Long-Lived Counters with Polylogarithmic Amortized Step Complexity

Mirza Ahad Baig
LaBRI, Bordeaux INP, France
CNRS, ReLaX, UMI2000, Siruseri, India
Chennai Mathematical Institute, Siruseri, India
mirzabaig.cmi@gmail.com

Danny Hendler
Ben-Gurion University of the Negev, Beer-Sheva, Israel
hendlerd@cs.bgu.ac.il

Alessia Milani
LaBRI, Bordeaux INP, France
milani@labri.fr

Corentin Travers
LaBRI, Bordeaux INP, France
travers@labri.fr

Abstract

A shared-memory counter is a well-studied and widely-used concurrent object. It supports two operations: An `Inc` operation that increases its value by 1 and a `Read` operation that returns its current value. Jayanti, Tan and Toueg [16] proved a linear lower bound on the *worst-case* step complexity of obstruction-free implementations, from read and write operations, 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 paper, we address this gap. We present the first wait-free n-process counter, implemented using only read and write operations, whose amortized operation step complexity is $O(\log^2 n)$ in all executions. This is the first non-blocking read/write counter algorithm 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 optimal up to a logarithmic factor.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Concurrent algorithms

Keywords and phrases Shared Memory, Wait-freedom, Counter, Amortized Complexity, Concurrent Objects

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.3

Funding Mirza Ahad Baig, Alessia Milani and Corentin Travers are supported by ANR projects Descartes and FREDDA. Mirza Ahad Baig is additionally supported by UMI Relax. Danny Hendler is supported by the Israel Science Foundation (grant 380/18).

Acknowledgements We thank the anonymous reviewers for their many helpful comments.

1 Introduction

A shared-memory *counter* [18] is a well-studied and widely-used concurrent object [2, 5, 7, 10, 17]. A counter 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 an *atomic snapshot* [1, 3, 7] object, allowing 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 snapshot can be implemented, using reads and writes only, in step complexity linear in the number of processes n [8, 14], so can counters.

Indeed, a well-known result by Jayanti, Tan and Toueg [16] proved a linear lower bound on the worst-case step complexity of obstruction-free read/write implementations of a large class of shared objects that includes counters. Aspnes, Attiya and Censor-Hillel [4] observed that the lower bound holds only when numerous operations are applied to the object and does not rule out the possibility of obtaining 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. Specifically, they presented a wait-free counter for which the step complexities of `Inc` and `Read` operations are $O\big(min(\log n \log v, n)\big)$ and $O\big(min(\log v, n)\big)$, respectively, where v is the object's current value. However, the worst-case and amortized step complexities of the counter algorithm of [4] 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 [16] 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 presenting the first wait-free read/write counter whose amortized step complexity is polylogarithmic. This is the first non-blocking read/write counter that provides sub-linear amortized step complexity in *executions of arbitrary length*. Our counter implementation is based on the counter algorithm presented in [4]. 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 [4], thus obtaining a counter with $O(\log^2 n)$ amortized step complexity. 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 objects's current value. Thus, executions of arbitrary length can have linear amortized complexity. Aspnes and Censor-Hiller [6] 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 unbounded max algorithm makes a similar assumption. The max register algorithm of [6] is randomized since it relies on a randomized helping mechanism, whereas ours is deterministic.

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 [15]. Attiya and Hendler [9] presented lower bounds on the time and space complexities of obstruction-free implementations of several objects from k-word compare-and-swap operations. Specifically, using an information-theoretic argument as well, they proved a logarithmic lower bound on the *amortized* step complexity of implementing an obstruction-free one-time `fetch&increment` object [9, Theorem 9]. Their proof can be modified in a straightforward manner to establish the same result for counters, implying that our algorithm is optimal in terms of amortized step complexity up to a logarithmic factor.

The rest of this paper is organized as follows. We present the system model we assume and additional required definitions in Section 2. In Section 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 [4] (instead of using the max register algorithm of [4]) we obtain a linearizable counter with polylogarithmic amortized step complexity. The paper is concluded with a short discussion in Section 5.

