementation is *obstruction-free* [HLM03] if after any finite execution, any process can complete its operation by taking a bounded number of steps when no other process takes a step.

Complexity measure. The *amortized step complexity* is defined as the worst-case (taken over all possible finite executions) average number of steps performed by operations. It measures the performance of an implementation as a whole rather than the performances of individual operations. Indeed, in an execution of a lock-free implementation, some operations may never terminate and the worst-case operation step complexity may thus be unbounded. Amortized step complexity is formally defined as follows. We denote by $nsteps(op, e)$ the number of steps performed by an operation *op* in e and by $OP(e)$ the set of operations that are invoked in e. The amortized step complexity of an implementation A is then:

$$AmtSteps(A) = \max_{e\text{: finite execution of } A} \frac{\sum_{op \in OP(e)} nsteps(op, e)}{|OP(e)|}.$$

Max registers. A *max register* `MaxReg` supports two operations: `WriteMax(MaxReg,`v`)` writes v to `MaxReg` where v is a non-negative integer. `ReadMax(MaxReg)` returns the maximum value previously written. If no value has been previously written, it returns 0 which is the initial value of the max register. For an integer $m > 0$, a *bounded* max register $\texttt{MaxReg}_m$ is a max register for which the input v of any `WriteMax` operation is restricted to the set $\{0, \ldots, m-1\}$. An *unbounded* max register `UnboundedMaxReg` can store any non-negative integer.

3 Polylogarithmic Amortized Step Complexity Max Register

The pseudo-code of our unbounded max register is presented in Algorithm 1. Lines in black font constitute a lock-free version of the algorithm, which we describe and analyze in this section. Lines in lighter (metal) color add a helping mechanism that makes the algorithm wait-free. For presentation simplicity, we defer the description of this mechanism to Subsection 3.3.

We proceed with a description of Algorithm 1. An $\texttt{UnboundedMaxReg}_m$ object M consists of an infinite number of shared bounded $\texttt{MaxReg}_m$ max registers, denoted $\texttt{max}_j$, for all $j \in \mathbb{N}_0$. Register $\texttt{max}_j$ will be used for representing values in the range $[m \cdot j, m \cdot (j+1) - 1]$. Hence, the subscript m in the type $\texttt{UnboundedMaxReg}_m$ refers to the bound m of the bounded max registers used by objects of this type. Each bounded max register $\texttt{max}_j$ is associated with a shared $\texttt{switch}_j$ bit which is stored in a read/write register. All max registers and their corresponding switches are initialized to 0. Each process i has a local variable $\texttt{last}_i$, storing the index j of the bounded register $\texttt{max}_j$ that will be accessed next by i's `Read` operation. $\texttt{last}_i$ is initialized to 0 for each process i.

The ***`Write`*** *function.* To write value v, process i first computes the index k of the bounded max register to write to and the residue v' to be written to it (Lines 2-3). Here and in what follows, the residue v' of v is the remainder of the division of v by m. Next, i checks in Line 4 whether $\texttt{max}_k$ is *obsolete*. We say that a (bounded) max register is obsolete if its corresponding switch is set, indicating that values were already written to max registers with higher indexes and thus $\texttt{max}_k$ should no longer be accessed. If $\texttt{max}_k$ is obsolete, i does not need to write to it, so it proceeds to Line 12 for increasing its *last* index, if required, and returns. Otherwise, $\texttt{max}_k$ is not obsolete, so i writes to it the residue v' (Line 5). If the max object written to is not the first (Line 6), then i ensures that the previous max object is obsolete (Lines 8-11), updates its *last* index (Line 12), if required, and returns.

The ***`Read`*** *function.* Process i scans the switches in increasing order in Lines 15-16, increasing the value of its `last` index in the process, until it finds the first non-obsolete bounded max register (this might never happen.). If it does, it reads the maximum residue previously written to that max object (Line 19), adds to the residue a multiple of m corresponding to the index of that max register and returns the sum (Line 20).

Algorithm 1 Unbounded Max Register $\texttt{UnboundedMaxReg}_m$, code for process i.

Shared variables:

$\texttt{switch}_j \in \{0,1\}$: a 1-bit register for each $j \in \mathbb{N}_0$, initially all 0

$\texttt{max}_j$: a $\texttt{MaxReg}_m$ object for each $j \in \mathbb{N}_0$, initially all 0

$\texttt{last}_i \in \mathbb{N}_0$: smallest index j such that process i has not yet accessed $\texttt{max}_j$, initially 0

$\texttt{H}[n]$ initially all $(-1,0,-1)$: helping array of integer-triplets, entry i written by process i

`hCount` initially 0: an integer storing the number of times i wrote to $\texttt{H}[i]$

1: **function** $\texttt{Write}(\texttt{UnboundedMaxReg}_m, v)$
2: $\quad v' \leftarrow v \bmod m$
3: $\quad k \leftarrow \lfloor \frac{v}{m} \rfloor$
4: $\quad$**if** $\texttt{switch}_k = 0$ **then**
5: $\quad\quad \texttt{WriteMax}(\texttt{max}_k, v')$
6: $\quad\quad$**if** $k > 0$ **then**
7: $\quad\quad\quad curMax \leftarrow \texttt{ReadMax}(\texttt{max}_{k-1}) + (k-1)\cdot m$
8: $\quad\quad\quad$**if** $\texttt{switch}_{k-1} = 0$ **then**
9: $\quad\quad\quad\quad \texttt{hCount} \leftarrow \texttt{hCount} + 1$
10: $\quad\quad\quad\quad \texttt{H}[i] \leftarrow (k-1, \texttt{hCount}, curMax)$
11: $\quad\quad\quad\quad \texttt{switch}_{k-1} \leftarrow 1$
12: $\quad \texttt{last}_i \leftarrow \max(k, \texttt{last}_i)$
13: **function** $\texttt{Read}(\texttt{UnboundedMaxReg}_m)$
14: $\quad$**local** c initially 0
15: $\quad$**while** $\texttt{switch}_{\texttt{last}_i} \neq 0$ **do**
16: $\quad\quad \texttt{last}_i \leftarrow \texttt{last}_i + 1$, $c \leftarrow c + 1$
17: $\quad\quad$**if** $(c \bmod (n+2)) = 0$ **then**
18: $\quad\quad\quad$**if** $(hval \leftarrow \texttt{GetHelp}(c)) > 0$ **then return** $hval$
19: $\quad v \leftarrow \texttt{ReadMax}(\texttt{max}_{\texttt{last}_i})$
20: $\quad$**return** $v + (\texttt{last}_i \cdot m)$

3.1 Linearizability

The correctness of Algorithm 1 is guaranteed only in executions in which the max register's value is increased in bounded increments. This requirement is formalized by the following definition.

Definition 1 (ℓ-Bounded-Increment Execution) Let M be an `UnboundedMaxReg` object and let e be an execution. e is an ℓ-bounded-increment execution for M if for each write operation $op = \texttt{Write}(M, v)$ in e, with $v > \ell$, there exists a write operation $op' = \texttt{Write}(M, v')$ in e that precedes op, such that $v - \ell \leq v' < v$.

Section 4 presents an n-process unbounded counter implementation that uses `UnboundedMaxReg` objects. As we prove, all the executions of that implementation are n-bounded-increment executions for all the underlying unbounded max registers.

Let $m \geq n$, M be an $\texttt{UnboundedMaxReg}_m$ object, implemented by Algorithm 1, and let e be a finite and n-bounded-increment execution for M. The next lemma is a direct consequence of Definition 1.