2 Model and Preliminaries

Read/write shared memory. We consider a standard shared-memory model, where a set $\mathcal{P}$ of n crash-prone asynchronous processes communicate via shared *registers*, supporting only atomic read and write operations. A concurrent object *implementation* specifies the object's state representation and the algorithms processes follow when they perform operations supported by the object. An *execution* is a series of *steps* performed by processes as they follow their algorithms, in each of which a process applies at most a single read or write operation to a register (possibly in addition to some local computation). In what follows, we only consider finite executions. Roughly speaking, an implementation is *linearizable* [13] if each operation appears to take effect atomically at some point between its invocation and response; it is *wait-free* [11] if each process completes its operation if it performs a sufficiently large number of steps; it is *lock-free* if at least one process completes its operation after a sufficiently large number of steps is performed; it is *obstruction-free* [12] if each process completes its operation if it performs a sufficiently large number of steps when running solo. Operation Op_1 *precedes* operation Op_2 in an execution E, if Op_1's response appears in E before Op_2's invocation.

Complexity measure. The worst-case *amortized step complexity* (henceforth simply amortized step complexity) is defined as the worst-case (taken over all possible 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 infinite. More precisely, given a finite execution E, an operation Op *appears* in E if it is invoked in E. We denote by $Nsteps(Op, E)$ the number of steps performed by Op in E and by $Ops(E)$ the set of operations that appear in E. The amortized step complexity of an implementation A is then:

$$AmtSteps(A) = \max_{E\text{: finite execution of } A} \frac{\sum_{Op \in Ops(E)} Nsteps(Op, E)}{|Ops(E)|}.$$

Max registers. A *max register* r supports a `WriteMax`(r,v) operation that writes a non-negative integer $v \geq 0$ to r and a `ReadMax`(r) operation that returns the maximum value previously written to r. A *bounded* max register $\texttt{MaxReg}_m$ can assume values from $\{0, \ldots, m-1\}$, for some integer m. 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 $[m \cdot j, m \cdot (j+1) - 1]$. Henceforth, 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. All max registers and their corresponding switches are initialized to 0. Each process i has a variable $\texttt{last}_i$, storing the largest index j such that i accessed $\texttt{max}_j$, initialized to 0 as well.

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). 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 higher-indexed max registers 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.

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
$\texttt{last}_i \in \mathbb{N}_0$: stores the largest index j such that process i accessed $\texttt{max}_j$, initially 0
$\texttt{H}[n][n]$ initially all 0 : a 2D integer array, $\texttt{H}[i][j]$ used by process j to help process i
$\texttt{nextToHelp}_i$: identifer of last process helped by i

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

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). Once 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 (non-obsolete) max register and returns the sum (line 20).

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 E be an execution and let M be an `UnboundedMaxReg` object. We say that E is an ℓ-bounded-increment execution for M if for each write operation $op = \texttt{Write}(v)$ on M in E, with $v > \ell$, there exists a write operation $op' = \texttt{Write}(v')$ on M in E that precedes op, such that $v - \ell \leq v' < v$.*

In Section 4, we present an n-process unbounded counter implementation that uses `UnboundedMaxReg` objects. As we prove, all the executions of that counter are n-bounded-increment executions for all these objects. Let M be an $\texttt{UnboundedMaxReg}_m$ object, implemented by Algorithm 1, for $m \geq n$, and let E be an n-bounded-increment execution for M, we now show that M is linearizable in E. We classify every write operation W on M that appears in E to exactly one of the 4 following types.

(i) W did not yet execute line 4 in E.
(ii) W executed line 4 and read $switch_k = 0$, but its `WriteMax` in line 5 was not yet linearized.
(iii) W executed line 4, read $switch_k = 0$ and its `WriteMax` operation in line 5 was linearized. We say that W *is associated with* that `WriteMax` operation.
(iv) W executed line 4 and read $switch_k = 1$.