Lemma 1 *Let op be a* ***`Write`*** *operation on M with input v. If $\lfloor \frac{v}{m} \rfloor > 0$, there exists a* ***`Write`*** *operation op' that precedes op and whose input v' is such that $\lfloor \frac{v'}{m} \rfloor = \lfloor \frac{v}{m} \rfloor - 1$.*

Proof Let op_0 be a `Write` operation on M and let v_0 be its input value. Let us assume that $\lfloor \frac{v_0}{m} \rfloor = k > 0$. Since e is an n-bounded increment operation for M, there exists a `Write` operation op_1 on M whose input v_1 satisfies $v_0 - n \leq v_1 < v_0$ that precedes op_0 (Definition 1). In particular, this means that $v_0 > v_1$ and $\lfloor \frac{v_1}{m} \rfloor \in \{\lfloor \frac{v_0}{m} \rfloor - 1, \lfloor \frac{v_0}{m} \rfloor\} = \{k-1, k\}$ since $n \leq m$. If $\lfloor \frac{v_1}{m} \rfloor \neq k-1$, we repeat the same argument to identify a finite sequence of `Write` operations $op_2, \ldots, op_\ell$ on M with respective inputs $v_2, \ldots, v_\ell$ satisfying, for each $i, 2 \leq i \leq \ell$, that op_i precedes op_{i-1}, $v_i < v_{i-1}$, and $\lfloor \frac{v_i}{m} \rfloor \in \{\lfloor \frac{v_{i-1}}{m} \rfloor, \lfloor \frac{v_{i-1}}{m} \rfloor - 1\}$. Hence, there must exist a `Write` operation op' that precedes op and whose input

v' is such that $\lfloor \frac{v'}{m} \rfloor = k - 1$ (for example, op' can be taken as the operation with the smallest index i in the sequence such that $\lfloor \frac{v_i}{m} \rfloor < \lfloor \frac{v_{i+1}}{m} \rfloor$).

If a switch is set in e, let K be the largest index of the switches that are set in e. Since e is finite, there are finitely many operations that are invoked in e. As each operation on M sets at most one switch, K is well-defined. Otherwise, let $K = -1$. For each $k, 0 \leq k \leq K$, let s_k denote the step in which 1 is written to $\texttt{switch}_k$ (at Line 11) for the first time. We observe that switches are set in order:

Lemma 2 *For each integer $k, 0 < k \leq K$, s_{k-1} occurs before s_k in e.*

Proof Let $k > 0$. By the code, the value of $\texttt{switch}_k$ is changed to 1 by a `Write` operation *op* (at Line 11) on the unbounded max register M whose input v is such that $\lfloor \frac{v}{m} \rfloor = k + 1$. By Lemma 1, *op* is preceded by a `Write` operation op' with input value v' satisfying $\lfloor \frac{v'}{m} \rfloor = k$. Since op' precedes *op* and $\texttt{switch}_k$ is set for the first time during the execution of *op*, the value of $\texttt{switch}_\texttt{k}$ is 0 when it is read by op' (at Line 4). As $k > 0$, Lines 6-11 are performed by op'. In particular, if the value of $\texttt{switch}_{k-1}$ is not already 1, it is changed to 1 by op' at Line 11. It thus follows that s_{k-1} precedes s_k.

We observe that when a `Write` operation on M whose input v is such that $\lfloor \frac{v}{m} \rfloor = k > 0$, the max register $\texttt{max}_{k-1}$ becomes obsolete before the operation terminates.

Observation 1 *Let op be a completed `Write` operation on M with input v. If $\lfloor \frac{v}{m} \rfloor > 0$, $s_{\lfloor \frac{v}{m} \rfloor - 1}$ occurs before op terminates.*

Proof Let *op* be a completed `Write` operation on M and let v be its input. Let us assume that $k = \lfloor \frac{v}{m} \rfloor > 0$. Since $k = \lfloor \frac{v}{m} \rfloor$, $\texttt{switch}_k$ is read by *op* on Line 4. If this read returns 1, the observation follows by Lemma 2. Otherwise, since $k > 0$, Lines 6-11 are executed during *op*. In particular, if $\texttt{switch}_{k-1}$ is not already set (Line 8), 1 is written to it by *op* in Line 11. Hence $\texttt{switch}_{k-1}$ is set before the end of *op*.

We say that a bounded max register $\texttt{max}_k$ is *active* during the interval in which its associated register $\texttt{switch}_k$ is not set, but $\texttt{switch}_{k-1}$ is. We define intervals $I_0, \ldots, I_{K+1}$ in which the bounded max registers $\texttt{max}_0, \ldots, \texttt{max}_{K+1}$ are active:

- Interval I_0 starts with the beginning of e and ends immediately before s_0.
- For $k, 1 \leq k \leq K$, interval I_k begins with s_{k-1} and ends immediately before s_k.
- Interval I_{K+1} begins with s_K and ends with the last step of e.

By Lemma 2, these intervals are well defined, in the sense that their beginning precedes their end. Note also that $I_0, \ldots, I_{K+1}$ form a partition of e.

We observe that each `Read` or `Write` operation on M accesses at most one of the bounded max register $\texttt{max}_0, \texttt{max}_1, \ldots$ (in Line 5 for a `Write` operation, and in Line 19 for a `Read` operation). For $k \geq 0$, let A_k be the set of operations on M in e that access the bounded max register $\texttt{max}_k$ (by performing a `WriteMax` in Line 5 in the case of a `Write` operation or a `ReadMax` in Line 19 in case of a `Read`). Let also B be the set of `Write` operations in e that perform a read of $\texttt{switch}_k$ at Line 4, for some $k \geq 0$ and that read returns 1.

As e is a finite n-bounded increment execution for M, the sets A_k are empty for large enough values of k:

Lemma 3 *Let $k \geq K + 3$. $A_k = \emptyset$.*

Proof The proof is similar to the proof of Lemma 2. Let $k \geq K + 3$ and let us assume towards a contradiction that there is an operation $op \in A_k$. Since the largest index of a switch that is set is K, and for a `Read` operation to access the bounded max register $\texttt{max}_k$, $\texttt{switch}_{k-1}$ has to be set, *op* is not a read operation. Hence, *op* is a `Write` operation, and as it accesses the bounded max register $\texttt{max}_k$, its input value v satisfies $\lfloor \frac{v}{m} \rfloor = k$. By Lemma 1, *op* is preceded by a `Write` operation op' whose input v' satisfies $\lfloor \frac{v'}{m} \rfloor = k - 1$. When op' terminates, it follows from Observation 1 that $\texttt{switch}_{k-2}$ is set. Since $k - 2 \geq K + 1$, a `switch` with index strictly larger than K is set in e, contradicting the definition of K.

Before defining linearization points, we show that each operation in A_k performs (at least) some of its steps when the bounded max register $\texttt{max}_k$ is active. In what follows, I_{op} denotes the execution interval of operation *op*.

Lemma 4 *For every $k, 0 \leq k \leq K + 1$ and for every operation $op \in A_k$, $I_k \cap I_{op} \neq \emptyset$.*

Proof Let $k, 0 \leq k \leq K + 1$ and let *op* be an operation in A_k. The proof is divided into three cases according to the value of k:

- $k = 0$. For *op* to access the bounded max register $\texttt{max}_0$, it has to read 0 from $\texttt{switch}_0$ (at Line 4 for a `Write` operation or at Line 15 for a `Read` operation.). This steps occurs after the beginning of e and before $\texttt{switch}_0$ is set, as once the value of a switch is changed to 1, it never changes. It thus follows that $I_{op} \cap I_0 \neq \emptyset$.

- $0 < k \leq K$. If *op* is a `Read` operation, it accesses the bounded max register $\texttt{max}_k$ (at Line 19) only if it has read 1 from $\texttt{switch}_{k-1}$ before reading 0 from $\texttt{switch}_k$. From the definition of interval I_k, this latter read occurs in I_k. If *op* is a `Write` operation, its read of $\texttt{switch}_k$ (at Line 4) returns 0. The execution interval of *op* thus contains a step performed before s_k. If *op* does not terminate (i.e., its execution interval ends with the end of e.), $I_{op} \cap I_k \neq \emptyset$. Otherwise, *op* terminates and when it does, $\texttt{switch}_{k-1}$ is set (Observation 1). Hence the execution interval of *op* contains the step s_{k-1} or a step performed after s_{k-1} and a a step performed before s_k. Thus $I_{op} \cap I_k \neq \emptyset$.
- $k = K+1$. For `Read` operation *op* in A_{K+1}, the proof is similar to the previous case. *op* has to read 1 from $\texttt{switch}_K$ and 0 from $\texttt{switch}_{K+1}$ in order to access the bounded max register $\texttt{max}_{K+1}$. This latter read occurs between s_K and the end of e.
  If *op* is a `Write` operation that accesses $\texttt{max}_{K+1}$ and does not terminate, its execution interval intersects I_{K+1}. Otherwise, as seen in the previous case, when *op* terminates $\texttt{switch}_K$ has been set (by *op* itself or by another operation). Hence, I_{op} contains s_K or a step performed after s_K and thus $I_{op} \cap I_K \neq \emptyset$.

To show that M is linearizable in e, we rely on a linearization μ of the `ReadMax` and `WriteMax` operations performed on the bounded max registers $\texttt{max}_0, \ldots, \texttt{max}_{K+1}$. As linearizability is composable and a linearizable bounded max register can be implemented from read-write registers [AAC12], μ exists.

We next define the linearization points of the operations in $\bigcup_{0 \leq k \leq K+1} A_k \cup B$:

- The linearization points of the operations in A_k are chosen as follows: each operation $op \in A_k$ is given a linearization point λ_{op} in the interval $I_k \cap I_{op}$ preserving the order in which the corresponding `ReadMax`/`WriteMax` operations on $\texttt{max}_k$ are linearized in μ. That is, for any two operation $op, op' \in A_k$, if the `ReadMax` or `WriteMax` performed by *op* on $\texttt{max}_k$ is linearized before the one performed by op' in μ, λ_{op} is before $\lambda_{op'}$.
- An operation *op* in B is linearized with the read of $\texttt{switch}_k$ performed by that operation (on Line 4).

By Lemma 4, the linearization points of all operations *op* in A_k, for $k, 0 \leq k \leq K+1$, are well defined since the intersection between the execution interval of *op* and I_k is not empty. Hence, each operation in $\bigcup_{0 \leq k \leq K+1} A_k \cup B$ is linearized within its execution interval.

Lemma 5 *The set* $\bigcup_{0 \leq k \leq K+1} A_k \cup B$ *contains every completed operation in* e.

Proof If all operations invoked in e are in $\bigcup_{0 \leq k \leq K+1} A_k \cup B$, then clearly the lemma is true. Assume, then, that there are operations that complete in e and are not in the set $\bigcup_{0 \leq k \leq K+1} A_k \cup B$. Let *op* be such an operation. If *op* accesses a bounded max register, it belongs to A_{K+2} by Lemma 3. *op* cannot be a `Read` operation, as a `Read` that accesses $\texttt{max}_{K+2}$ must read 1 from $\texttt{switch}_{K+1}$ (Line 15), but this switch is never set in e. *op* is thus a `Write` operation. By Observation 1, when a `Write` operation accessing $\texttt{max}_{K+2}$ terminates, $\texttt{switch}_{K+1}$ has been set. As $\texttt{switch}_{K+1}$ is never set in e, *op* does not terminate.

It remains to examine the case in which *op* does not access any bounded max register. If *op* is a `Write` operation, since it is not in set B, it has not read a switch in e, or has read 0 from some switch but has not yet performed a `WriteMax` to the corresponding bounded max register when e ends. In both cases, *op* does not terminate in e. If *op* is a `Read` operation, it has only read 1 from the switches it has accessed in e, or it has read 0 from some switch but has not yet performed a `ReadMax` to the corresponding bounded max register when e ends. Hence *op* does not terminate in e.

Finally, we show that the linearization is consistent with the sequential specification of a max register.

Lemma 6 *Let* $op \in \bigcup_{0 \leq k \leq K+1} A_k$ *be a* `Read` *operation that returns a value* v.

- *If* $v = 0$, *there is no* `Write` *operation with an input* $\neq 0$ *that is linearized before op.*
- *If* $v \neq 0$, *the largest input value of the* `Write` *operations linearized before op is* v.

Proof Let *op* be a `Read` operation that returns v. Let u, k be integers such that $v = k \cdot m + u$ and $0 \leq u < m$. Since *op* returns v, it follows from the code that it has performed a `ReadMax` on the bounded max register $\texttt{max}_k$ that returned u (Line 19 and Line 20). Hence, *op* belongs to A_k, for some $k \in \{0, \ldots, K+1\}$.

If $v = 0$, we have $k = u = 0$. Let us assume that there is a `Write` operation op' linearized before *op*. Let v' be the input of that operation. We first note that op' cannot belong to B. Indeed, any operation $op'' \in B$ reads 1 from a register $\texttt{switch}_k$, for some $k \geq 0$. op'' is thus linearized after this switch has been set to 1, which occurred after I_0 by Lemma 2 (recall that as $op \in A_0$, its linearization point is in the interval I_0). Hence any operation in B is linearized after *op*.

op' thus belongs to A_k for some $k \geq 0$. Since every operation in A_k, for every $k > 0$, is linearized after the operations in A_0, op' must be in A_0. As op' is linearized before *op*, its associated `WriteMax` operation on $\texttt{max}_0$ is linearized in μ before the `ReadMax` performed by *op*.

Since *op* reads $u = 0$ from the max register $\texttt{max}_0$, the value written by the `WriteMax` of op' is also 0. It thus follows that the input value v' of op' is $v' = 0 \cdot m + 0 = 0$.

We now consider the case $v \neq 0$. We first show that there is a `Write` operation op' with input v that is linearized before *op*. We then establish that any `Write` operation with input strictly greater than v (if any) is linearized after *op*.

1. The first part is divided into two cases, depending of the value of $u = v \mod m$.
   - $u = 0$. Note that, as $v \neq 0$, $k > 0$. Thus *op* performs a `ReadMax` on the bounded max register $\texttt{max}_k$ which returns 0. By the code, this happens after *op* has read 1 from the register $\texttt{switch}_{k-1}$. Let op' be the `Write` operation that sets this switch to 1, and let v' be its input value. Before writing 1 to $\texttt{switch}_{k-1}$, op' has performed a `WriteMax` on the bounded max register $\texttt{max}_k$ (Line 5) with some input value u'. Since the `ReadMax` on $\texttt{max}_k$ performed by *op* returns 0 and follows this `WriteMax`, $u' = 0$. Therefore the input value v' of op' is $v' = k \cdot m + u' = k \cdot m = v$. Since both *op* and op' perform an operation on $\texttt{max}_k$, they belong to A_k. As the `WriteMax` by op' precedes the `ReadMax` by *op*, op' is linearized before *op*.
   - $u \neq 0$. Recall that u is the value read from the bounded max register $\texttt{max}_k$ by *op*. Hence, there is a `WriteMax` operation on $\texttt{max}_k$ with input u that is linearized before this `ReadMax` in μ. By the code (Line 5), the `WriteMax` operation with input u is performed within a `Write` operation with input $v' = k \cdot m + u = v$. Let op' be this operation. Since op' accesses $\texttt{max}_k$, it belongs to A_k. Since its `WriteMax` is linearized in μ before the `ReadMax` of *op*, op' is linearized before *op*.
2. Let op' be a `Write` operation with input $v' > v$. Note that $k' = \lfloor \frac{v'}{m} \rfloor \geq \lfloor \frac{v}{m} \rfloor$. If $op' \in B$, it is linearized after $\texttt{switch}_{k'}$ has been set to 1. By Lemma 2, this occurs after the interval I_k in which *op* is linearized. Otherwise $op' \in A_{k'}$. If $k' > k$, op' is linearized after *op*. If $k' = k$, both op' and *op* access the bounded max register $\texttt{max}_k$, and are thus linearized in the interval I_k. As $v' > v$, the input $u' = v' \mod m$ written to $\texttt{max}_k$ by op' is strictly larger that the value $u = v \mod m$ read by *op*. Hence the `ReadMax` to $\texttt{max}_k$ by *op* must be linearized in μ before the `WriteMax` performed by op'. Thus *op* is linearized before op'.

Lemma 7 *Algorithm 1 without the helping mechanism is lock-free and* `Write` *operations are wait-free.*

Proof `Write` operations perform a single invocation of the wait-free `WriteMax` operation and a constant number of additional steps, hence they are wait-free. A `Read` operation may loop forever in Lines 15-16, searching for a non-obsolete max register, but only if `Write` operations keep making additional max registers obsolete (in Line 11), hence more and more `Write` operations complete. If no more `Write` operations complete, each `Read` operation is guaranteed to complete.