Similarly, we classify every read operation R on M to the following 2 types:

(i) R did not yet perform in E a `ReadMax` operation in line 19 that was linearized.
(ii) R read $switch_k = 0$, for some k, and its `ReadMax` in line 19 was linearized. We say that R *is associated with* that `ReadMax` operation.

We associate with each $k \in \mathbb{N}_0$ two sets of operations on M in E, denoted $Down_k$ and $Futile_k$. Operations on M are partitioned into these sets as follows:

- $Down_k$ contains `Write` operations of type (iii) and `Read` operations of type (ii) that are associated with `WriteMax`/`ReadMax` operations on $\texttt{max}_k$.
- $Futile_k$ contains `Write` operations of type (iv).

Operations that were not assigned to any *Down* or *Futile* set are `Write` operations of types (i) and (ii) and `Read` operations of type (i). All these operations did not complete in E and will not appear in its linearization. We refer to these as *removed operations*. The rest of the operations are linearized according to the following *ordering rules*.

1. For all pairs k, k' such that $k < k'$, all the operations in $Down_k$ are ordered before all the operations in $Down_{k'}$.
2. We order the operations within each set $Down_k$ according to the linearization order of the `WriteMax` and `ReadMax` operations on the $\texttt{max}_k$ register with which they are associated.

3. Rules 1-2 order all *Down* operations. Enumerate them as Dop_1, $Dop_2, \ldots, Dop_r$. For any futile operation $Fop \in \bigcup_k Futile_k$, define the set

$$\mathcal{S}_{Fop} = \{Dop \in \cup_{k \in \mathbb{N}_0} Down_k \mid Fop \text{ precedes } Dop \text{ in } E\}.$$

If S_{Fop} is empty, we put *Fop* after Dop_r. Otherwise, let Dop_i be the least operation in S_{Fop} according to the ordering on *Down* operations, we put *Fop* immediately before Dop_i. For each set of the *Futile* operations put either immediately before some Dop_i or after Dop_r, the set of *Futile* operations is ordered according to their real-time order in E.[1]

Rules 1–3 define a full ordering among all non-removed operations.

▶ **Observation 2.** *An operation Op on M in E is associated with a `ReadMax` or `WriteMax` operation on $\mathtt{max}_k$ if and only if $Op \in Down_k$.*

▶ **Observation 3.** *The sets $Down_k$ and* Futile_k *(for all values of k) are mutually exclusive and contain all the operations on M that appeared in E except for removed operations.*

▷ Claim 4. M's $switch_j$ switches are set to 1 in E in increasing order, starting from $switch_0$.

Proof. Follows since M is an $\mathtt{UnboundedMaxReg}_m$ object, for $m \geq n$, E is an n-bounded-increment execution for M, and from Lines 8,11. ◀

▷ Claim 5. For all $k' \leq k$, there are no two operations *Fop*, *Dop* such that $Fop \in Futile_k$, $Dop \in Down_{k'}$ and *Fop* is linearized before *Dop* in the ordering given by rules 1-3.

Proof. Suppose towards a contradiction that *Fop* is linearized before *Dop*. If *Fop* was placed after Dop_r when applying rule 3, we immediately reach a contradiction. Assume otherwise, then, from rule 3, there exists a *Down* operation Dop_i, such that *Fop* precedes Dop_i, *Fop* is linearized before Dop_i, and no *Down* operation is linearized between *Fop* and Dop_i. Consequently, it must be that *Dop* is linearized after Dop_i. From rules 1-2, we have that $Dop_i \in Down_{k_1}$ such that $k_1 \leq k' \leq k$. Since $Dop_i \in Down_{k_1}$, Dop_i reads 0 from $Switch_{k_1}$. However, *Fop* reads $Switch_k = 1$ before Dop_i starts, hence, by Claim 4, $Switch_{k_1} = 1$ when Dop_i starts. This is a contradiction. ◀

▶ **Lemma 6.** *Ordering rules 1-3 define a sequential order between E's non-removed operations that preserves the real-time order between non-overlapping operations in E.*

Proof. From ordering Rule 2, Observation 2 and the linearizability of the max_j objects, for each j, the real-time order between all operations in $Down_k$ is preserved. From Claim 4, M's switches are set to 1 in increasing order. Consequently, for any two operations $Op \in Down_k$ and $Op' \in Down_{k'}$, such that $k < k'$, Op' does not precede Op in E. It follows that the real-time order between each pair of *Down* operations is preserved by the linearization.