3.2 Step Complexity Analysis

The step complexity analysis provided in this section relates to the implementation of Algorithm 1 without the helping mechanism. Recall that $OP(e)$ denotes the set of all operations that are invoked in e. Let $OP_R(e)$ (resp. $OP_W(e)$) denote the set of all `Read` operations (resp. all `Write` operations) that are invoked in e. For an operation *op*, we let $nsteps(op, e)$ denote the number of steps performed by *op* in e.

Lemma 8 *If $m \geq n^2$, then the $\texttt{UnboundedMaxReg}_m$ implementation of Algorithm 1 has amortized step complexity of $O(\log m)$ in any n-bounded-increment execution.*

Proof Let e be an n-bounded-increment execution. We wish to bound:

$$AmtSteps(e) = \frac{\sum\limits_{op \in OP(e)} nsteps(op, e)}{|OP(e)|}.$$

Let r be the number of `Read` operations and w be the number of `Write` operations in $OP(e)$. `WriteMax` and `ReadMax` operations on an m-bounded max register perform $O(\log m)$ steps each. Clearly from the pseudo-code of Algorithm 1, each `Write` operation performs a constant number of steps in addition to possibly invoking a single `WriteMax` operation, thus the step complexity of each `Write` operation is $O(\log m)$.

A `Read` operation *op* performs $loop_{op} + O(\log m)$ steps, where $loop_{op}$ is the number of steps performed in the while loop of Lines 15-16 and $O(\log m)$ is the number of steps performed by the invocation of `ReadMax` in Line 19. We get:

$$AmtSteps(e) = O\left(\frac{\sum_{op \in OP_W(e)} \log m + \sum_{op \in OP_R(e)} \log m + loop_{op}}{w + r}\right)$$

If $r = 0$, then clearly $AmtSteps(e) = O(\log m)$. So assume that $r > 0$. From Lines 12 and 16, for every process i, $last_i$ never decreases and is incremented once in every iteration of the while loop of Lines 15-16. Therefore:

$$\sum_{op \in OP_R(e)} loop_{op} = O\Big(r + \sum_{i\in\mathcal{P}} last_i\Big).$$

Consequently,

$$AmtSteps(e) = O\left(\frac{w \cdot \log m + r \cdot \log m + (r + \sum_{i\in\mathcal{P}} last_i)}{w + r}\right).$$

Assume that max register $\mathtt{max}_k$, for $k > 0$, is accessed in e. Since e is an n-bounded-increment execution and all $\mathtt{max}_j$ registers are m-bounded, at least $\frac{m \cdot k}{n}$ *Write* operations have been linearized prior to this access. Letting $\mathcal{L} = \max_{i\in\mathcal{P}} last_i$ denote the maximum value of all $last_i$ variables at the end of e, we get that $w \geq (\mathcal{L} - 1) \cdot m/n$. Furthermore, $\sum_{i\in\mathcal{P}} last_i \leq n \cdot \mathcal{L}$. Thus,

$$\begin{aligned} AmtSteps(e) &= O\left(\frac{w \log m + r \log m + (r + n \cdot \mathcal{L})}{w + r}\right) \\ &= O\left(\frac{(w + r)\log m}{w + r} + \frac{r}{w + r} + \frac{n \cdot \mathcal{L}}{w + r}\right) \\ &= O\left(\log m + \frac{n \cdot \mathcal{L}}{\frac{m}{n}(\mathcal{L} - 1) + r}\right) \\ &= O\left(\log m + \frac{\frac{n^2}{m}\mathcal{L}}{(\mathcal{L} - 1) + \frac{n}{m} r}\right). \end{aligned}$$

The lemma now follows, since $r > 0$ and $m \geq n^2$ hold.

Theorem 1 *Algorithm 1 is a linearizable implementation of an unbounded max register with amortized step complexity of $O(\log m)$ in any n-bounded-increment execution, if $m \geq n^2$. The algorithm (without the helping mechanism) is lock-free.*

Proof For any finite execution e, we define linearization points for a subset (namely, $\bigcup_{0\leq k\leq K+1} A_k \cup B$) of the operations on M invoked in e. By Lemma 5, this set includes every operation on M that completes in e. By definition, each linearization point is within the execution interval of the corresponding operation. The values returned in a sequential execution in which the linearized operations are performed in the order of their linearization point is consistent with the semantics of max registers (Lemma 6). Algorithm 1 is thus a linearizable implementation of a max register. By Lemma 7, it is lock-free, and by Lemma 8, its amortized step complexity in n-bounded executions is $O(\log m)$, provided that $m \geq n^2$.

3.3 The Helping Mechanism

We now explain the helping mechanism that makes Algorithm 1 wait-free (presented in the metal-colored lines of that algorithm). In Algorithm 1 (without metal-colored lines), a `Read` operation may not terminate because there are concurrent `Write` operations that keep making new bounded max registers obsolete. To avoid this, before making a max register obsolete (by setting the corresponding switch), a process announces the current value v of the max register to a shared helping array `H`. Entry `H`$[i]$ is used by `Write` operations by process i and stores a triplet of values. The first value is the index of the max register that i is about to make obsolete (in Line 11) immediately after its write to entry $H[i]$ (in Line 10). The second value is a sequence number that counts the number of writes done by the helping process to $H[i]$ so far. The third value is a (maximum) value of M that process i's `Write` operation was able to compute (in Line 7). Each entry `H`$[i]$ is initialized to $(-1, 0, -1)$. The value -1 in the third component of the triplet means here that process i has not yet written to `H`.

Every $\Theta(n)$ steps, a `Read` operation *op* reads the array `H` looking for a value of M that can be safely returned. Sequence numbers allow to determine such values, that is, values of M at some point in the execution interval of *op*. If no suitable value is found in `H`, the first element of each triplet is used to determine the index of a bounded max register that has not yet been made obsolete by `Write` operations. The cost of looking for help is $O(n)$ since it consists essentially in reading the n entries of the array. As looking for help is performed every $\Theta(n)$ steps, helping does not increase the amortized step complexity of `Read` operations.

A more detailed explanation follows. Helping is attempted by process i inside `Write` operations, just before i is about to make another max register obsolete. Specifically, if i just wrote to a max register $k > 0$ (Lines 5-6), it reads the maximum residue written so far to $\mathtt{max}_{k-1}$, computes the corresponding value of M based on it and stores it to a local variable *curMax* (Line 7). If $\mathtt{switch}_{k-1}$ is 0 (Line 8), then $\mathtt{max}_{k-1}$ must be made obsolete. As we prove, in this case, *curMax* was indeed a value of M at some point during the execution interval of this `Write` operation. Process i increments `hCount` in Line 9, helps by writing the triplet of values to `H`$[i]$ (Line 10), and then sets $\mathtt{switch}_{k-1}$ in Line 11.

The goal of the helping mechanism is to ensure that every `Read` operation eventually completes. Every $n+2$ iterations of the while loop of Lines 15-18, the `GetHelp` utility function is called, receiving an integer that is a multiple of $n + 2$, indicating whether or not this is its

Algorithm 2 The `GetHelp` utility function, code for process i.

Shared variables:
$\mathtt{H}_i[n]$: an array accessed only by process i, to which the H array is copied

```
21: function GetHelp(c)
22:     if c = (n + 2) then
23:         for j ∈ {0, ..., n − 1} do
24:             H_i[j] ← H[j]
25:     else
26:         for j ∈ {0, ..., n − 1} do
27:             if H[j].second − H_i[j].second ≥ 2 then return H[j].third
28:     for j ∈ {0, ..., n − 1} do
29:         last_i ← max(H[j].first, last_i)
30:     return 0
```

first invocation by the current `Read` operation (Line 14, Lines 17-18). If `GetHelp` returns a positive value then, as we prove, this value was indeed M's value at some point during the execution interval of this `Read` operation, so it returns this value in Line 18. Otherwise, the search for a non-obsolete max register is resumed. The number of iterations $n+2$ to be performed before looking for help is chosen so that (as we prove) the worst case complexity of `Read` operations is $\Theta(n)$.