It remains to argue about *Futile* operations. Let $Dop_1, Dop_2, \ldots, Dop_r$ be the linear order among all *Down* operations, as specified by rules 1-2. Let Fop_1, Fop_2 be *Futile* operations such that Fop_1 is linearized before Fop_2. There are two cases to consider. If both operations are put after Dop_r or immediately before the same operation Dop_i then, according to rule 3, their order preserves E's real-time order. Otherwise, there exists at least one *Down* operation linearized between them. Let *Dop* be the first *Down* operation ordered

[1] In general, this induces a partial order on *Futile* operations, which can be extended to a full order arbitrarily.

after Fop_1. From rule 3, Fop_1 precedes Dop in real-time order. Suppose Fop_2 precedes Fop_1 in real-time order, then $Dop \in \mathcal{S}_{Fop_2}$ holds. Since Fop_2 is linearized by rule 3 before all the $Down$ operations that follow it in real-time order, this is a contradiction.

Let Fop and Dop respectively be a $Futile_k$ and a $Down_{k'}$ operation. Suppose Fop is linearized before Dop but Dop precedes Fop in real-time order. If $k' \leq k$, then, by Claim 5, this is a contradiction. Assume, then, that $k < k'$ holds and consider the application of ordering rule 3 to Fop. If Fop is put after Dop_r, then it is linearized after Dop, which is a contradiction. Assume, then, that Fop is put by rule 3 immediately before some $Dop_i \in \mathcal{S}_{Fop}$, so Fop precedes Dop_i in real-time order. It follows that Dop precedes Dop_i in real-time order, so Dop is linearized before Dop_i. Since no $Down$ operation can be linearized between Fop and Dop_i, it follows that Dop is linearized before Fop. This is a contradiction.

Finally, suppose that Fop precedes Dop in real-time order. In this case, from rule 3, Fop is linearized before Dop, preserving real-time order. ◀

▶ **Lemma 7.** *The linearization defined by ordering rules 1-3 satisfies the sequential semantics of a max register.*

Proof. For an integer $v \in \mathbb{N}_0$, let $v' = v \mod m$ and $k = \lfloor v/m \rfloor$. Consider a `Read` operation Op_r on M in E that returns value v. Then Op_r is associated with a `ReadMax`($\mathtt{max}_k$) operation that returned v'. First, we prove that there is a `Write` operation Op_w that wrote v to M and Op_w is linearized before Op_r. From the linearizability of $\mathtt{max}_k$, there is a `WriteMax`($\mathtt{max}_k, v'$) operation that is linearized before the `ReadMax`($\mathtt{max}_k$) associated with Op_r. Thus, there is a `Write` operation $Op_w(v' + k \cdot m)$ on M and, by Observation 2, it belongs to $Down_k$, so Op_w is ordered before Op_r according to ordering rule 2. To conclude the proof, we show that there is no `Write` operation Op_1 that writes value $v_1 > v$ to M and is linearized before Op_r. Suppose towards a contradiction that Op_1 exists. The following two cases exist:

- $\lfloor v_1/m \rfloor = k$. This implies that $(v_1 \mod m) > (v \mod m)$. If $Op_1 \in Down_k$, this contradicts the linearizability of $\mathtt{max}_k$, because the `ReadMax` operation associated with Op_r does not return the maximum value written to $\mathtt{max}_k$ before it. If $Op_1 \in Futile_k$, this contradicts Claim 5.
- $\lfloor v_1/m \rfloor > k$. In this case, either $Op_1 \in Down_{k'}$ or $Op_2 \in Futile_{k'}$, for some $k' > k$. In the first case, Op_1 is linearized after Op_r by rule 1. In the second case, Claim 5 ensures that Op_1 is linearized after Op_r. ◀

▶ **Lemma 8.** *Algorithm 1 (without the helping mechanism) is lock-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). 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. In the following, we denote by $Ops(E)$ the set of all operations that appear in E and by $Ops_R(E)$ (resp. $Ops_W(E)$) the set of all `Read` operations (resp. all `Write` operations) that appear in E. For an operation Op, we let $Nsteps(Op, E)$ denote the number of steps performed by Op in E.

▶ **Lemma 9.** *If $m \geq n^2$, then the* `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:

$$AmtSteps(E) = \frac{\sum_{op \in Ops(E)} Nsteps(op, E)}{|Ops(E)|}. \tag{1}$$

Let r be the number of read operations and w be the number of write operations in $Ops(E)$. `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:

$$AmtSteps(E) = O\Big(\big(\sum_{op \in Ops_W(E)} \log m + \sum_{op \in Ops_R(E)} \log m + loop_{op}\big)/(w+r)\Big). \tag{2}$$

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

$$\sum_{op \,\in\, Ops_R(E)} loop_{op} = O\Big(r + \sum_{i \in \mathcal{P}} last_i\Big). \tag{3}$$

Consequently,

$$AmtSteps(E) = O\Big(\frac{w \cdot \log m + r \cdot \log m + (r + \sum_{i \in \mathcal{P}} last_i)}{w + r}\Big). \tag{4}$$

Assume that max register $\mathtt{max}_\alpha$ is accessed in E. Since E is an n-bounded-increment execution and all $\mathtt{max}_j$ registers are m-bounded, at least $m \cdot (\alpha - 1)/n$ $Write$ operations have completed 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 $w \geq m \cdot (\mathcal{L} - 1)/n$. Furthermore, $\sum_{i \in \mathcal{P}} last_i \leq n \cdot \mathcal{L}$. Thus,

$$\begin{aligned} AmtSteps(E) &= O\Big(\frac{w \log m + r \log m + (r + n \cdot \mathcal{L})}{w + r}\Big) = O\Big(\frac{(w+r)\log m}{w+r} + \frac{r}{w+r} + \frac{n \cdot \mathcal{L}}{w+r}\Big) \\ &= O\Big(\log m + \frac{n \cdot \mathcal{L}}{\frac{m}{n}(\mathcal{L} - 1) + r}\Big) = O\Big(\log m + \frac{\frac{n^2}{m}\mathcal{L}}{(\mathcal{L} - 1) + \frac{n}{m} r}\Big). \end{aligned} \tag{5}$$

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

From Lemmata 6–9, we obtain:

▶ **Theorem 10.** *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.*

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

Shared variables:

  $\texttt{HR}_i[n]$: an integer array, to which the i'th row in the H array is copied

  $\texttt{C}_i[n]$: an integer array, counting number of writes by each helper for process i

1: **function** $\texttt{GetHelp}(c)$
2:   **if** $c = n$ **then**
3:     **for** $j \in \{0, \ldots, n-1\}$ **do**
4:       $\texttt{HR}_i[j] \leftarrow \texttt{H}[i][j], \texttt{C}_i[j] \leftarrow 0$
5:   **else**
6:     **for** $j \in \{0, \ldots, n-1\}$ **do**
7:       **if** $\texttt{HR}_i[j] < \texttt{H}[i][j]$ **then**
8:         $\texttt{HR}_i[j] \leftarrow \texttt{H}[i][j]$, $\texttt{C}_i[j]++$
9:         **if** $\texttt{C}_i[j] = 2$ **then return** $\texttt{HR}_i[j]$
10:   **return** 0

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). It uses a 2-dimensional shared array `H`. Entry $\texttt{H}[i][j]$ is used by process j to help process i by writing to it a (maximum) value of M that process j was able to compute. Each process i owns variable $\texttt{nextToHelp}_i$, storing the index of the next process it should help. Helping is attempted by process i inside `Write` operations, whenever i is about to make another max register obsolete. Specifically, if i is about to write to a max register $k > 0$ (line 6), it reads the maximum residue written so far to $\texttt{max}_{k-1}$, computes the corresponding value of M based on it and stores it to a local variable *curMax* (line 7). If $\texttt{switch}_{k-1}$ is 0 (line 8), then $\texttt{max}_{k-1}$ 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 i'th `Write` operation, so i attempts to help process $\texttt{nextToHelp}_i$ by writing to the appropriate entry of array `H` and increments $\texttt{nextToHelp}_i$ modulo n (lines 9–10).