The pseudo-code of `GetHelp` is presented by Algorithm 2, described next. In its first invocation by a `Read` operation *op* (performed by some process i), initialization is done by copying the `H` array to an array $\mathtt{H}_i$ which is used only by process i (Lines 22-24). In the first invocation, 0 is returned (Line 30), indicating that a maximum value is not yet available. Before returning 0, the loop of Lines 28-29 is performed. In each iteration of this loop, process i increases $\mathtt{last}_i$ (in Line 29) if the first triplet-value it reads is larger than its current value of $\mathtt{last}_i$, establishing that the current value is of an obsolete max register. This is required for bounding the worst-case step complexity of the algorithm.

In each subsequent invocation of `GetHelp` (Lines 25-27), if any, i checks, for each j, if j updated the second triplet-value of $\mathtt{H}[j]$ at least twice since *op* was invoked. If this is the case then, as we prove, the last maximum value written by j was indeed M's value at some point during *op*'s execution interval, so `GetHelp` returns it in Line 27 and then *op* returns this value in Line 18 of Algorithm 1. If the condition of Line 27 is not satisfied in any iteration, *op* updates $\mathtt{last}_i$ (if required) in the loop of Lines 28-29 and returns 0 in Line 30, signifying that i was not helped.

3.4 Linearizability and Complexity of the Full Algorithm

In this section we prove that the algorithm with the helping mechanism (henceforth *the full algorithm*) is linearizable. Let e be a finite execution. We partition `Read` and `Write` operations of e as we did in Section 3.1, except that now a `Read` operation may complete without accessing a bounded max register. Indeed, a `Read` operation may return in Line 18 of Algorithm 1 after being helped. Let us denote the set of such operations by $\mathcal{H}$.

Let op_r be a `Read` operation in $\mathcal{H}$ by process i that returns value u and let $k' = \lfloor \frac{u}{m} \rfloor$, then there is a `Write` operation by a process $j \neq i$, concurrent with op_r, that wrote u to $\mathtt{H}[j]$ (in Line 10 of Algorithm 1) after performing a `ReadMax` operation on $max_{k'}$ (in Line 7 of Algorithm 1) and op_r returns value u after reading it from $\mathtt{H}[j]$ (in Line 27 of `GetHelp`). We say that op_r *is associated with* that `ReadMax` operation.

Sets B and A_k, $k \geq 0$ are defined as in Section 3.1, except that for each $k \geq 0$, set A_k contains in addition the operations in $\mathcal{H}$ whose associated `ReadMax` is performed on $\mathtt{max}_k$.

Linearization points are defined in the same way as in Section 3.1. Let μ be a linearization of the operations performed in e on the bounded max registers $\mathtt{max}_k, k \geq 0$. Each `Write` operation $op \in A_k$ performs a `WriteMax` on the bounded max register $\mathtt{max}_k$ at Line 5. We say that *op* is associated with this `WriteMax`. Similarly, each `Read` operation $op \in (A_k \setminus \mathcal{H})$ performs a `ReadMax` on $\mathtt{max}_k$ at Line 19. We say that *op* is associated with this `ReadMax`. Each operation $op \in A_k$ is thus associated with an operation performed on $\mathtt{max}_k$. The linearization point λ_{op} of each operation $op \in A_k$ is chosen in the interval $I_k \cap I_{op}$. Furthermore, as for the lock-free algorithm, for any two operations $op, op' \in A_k$, if in μ the operation *op* is associated with is linearized before the one op' is associated with, we choose λ_{op} and $\lambda_{op'}$ such that λ_{op} precedes $\lambda_{op'}$.

It is easily verified that Lemma 1, Lemma 2, and Observation 1 hold also for the full algorithm. Recall that K is the largest index of the switches that are set in e. The following lemmas extend Lemma 3 and Lemma 4 to take into account the operations in $\mathcal{H}$.

Lemma 9 *For every $k \geq K+1$, $\mathcal{H} \cap A_k = \emptyset$.*

Proof Let op_r be a `Read` operation in $\mathcal{H}$ and let v be the value it returned. Let $k = \lfloor \frac{v}{m} \rfloor$. op_r is thus associated with a `ReadMax` operation op_{rm} performed on the bounded max register $\mathtt{max}_k$. This latter operation is executed by some process j while it is performing a `Write` operation on M with some input w. From the code

(Line 7), we have $\lfloor \frac{w}{m} \rfloor = k+1$. By Lemma 1, this `Write` operation is preceded by another `Write` operation op'_w on M whose input w' satisfies $\lfloor \frac{w'}{m} \rfloor = k+1-1 = k$. Moreover, when op'_w terminates, we have $\texttt{switch}_{k-1} = 1$ (observation 1). Hence $k-1 \leq K$ and the lemma follows.

Lemma 10 *Let $op \in \mathcal{H}$ and let k be the index of the bounded max register $\texttt{max}_k$ on which the* `ReadMax` *operation it is associated with is performed. $I_k \cap I_{op} \neq \emptyset$.*

Proof Let op_r be a `Read` operation on M that returns a value v on Line 18 after having received help. Let i be the process that performs op. i has thus read value v from some entry $H[j]$ of the shared array H. v is computed from the value u obtained by process j after performing a `ReadMax` on a bounded max register $\texttt{max}_k$ (Line 7). More precisely, we have $v = u + k \cdot m$, and the `ReadMax` by j that returns u is the operation op_r is associated with. We denote by op_{rm} this `ReadMax` operation. By the condition on Line 27, j has updated $H[j]$ at least once since the beginning of op_r and before writing v to $H[j]$. By the code, op_{rm} thus takes place between j's previous update of $H[j]$ and before it writes v to $H[j]$. Hence, op_{rm} is performed in its entirety within the execution interval of op_r.

Since before writing v to $H[j]$ and after performing op_{rm}, j checks if $\texttt{switch}_k$ is still not set (Lines 8-10), op_{rm} completes before s_k. If $k = 0$, op_{rm} takes place in I_0 and hence $I_{op_r} \cap I_0 \neq \emptyset$, as required. Let us assume that $k > 0$. op_{rm} is executed while j is performing a `Write` op_w on M with some input w, with $\lfloor \frac{w}{m} \rfloor = k+1$. By Lemma 1, this `Write` is preceded by another `Write` on M with input w' such that $\lfloor \frac{w'}{m} \rfloor = k > 0$. Hence, by Observation 1, s_{k-1} occurs before op_w starts. Therefore, op_{rm} is performed between s_{k-1} and before s_k. It thus follows that $I_{op_r} \cap I_k \neq \emptyset$.

In the full algorithm, operations terminate either after completing Line 12 in the case of a `Write` operation, Line 20 in the case of a `Read` operation that does not receive help, or Line 18 in the case of a `Read` operation that receives help. For the first two cases, Lemma 5 shows that these operations are contained in the set $\bigcup_{0 \leq k \leq K+1} A_k \cup B$. By definition, $\mathcal{H}$ is the set of the `Read` operations that terminate after having received help. Each operation op in $\mathcal{H}$ is also contained in some set A_k (k is the index of the bounded max register on which the `MaxRead` op is associated with is performed). By Lemma 9, for every $k \geq K+1$, no operation in $\mathcal{H}$ is contained in A_k. Therefore, Lemma 5 holds also for the full algorithm and after operations in $\mathcal{H}$ have been included in the sets $A_k, k \geq 0$.

We now extend Lemma 6 to `Read` operations that complete after having been helped.

Lemma 11 *Let op be* `Read` *operation contained in $\mathcal{H}$, and let v be the value it returns.*

- *If $v = 0$, there is no* `Write` *operation with an input $\neq 0$ that is linearized before op.*
- *If $v \neq 0$, the largest input value of the* `Write` *operations linearized before op is v.*

Proof Let op_{rm} be the `ReadMax` operation op is associated with. op_{rm} is performed on the bounded max register $\texttt{max}_k$, where $k = \lfloor \frac{v}{m} \rfloor$ and returns $u = v \mod m$. The proof is essentially the same as in Lemma 6.

If $v = 0$, $k = u = 0$. Hence, any `Write` operation op_w linearized before op is contained in A_0, and the input of the `WriteMax` performed on $\texttt{max}_0$ by op_w must be 0. It thus follows that the input of op_w is $0 \cdot m + 0 = 0$.

Otherwise, $v \neq 0$. We show (1) that there is a `Write` operation with input v that is linearized before op, and (2) that any `Write` operation with input $v' > v$ (if any) is linearized after op.