The goal of the helping mechanism is to ensure that every `Read` operation eventually completes. Every n iterations of the while loop of lines 15–18, the `GetHelp` utility function is called, receiving an integer that is a multiple of n, indicating whether or not this is its first invocation by the current `Read` operation (line 14, lines 17–18). If `GetHelp` returns a positive value then, as we prove, this was indeed M's value at some point during the execution interval of `Read`, so it returns this value in line 18. Otherwise, the search for a non-obsolete max register is resumed.

The pseudo-code of `GetHelp` is presented by Algorithm 2, described next. In its first invocation by `Read` operation R (performed by some process i), initialization is done by copying the i'th row of the `H` array to array $\texttt{HR}_i$ and initializing all elements of a second array $\texttt{C}_i$ to 0 (lines 2–4). Both $\texttt{HR}_i$ and $\texttt{C}_i$ are only accessed by process i. Element $\texttt{C}_i[j]$ counts the number of times in which i observed that it was helped by process j in the course of R. In the first invocation, 0 is returned (line 10), indicating that a maximum value is not yet available. In each subsequent invocation of `GetHelp` (lines 5–9), if any, i checks, for each j, if it was helped by j since the last time it read $\texttt{H}[i][j]$, in which case it updates $\texttt{HR}_i[j]$ and increments $\texttt{C}_i[j]$. If i was helped by some process j at least twice since R started then, as we prove, the maximum value computed by j for i was indeed M's value at some point during R's execution interval, so `GetHelp` returns it in line 9 and R then returns this value in line 18 of Algorithm 1. Otherwise, 0 is returned in line 10.

3.4 Correctness

In this section we prove that the algorithm with the helping mechanism (henceforth *the full algorithm*) is linearizable. We classify read and write operations to types as we did in Section 3.1, except that now we have a 3'rd class of read operations – those that return in line 18 of Algorithm 1 after being helped. We say that these are `Read` operations of type (iii).

Let R be a type (iii) `Read` operation by process i that returns value u and let $k' = \lfloor u/m \rfloor$, then there is a `Write` operation W by process j, concurrent with R, that wrote u to $\mathtt{H}[i][j]$ (in Line 9 of Algorithm 1) after performing a `ReadMax` operation on $max_{k'}$ (in Line 7 of Algorithm 1) and R returns value u after reading it from $\mathtt{H}[i][j]$ (Lines 8, 9 of `GetHelp`). We say that R *is associated with* that `ReadMax` operation.

As in Section 3.1, we partition the operations of E to the sets $Down_k$ and $Futile_k$, except that we now add each `Read` operation of type (iii) that is associated with a `ReadMax` on $\mathtt{max}_k$ to $Down_k$. We use the ordering rules defined in Section 3.1 to linearize all of E's non-removed operations. It is easily verified that Observations 2–3 and Claim 4 hold also with the extended definition of the sets $Down_k$.

▶ **Observation 11.** *Let R be a type (iii) `Read` operation associated with a `ReadMax` operation R' on $max_{k'}$. Then all throughout the execution of R', $\mathtt{switch}_{k'} = 0$ holds.*

Proof. Immediate from Claim 4 and the fact that the `Write` operation that invokes R' in line 7 of Algorithm 1 writes the value read by R' (in line 9) to the `H` array only after verifying that $\mathtt{switch}_{k'} = 0$ holds (in line 8). ◀

Based on Observation 11, we now prove that Claim 5 holds for full algorithm.

▷ Claim 12. In any ordering of operations for the full algorithm, for all $k' \leq k$, there are no two operations *Fop*, *Dop* such that $Fop \in Futile_k$, $Dop \in Down_{k'}$ and *Fop* is linearized before *Dop* in the ordering given by rules 1–3.

Proof. Suppose towards a contradiction that *Fop* is linearized before *Dop*. If *Fop* was placed after Dop_r when applying rule 3, then we immediately reach a contradiction. Assume otherwise, then, from rule 3, there exists a *Down* operation Dop_i, such that *Fop* precedes Dop_i, *Fop* is linearized before Dop_i, and no *Down* operation is linearized between *Fop* and Dop_i. Consequently, it must be that *Dop* is linearized after Dop_i. From rules 1-2, we have that $Dop_i \in Down_{k_1}$ such that $k_1 \leq k' \leq k$. Since $Dop_i \in Down_{k_1}$, either Dop_i reads 0 from $Switch_{k_1}$ or, otherwise, it is a type (iii) `Read` operation, in which case, from Observation 11, $Switch_{k_1} = 0$ holds at some point during its execution. However, *Fop* reads $Switch_k = 1$ before Dop_i starts, hence, by Claim 4, $Switch_{k_1} = 1$ when Dop_i starts. This is a contradiction. ◁

We next show that Lemma 6 holds also for the full algorithm.

▶ **Lemma 13.** *Ordering rules 1-3 for the full algorithm define a sequential order between E's non-removed operations that preserves the real-time order between non-overlapping operations in E.*

Proof. In Lemma 6, the corresponding claim was proven w.r.t. the algorithm without the helping mechanism for operations of all types, except for `Read` operations of type (iii). A type (iii) operation $R \in Down_k$ by process i is associated with a `ReadMax` operation R' on $\mathtt{max}_k$ invoked from a concurrent `Write` operation, performed by some process $j \neq i$. Since the condition of line 9 of `GetHelp` was satisfied when evaluated by i, the execution interval of R'

is fully contained within that of R. It follows that R can be linearized when R' is linearized on $\texttt{max}_k$. Thus, R is ordered w.r.t. other operations in $Down_k$ by applying ordering rule 2 to R' breaking ties arbitrarily which, from Observation 2 and the linearizability of max_k, ensures that real-time order is maintained between all operations in $Down_k$.

Let $Op \in Down_k$ and $Op' \in Down_{k'}$ be two operations such that $k < k'$. From Claim 4, M's switches are set to 1 in increasing order. Based on this and on Observation 11 (which is required if either Op or Op' is a type (iii) `Read` operation), Op' does not precede Op in E. it follows that the real-time order between each pair of $Down$ operations is preserved by the linearization.

Let Fop and Dop respectively be a $Futile_k$ and a $Down_{k'}$ operation. Suppose Fop is linearized before Dop but Dop precedes Fop in real-time order. If $k' \leq k$, then, by Claim 12, this is a contradiction. The rest of the proof proceeds exactly as in the proof of Lemma 6. ◀

It is easily verified that Lemma 7 holds also for the full algorithm. The only change required in its proof is to use Claim 12 instead of Claim 5.

▷ Claim 14. If a monotonically-increasing sequence of values is written to M, then some process performs line 9 of Algorithm 1 infinitely often.

Proof. If a 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 9 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 15.** *The full Algorithm 1 is wait-free.*

Proof. As proven in Lemma 8, the algorithm is lock-free and `Write` operations are wait-free. It remains to show that `Read` operations are wait-free as well. From Claim 14, if a monotonically-increasing sequence of values is written to M, then there is some process j that performs line 9 of Algorithm 1 infinitely often. Thus, any `Read` operation, say by process i, eventually either finds a non-obsolete `max` register in line 15 or increments $\texttt{C}_i[j]$ twice in line 9 of `GetHelp` and is therefore able to terminate.