1. We consider two cases depending on the value of $u = v \mod m$.
   - $u = 0$. As $v \neq 0$, $k > 0$. By the code, op_{rm} is performed by a `Write` operation op' whose input v' is such that $\lfloor \frac{v'}{m} \rfloor = k+1$. It follows from Lemma 1 that op' is preceded by a `Write` operation op'' whose input v'' satisfies $\lfloor \frac{v''}{m} \rfloor = k > 0$. By Observation 1, $\texttt{switch}_{k-1}$ has been set before op'' terminates, and thus also before op_{rm} is performed. Let op''' be the `Write` operation that sets $\texttt{switch}_{k-1}$ to 1 and let v''' its input value. Note that $\lfloor \frac{v'''}{m} \rfloor = k$, and by the code, op''' writes $u''' = v''' \mod m$ to $\texttt{max}_k$ (Line 5) before setting $\texttt{switch}_{k-1}$ to 1 (on Line 11). op''' is thus contained in A_k and as the `WriteMax` it performs on $\texttt{max}_k$ precedes op_{rm}, it is linearized before op. Moreover, as op_{rm} returns 0, $u''' = 0$. Hence, the input of op''' is $v''' = k \cdot m = v$, as required.
   - $u \neq 0$. Since op_{rm} returns u, and the initial value of $\texttt{max}_k$ is 0, there is a `WriteMax` operation op_{wm} on $\texttt{max}_k$ with input u that is linearized before op_{rm}. By the code op_{wm} is performed within a `Write` operation op' (on Line 5) whose input is $v' = k \cdot m + u = v$. op' is thus contained in A_k and as op_{wm} is linearized before op_{rm}, op' is linearized before op.
2. Let op' be a `Write` operation with input $v' > v$, and let $k' = \lfloor \frac{v'}{m} \rfloor$. Note that $k' \geq k$. If op' is contained in B, it is linearized after $\texttt{switch}_{k'}$ is set. This occurs after I_k (Lemma 2) which is the interval that contains the linearization point of op. Otherwise $op' \in A_{k'}$. If $k' > k$, op' is linearized after op. If $k = k'$, op' performs a `WriteMax` on $\texttt{max}_k$

with input $u' = v' \mod m$. As $v' > v$, $u' > u$. As op_{rm} is performed on the same bounded max register $\mathtt{max}_k$ and returns u, op_{rm} is linearized before that WriteMax. Therefore op' is linearized after op.

Claim 1 *If an infinite and monotonically increasing sequence of values is written to M, then some process performs Line 10 of Algorithm 1 infinitely often.*

Proof If an infinite and monotonically-increasing sequence of values is written to M, then max registers are made obsolete infinitely often. Since a max register is only made obsolete in Line 11 of Algorithm 1, it is immediate from the code that Line 10 of that algorithm is performed infinitely often as well. Since the number of processes is finite, it follows that some process performs that line infinitely often.

Lemma 12 *The full Algorithm 1 is wait-free.*

Proof As proven in Lemma 7, the algorithm is lock-free and Write operations are wait-free. It remains to show that Read operations are wait-free as well. Assume for contradiction that there is an infinite execution e in which a Read operation op takes infinitely many steps. If there is no infinite monotonically-increasing sequence of values that is written to M then, starting from some point in e, M's value does not increase. The set of obsolete max object stops growing, hence op eventually reaches a non-obsolete max register and completes.

Otherwise, there is such a sequence of monotonically increasing values. From Claim 1, there is some process j that performs Line 10 of Algorithm 1 infinitely often. Thus, op eventually evaluates the condition on Line 27 of Algorithm 2 as true and is therefore able to terminate.

Claim 2 *Let e be an execution in which max register $\mathtt{max}_k$ becomes obsolete. Then, after s_k, the array* H *contains a triplet whose first value is at least k.*

Proof Immediate from Lemma 2 and from the fact that each process i writes to H[i] (in Line 10) a triplet of values whose first component is k just before writing 1 to $\mathtt{switch}_{k-1}$ (in Line 11).

Claim 3 *Let e be an execution, and let s and s' be two steps in e between which at least j switches are made obsolete. Then at least $j-1$ different writes to the* H *array (in Line 10) are performed between the steps s and s'.*

Proof From Lines 10-11, each Write operation op by a process i writes to H[i] (in Line 10) immediately before writing 1 to $\mathtt{switch}_{k-1}$ (in Line 11). Thus, between two consecutive executions of Line 11 by any process i, i writes to H[i]. Let k be the smallest index of a max register that has not been made obsolete in step s or before. From Lemma 2, the steps $s_k, \ldots, s_{k+j-1}$ in which the max registers $k, \ldots, k+j-1$ are made obsolete occur in this order in e, and before s'. Moreover, any process i that is about to perform Line 11 after s is about to write to $\mathtt{switch}_k$ or to a switch with a smaller index. It follows that at least a single distinct write to an entry H is done after $s_{j'}$ and before $s_{j'+1}$, for $k \le j' < k+j-1$. Hence there are at least $j-1$ such writes.

Theorem 2 *If $m = n^2$, then the full algorithm is a wait-free linearizable n-process implementation of an unbounded max register with amortized step complexity of $O(\log m)$ in any n-bounded-increment execution. In any such execution, the worst-case step complexity of* Write *operation is $O(\log n)$ and that of* Read *is $O(n)$.*

Proof For any execution e of the full algorithm, we defined linearization points for a subset of the operations that are invoked on M in e. This subset includes every operation on M that completes in e (Lemma 5, which holds for the full algorithm). Lemma 4 and Lemma 10 ensure that each linearization point is within the execution interval of the corresponding operation. The order of the operations induced by their linearization is consistent with the semantic of max registers (Lemma 6 and Lemma 11). The full algorithm is thus linearizable, and wait-free by Lemma 12.

It remains to argue regarding its complexity. In Algorithm 2, every iteration of the for loop at either Line 23 or Line 26 incurs a constant number of steps. Thus, every invocation of GetHelp incurs $O(n)$ steps. In Algorithm 1, a Write operation performs at most one WriteMax and at most one ReadMax operation, incurring a total of $O(\log m)$ steps. We note that any Read operation invokes GetHelp once every $\Theta(n)$ steps when the condition of Line 17 of Algorithm 1 is satisfied. Thus, at any point in the course of the execution, the number of steps taken by a Read operation op inside GetHelp is $O(loop_{op})$ (recall that $loop_{op}$ is the number of steps performed in the while loop of Lines 15-16). Consequently, as in the proof of Lemma 8, we get:

$$AmtSteps(e) = O\left(\frac{\sum_{op \in OP_W(e)} \log m + \sum_{op \in OP_R(e)} \log m + loop_{op}}{w+r}\right) = O(\log m).$$

The worst-case step complexity of a Write operation is logarithmic in m, since it applies at most a single WriteMax operation (in Line 5) and at most a single ReadMax operation (in Line 7) plus a constant number of additional steps. As we choose $m = n^2$, it is also logarithmic in n. It remains to prove that the worst-case complexity of Read operations is linear.

Let *op* be a Read operation by process i. We establish that GetHelp is invoked at most twice by *op*. We do so by proving that after *op* invokes GetHelp twice, it completes by returning in Line 18.

In the first invocation of GetHelp, let s be the last step performed by i in the first loop (Lines 22-24). Let α denote the smallest index of a bounded max register that has not been set before s. Let α' denote the value of $\texttt{last}_i$ when this first invocation returns. From Claim 2 and because in the first invocation of GetHelp $\texttt{last}_i$ is updated in the loop of Lines 28-29, after s, $\alpha' \geq \alpha - 1$.

After the first invocation returns, and before invoking GetHelp for the second time, i reads 1 from the $n+2$ switches $\texttt{switch}_{\alpha'}, \ldots, \texttt{switch}_{\alpha'+n+1}$. Let s' denote the first step by i in the second invocation of GetHelp. Because $\alpha' \geq \alpha - 1$, at least $n + 1$ bounded registers $\texttt{max}_{\alpha}, \ldots, \texttt{max}_{\alpha+n}$ are made obsolete between s and s'. It thus follows from Claim 3 that at least n different writes are made to array H between s and s'.