Otherwise, there is no such sequence of monotonically-increasing values. Thus, starting from some point in the execution, M's value does not increase, so the set of obsolete `max` object stops growing, hence every `Read` operation that does not fail-stop eventually reaches a non-obsolete `max` register and completes. ◀

▶ **Theorem 16.** *If $m \geq n^2$, 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.*

Proof. From Lemmata 7, 13 and 15 the full algorithm is linearizable and wait-free, so it remains to argue regarding its complexity. In Algorithm 2, every iteration of the `for` loop at either line 3 or line 6 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 $k \cdot n$ steps, for some $k > 1$, when $c = 0 \mod n$. Thus, at any point in the course of the execution, the number of steps taken by a `Read` operation R inside `GetHelp` is $O(loop_R)$. Consequently, as in the proof of Lemma 9, we get:

$$AmtSteps(E) = O\Big(\big(\sum_{op \in Ops_W(E)} \log m + \sum_{op \in Ops_R(E)} \log m + loop_{op}\big)/(w+r)\Big) = O(\log m). \quad (6)$$

◀

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

Shared variables:
R: an n-process `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)

4 Wait-Free Counter with Polylogarithmic Amortized Step Complexity

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. [4], except that the latter uses the max registers of [4], whose amortized step complexity is linear for 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 `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 `ReadMax` 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 17.** *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 16. Increment operations on C_1 are linearized when the `WriteMax` operation invoked in line 4 is linearized and read operations on C_1 are linearized when the `ReadMax` operation invoked in line 11 is linearized.

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

Inductive Step.

▶ **Sub-Lemma 17.1.** *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 `WriteMax`() 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. 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 $u_1 \geq c_0 + c_1$.

As 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 $v - n \leq v'$. 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 operation 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 Sub-lemma 17.1 and Theorem 16 we conclude that $C_n.R$ is linearizable in E. Based on this, the proof proceeds similarly to the proof of [4, Lemma 4].

From IH, C_j.`left` and C_j.`right` are linearizable counters. 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 lies 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 Sub-lemma 17.1. 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 18.** *Algorithm 3 has $O(\log^2 n)$ amortized operation 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 $\texttt{UnboundedMaxReg}_{n^2}$ objects, as the recursive calls in lines 7–9 and 11 unfold. Letting $COps(E)$ denote the number of operations on $\mathcal{C}$ that appear in E, the total number of `ReadMax`/`WriteMax` operations on all the implementation's $\texttt{UnboundedMaxReg}_{n^2}$ objects is therefore $O\big(\log n \cdot COps(E)\big)$. From Theorem 16, letting $m = n^2$, it follows that the total number of steps performed in E is $O\big(\log^2 n \cdot COps(E)\big)$. ◀

▶ **Theorem 19.** *Algorithm 3 is a wait-free linearizable n-process implementation of an unbounded counter with amortized step complexity of $O(\log^2 n)$.*

Proof. From Lemma 17, the algorithm is linearizable. From Lemma 15, all the `Unbounded MaxReg` objects used by Algorithm 3 are wait-free, thus, clearly from the pseudo-code, Algorithm 3 is wait-free as well. The claimed complexity follows from Lemma 18. ◀

Attiya and Hendler proved 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 [9, Theorem 9]. Their proof can be easily adapted to obtain the following result:

▶ **Lemma 20.** *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 20 establishes that every non-blocking 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 non-blocking read/write counter algorithm that provides sub-linear amortized step complexity in all executions, regardless of their length. The amortized operation 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.

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 [4].

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. A second question is that of finding a bounded-space read/write counter with sub-linear amortized step complexity. These questions are left for future work.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
2. 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.
3. James Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.
4. James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012.
5. James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. *ACM Trans. Algorithms*, 6(2):25:1–25:23, 2010.
6. 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.
7. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.
8. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.
9. 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.
10. Michael A. Bender and Seth Gilbert. Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 728–737. IEEE, 2011.
11. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
12. 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.

13 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.

14 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *International Workshop on Distributed Algorithms*, pages 130–140. Springer, 1994.

15 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 '98*, pages 201–210, 1998.

16 Prasad Jayanti, King Tan, and Sam Toueg. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM J. Comput.*, 30(2), 2000.

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

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