Consequently, there exists a process $\ell \neq$ i that updates $\texttt{H}[\ell]$ (in Line 10) at least twice between s and s'. It follows from the fact that process ℓ increments its second triplet-value before each such update (in Line 9) that, in iteration ℓ of the loop of Lines 26-27 of the second iteration of GetHelp, the condition of Line 27 is satisfied. Hence GetHelp returns a non-zero value and *op* completes in Line 18.

To conclude, we observe that in each invocation of GetHelp $O(n)$ steps are performed. Moreover, between two invocation of GetHelp or between its beginning and its first first invocation of GetHelp (if any), a Read operations performs $O(n)$ steps. As GetHelp is invoked at most twice by any Read operation, the worst-case complexity of Read operations is $O(n)$.

4 Wait-Free Counter with Polylogarithmic Amortized Step Complexity

Algorithm 3 An n-process counter C_j, code for process i.

Shared variables:
R: an n-process $\texttt{UnboundedMaxReg}_{n^2}$ object, initially 0
If $j > 1$: left: a $C_{\lceil j/2 \rceil}$ counter object, initially 0
right: a $C_{j-\lceil j/2 \rceil}$ counter object, initially 0

1: **function** Inc(C_j)
2: **if** $j = 1$ **then**
3: $v \leftarrow$ ReadMax(R)
4: WriteMax(R, $v + 1$)
5: **else**
6: **if** i's C_1 leaf-counter is on the left sub-tree **then** Inc(left) **else** Inc(right)
7: $v_0 \leftarrow$ read(left)
8: $v_1 \leftarrow$ read(right)
9: WriteMax(R, $v_0 + v_1$)
10: **function** Read(C_j)
11: **return** ReadMax(R)

Algorithm 3 presents a wait-free recursive construction of a linearizable counter that has polylogarithmic amortized step complexity in all executions, regardless of their length. The algorithm is essentially the same as the (non-recursive) counter construction of Aspnes et al. [AAC12], except that the latter uses the max registers of [AAC12], whose amortized step complexity is linear in sufficiently long executions, whereas ours uses our wait-free unbounded max registers.

Let C_j denote a counter, shared by n processes, implemented by Algorithm 3. For simplicity and without loss of generality, assume in the following that each of n and j is an integral power of 2. C_j's value is stored in an n-process wait-free unbounded max register R, which is of type $\texttt{UnboundedMaxReg}_{n^2}$. If $j > 1$ holds, then C_j also contains two $C_{j/2}$ child-counters – left and right. A counter C_n serves as a root of a tree of counters and all processes can invoke Inc operations on C_n. At the bottom layer of the tree, each process i is associated with a single C_1 leaf-counter on which only i can invoke Inc operations.

To read C_j, process i simply invokes a Read operation on C_j's R object and returns the response (Line 11). Incrementing a C_1 object consists of simply reading R and writing to it a value larger by one (Lines 3-4). To increment a C_j counter, for $j > 1$, process i increments either the left or the right child counter, depending on whether its C_1 leaf-counter is on the left or the right subtree of C_j, reads the values of both child counters and writes their sum to R (Lines 6-9). Observe that at most j distinct processes can invoke Inc operations on any specific C_j counter.

In the following proofs we let $\mathcal{C}$ denote a C_n object implemented by Algorithm 3 and e be an execution of $\mathcal{C}$.

Lemma 13 *The C_j counter implementation of Algorithm 3 is linearizable.*

Proof The proof is by induction on j.

Base Case. For $j = 1$, the UnboundedMaxReg object R of a C_1 counter may only be incremented by a single process. Since R's value is always increased by exactly 1, the execution is 1-bounded-increment for R, so the correctness of R follows from Theorem 2. Increment operations on C_1 are linearized when the Write operation invoked in Line 4 is linearized and read operations on C_1 are linearized when the `Read` operation invoked in Line 11 is linearized.

Induction Hypothesis. For all $k < j$, C_k is a linearizable counter and the value of its max object R is never increased by a `Write` operation by more than k.

Inductive Step.

Lemma 14 *e is a j-bounded-increment execution for $C_j.R$.*

Proof The proof is divided into two parts. We first prove the left-hand inequality of Definition 1. Let e' be a prefix of e immediately after which process p is about to invoke a `Write()` operation op_v on $C_j.R$ with input v (in Line 9). Let $\mathcal{I}$ be the set of `Inc` operations that have completed on C_j in e'. Observe that each operation $op \in \mathcal{I}$ has performed one `Inc` operation on either C_j.`left` or C_j.`right`. We partition $\mathcal{I}$ accordingly: $\mathcal{I} = \mathcal{I}_0 \cup \mathcal{I}_1$, where for any $op \in \mathcal{I}$, $op \in \mathcal{I}_0$ if op performed an `Inc` operation on C_j.`left` and $op \in \mathcal{I}_1$ if op performed an `Inc` operation on C_j.`right`.

By IH, both C_j.`left` and C_j.`right` are linearizable counters. Let $op_0 \in \mathcal{I}_0$ be the operation whose `Inc` operation on C_j.`left` is linearized last among all `Inc` operations on C_j.`left` performed by the operations in $\mathcal{I}_0$. Let c_0 be the value of C_j.`left` immediately after the `Inc` operation on that object by op_0. op_1 and c_1 are defined similarly. From Lines 7-9, for each $r \in \{0, 1\}$, after performing an `Inc` operation on either C_j.`left` or C_j.`right`, op_r performs *read* operations on both C_j.`left` and C_j.`right` before writing the sum u_r of the values read to $C_j.R$. We show that $v' = \max\{u_0, u_1\} \geq c_0 + c_1$. Indeed, assume that op_0's *read* operation on $C_j.right$ returns a value strictly smaller than c_1 (note that, otherwise, $u_0 \geq c0 + c1$). Then, op_1's `Inc` operation on C_j.`right` is linearized after op_0's *Read* operation on C_j.`right`. It thus follows that op_1's *read* operation on C_j.`left` starts after op_0's `Inc` operation on C_j.`left` has completed. We thus conclude that $u_1 \geq c_0 + c_1$.

Since both op_0 and op_1 have completed in e', a `WriteMax` operation on R of value $v' \geq c_0 + c_1$ has completed in e'. If $v \leq v'$ then $v - j \leq v'$ and the claim holds. Otherwise, again from Lines 7-9, the operand v of the `WriteMax` operation op_v is the sum of the values v_0, v_1 returned by the `Read` operations performed on the counters C_j.`left` and C_j.`right`, respectively. $v_0 = c_0 + \delta$, for $\delta > 0$, implies that there are δ `Inc` operations on C_j.`left` that have been linearized after the `Inc` operation on the same counter by op_0. From the definition of op_0, these δ operations take place within δ `Inc` operations on C_j that did not complete in e'. The same argument applies for v_1. Since there are at most j processes that may invoke `Inc` operations on C_j and thus at most j incomplete `Inc` operations on C_j after e', it follows that $v = v_0 + v_1 \leq j + c_0 + c_1$. Hence, there is a value $v' = \max\{u_0, u_1\}$ such that $v - v' \leq j$ and a `WriteMax`(v') on R has completed before the operation $op_v =$ `WriteMax`(v) on R starts.

We next prove both inequalities of Definition 1. Let op be a `WriteMax` operation on $C_j.R$ with input $v > j$. The first part above established that there exists a `WriteMax` operation op' on $C_j.R$ with input v' that finishes before op starts, such that $v - j \leq v'$. Assume that $v' \geq v$. Let $\mathcal{O}_>$ be the set of `WriteMax` operations on $C_j.R$ that (1) precede op and (2) whose input is larger than or equal to v. We define a partial order $\prec$ on the operations in $\mathcal{O}_>$ as follows:

$$\forall W, W' \in \mathcal{O}_>, W \prec W' \iff W \text{ precedes } W' \text{ in } e.$$

Let us observe that $\mathcal{O}_>$ is non-empty and finite. The latter is because e is finite and so only finitely many operations precede op in e and the former follows from the existence of op'. Consider any minimal element in the partially ordered set $\mathcal{O}_>$, that is any operation W such that for any operation $W' \in \mathcal{O}_>$, W' does not precede W. Since $\mathcal{O}_>$ is finite, there is at least one such operation W. Let in_W denote its input. Since $W \in \mathcal{O}_>$, we have $in_W \geq v$. Also, by applying the left-hand inequality (proved in the first part of the proof) to W, there exists an operation W' with input $in_{W'}$ that precedes W such that $in_{W'} \geq in_W - j \geq v - j$. As $W' \prec W$, and W is chosen as a minimal element of $\mathcal{O}_>$, it follows that $W' \notin \mathcal{O}_>$. Since W' precedes both W and op, we get that $in_{W'} < v$, which concludes the proof.

From Lemma 14 and Theorem 2 we conclude that $C_n.R$ is linearizable in e. Based on this, the proof proceeds similarly to the proof of [AAC12, Lemma 4].

From IH, the counters C_j.`left` and C_j.`right` are linearizable. We associate with every increment operation op on C_j a value as follows. Let c_0 and c_1 respectively denote the values of C_j.`left` and C_j.`right` immediately after p's increment of C_j's child (corresponding to p's identifer), in Line 6, is linearized. Then we associate with op the value $v = c_0 + c_1$. We linearize an *Inc* operation op, associated with value v, when a value $v' \geq v$ is first written to $C_j.R$ in Line 9 (either by p or by another process). We linearize a *Read* operation on C_j when it reads $C_j.R$ in Line 11.

We now prove that each linearization point is within its operation execution interval. Consider an `Inc` operation op associated with value v. A value $v' \geq v$ cannot be written to $C_j.R$ before op starts, because, from the linearizability of C_j.`left` and C_j.`right`, before *op* starts, the sum of these two counters is less than $c_0 + c_1$. Since *op* itself writes value v to $C_j.R$ before it terminates, the linearization point occurs before *op* terminates. The fact that the linearization point of a *Read* operation on C_j lies within its execution interval follows immediately from the linearizability of $C_j.R$, established by Lemma 14. Finally, the linearization points result in a valid sequential execution, because every *Read* operation on C_j that returns value v is preceded by exactly v `Inc` operations on C_j.

Lemma 15 *Algorithm 3 has $O(\log^2 n)$ amortized step complexity.*

Proof From Algorithm 3 and the fact that $\mathcal{C}$ is shared by n processes, every operation on $\mathcal{C}$ applies a constant number of `ReadMax`/`WriteMax` operations to each of $O(\log n)$ different `UnboundedMaxReg` objects, as the recursive calls in Lines 7-9 and 11 unfold. Letting $ncops(e)$ denote the number of operations that are invoked on $\mathcal{C}$ in e, the total number of `ReadMax`/`WriteMax` operations on all the implementation's `UnboundedMaxReg` objects is therefore $O\big(\log n \cdot ncops(e)\big)$. From Theorem 2, letting $m = n^2$, it follows that the total number of steps performed in e is $O\big(\log^2 n \cdot ncops(e)\big)$.

Lemma 16 *for both `Inc` and `Read` operations, Algorithm 3 has $O(n)$ worst-case step complexity.*

Proof A `Read` operation on the counter invokes a single `Read` on an $\texttt{UnboundedMaxReg}_{n^2}$ object hence, from Theorem 2, its step complexity is $O(n)$. It remains to argue about the worst-case step complexity of `Inc` operations.

Algorithm 3 uses at its root (layer 0 of the counters tree) an unbounded max register for n processes. More generally, each tree layer $i \in \{0, \ldots, \log n\}$, consists of unbounded max registers for $\frac{n}{2^i}$ processes. Unfolding the recursion of Algorithm 3, an `Inc` operation I on the root results in the following operations.

- At each of the tree layers, a single `Write` operation is applied to a single `UnboundedMaxReg` object. From Theorem 2, the total worst-case step complexity of all these `Write` operations is $O(\log^2 n)$.
- At most two `Read` operations are applied to a single `UnboundedMaxReg` object at each of the tree layers. From Theorem 2, there is a constant c such that the number of steps taken by a `Read` operation on an `UnboundedMaxReg` object for j processes is at most $c \cdot j$. Thus the total number of steps incurred by all the `Read` operations triggered by I is at most $\sum_{j\in\{0,\ldots,\log n\}} 2c \cdot 2^j = O(n)$.

Theorem 3 *Algorithm 3 is a wait-free linearizable n-process implementation of an unbounded counter with amortized step complexity of $O(\log^2 n)$ and worst-case step complexity of $O(n)$.*

Proof From Lemma 13, the algorithm is linearizable. By Lemma 12, all the `UnboundedMaxReg` objects used by Algorithm 3 are wait-free. Therefore, clearly from the pseudo-code, Algorithm 3 is wait-free as well. The claimed amortized and worst-case complexities follow from Lemma 15 and Lemma 16 respectively.

A logarithmic lower bound on the amortized step complexity of implementing an obstruction-free one-time `fetch&increment` object from read, write and k-word compare-and-swap operations was proved by Attiya and Hendler in [AH10, Theorem 9]. Their proof can be easily adapted to obtain the following result:

Lemma 17 *Any n-process obstruction-free implementation from read/write registers of a counter object has an execution that contains $\Omega(n \log n)$ steps, in which every process performs a single `Inc` operation followed by a single `Read` operation.*

Lemma 17 shows that every lock-free read/write counter implementation has an execution whose amortized step complexity is at least logarithmic in the number of processes, showing that our counter algorithm is optimal in terms of amortized step complexity up to a logarithmic factor.

5 Discussion

In this work, we presented the first lock-free read/write counter algorithm that provides sub-linear amortized step complexity in all executions, regardless of their length. The amortized step complexity of our algorithm is $O(\log^2 n)$, where n is the number of processes sharing the implementation. This is optimal up to a logarithmic factor, since there exists a logarithmic lower bound on the amortized step complexity of n-process one-time counters. In contrast, the amortized step complexity of the counter algorithm of [AAC12] deteriorates as the number of `Inc` operations increases and eventually becomes linear in n.

It is unclear whether there exists a wait-free (or even lock-free or obstruction-free) read/write counter implementation with $o(\log^2 n)$ amortized step complexity. Interestingly, a similar gap between an $O(\log^2 n)$ upper bound and an $\Omega(\log n)$ lower bound exists for the *worst-case* step complexity of counters [AAC12].

The space complexity of our counter is infinite, since it uses our unbounded max registers, and each of these encapsulates an infinite number of bounded max registers. Finding a bounded-space read/write counter with sub-linear amortized step complexity is another open question. These questions are left for future work.

Acknowledgements We thank the anonymous reviewers of preliminaries versions of this work for their many helpful comments.

References

[AAC12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012.

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[AC10] James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. *ACM Trans. Algorithms*, 6(2):25:1–25:23, 2010.

[AC13] James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In *27th International Symposium on Distributed Computing (DISC)*, volume 8205 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2013.

[AF01] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[AH90] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.

[AH10] Hagit Attiya and Danny Hendler. Time and space lower bounds for implementations using k-CAS. *IEEE Trans. Parallel Distrib. Syst.*, 21(2):162–173, 2010.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[AKK+14] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing*, 27(6):393–417, 2014.

[And93] James Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of American Federation of Information Processing Societies (AFIPS)*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BG11] Michael A. Bender and Seth Gilbert. Mutual exclusion with o(log^2 log n) amortized work. In *52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 728–737. IEEE Computer Society, 2011.

[DS00] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. *J. Parallel Distributed Comput.*, 60(7):853–890, 2000.

[EW13] Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In *Proceedings of the 27th International Symposium on Distributed Computing, (DISC)*, volume 8205 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2013.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529. IEEE Computer Society, 2003.

[HSW96] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Comput.*, 9(4):193–203, 1996.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[IC94] Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In Gerard Tel and Paul M. B. Vitányi, editors, *8th International Workshop on Distributed Algorithms (WDAG)*, volume 857 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1994.

[Jay98] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 201–210, 1998.

[JTT00] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2), 2000.

[KW17] Pankaj Khanchandani and Roger Wattenhofer. Brief announcement: Fast shared counting using $O(n)$ compare-and-swap registers. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 95–96. ACM, 2017.

[MT97] Shlomo Moran and Gadi Taubenfeld. A lower bound on wait-free counting. *J. Algorithms*, 24(1):1–19, 1997.

[MTY96] Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent counting. *J. Comput. Syst. Sci.*, 53(1):61–78, 1996.