HAL
open science

Upper and Lower Bounds for Deterministic Approximate Objects

Danny Hendler, Adnane Khattabi, Alessia Milani, Corentin Travers

To cite this version:

Danny Hendler, Adnane Khattabi, Alessia Milani, Corentin Travers. Upper and Lower Bounds for Deterministic Approximate Objects. ICDCS 2021 - 41st IEEE International Conference on Distributed Computing Systems, Jul 2021, Washington (virtual), United States. pp.438-448, 10.1109/ICDCS51616.2021.00049 . hal-03202712v2

HAL Id: hal-03202712

https://hal.science/hal-03202712v2

Submitted on 28 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Upper and Lower Bounds for Deterministic Approximate Objects

Danny Hendler
Ben-Gurion University of the Negev
Be'er Sheva, Israel
hendlerd@bgu.ac.il

Adnane Khattabi
LaBRI, University of Bordeaux
Bordeaux, France
adnane.khattabi-riffi@u-bordeaux.fr

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

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

***Abstract*—Relaxing the sequential specification of shared objects has been proposed as a promising approach to obtain implementations with better complexity. In this paper, we study the step complexity of relaxed variants of two common shared objects: max registers and counters. In particular, we consider the k-multiplicative-accurate max register and the k-multiplicative-accurate counter, where read operations are allowed to err by a multiplicative factor of k (for some $k \in \mathbb{N}$). More accurately, reads are allowed to return an approximate value x of the maximum value v previously written to the max register, or of the number v of increments previously applied to the counter, respectively, such that $v/k \leq x \leq v \cdot k$. We provide upper and lower bounds on the complexity of implementing these objects in a wait-free manner in the shared memory model.**

***Keywords*-Distributed computing, distributed algorithms, shared memory, fault tolerance, concurrent data structures, relaxed specifications**

I. Introduction

With the ubiquitousness of multi-core and multi-processor systems, there is a growing need to gain better understanding of how to implement concurrent objects with improved complexity, while maintaining the natural correctness guarantee provided to programmers by linearizability. Relaxing the sequential specification of linearizable concurrent objects is one promising approach of achieving this [1], [2]. An object's *sequential specification* defines its correct behavior in sequential executions. Roughly speaking, *linearizability* [3] guarantees that any concurrent execution is equivalent to a sequential one.

There is empirical evidence that relaxing the sequential specification of some common objects, e.g. queues and counters, yields improved performance of linearizable implementations, e.g [2], [4]. However, the theoretical principles to implement concurrent objects more efficiently by relaxing their sequential specification are not yet clear.

In this paper, we study relaxed-semantics variants of two well-known concurrent objects – counters and max registers, in the classical shared memory model. In particular, we investigate the extent to which allowing wait-free linearizable implementations of these objects to return approximate values, rather than accurate ones, may improve their step complexity.

A counter is a linearizable object that supports a $CounterIncrement$ operation and a $CounterRead$ operation. The sequential specification of a counter requires that a $CounterRead$ operation returns the number of $CounterIncrement$ operations that precede it. A relaxed variant of the counter is the *k-multiplicative-accurate* counter, defined in [5], where a $CounterRead$ operation returns an approximate value x of the number v of $CounterIncrement$ operations that precede it, such that $v/k \leq x \leq v \cdot k$ for some parameter $k > 0$.

A max register r supports a $Write(v)$ operation that writes a non-negative integer v to r and a $Read$ operation that returns the maximum value previously written to r, [5]. We define the k-multiplicative-accurate max register by allowing a $Read$ operation to return an approximate value x of the largest value v written before it, such that $v/k \leq x \leq v \cdot k$ for some parameter $k > 0$.

A. Related Work

A well-known result by Jayanti, Tan and Toueg [6] proved a linear lower bound in the number of processes n on the worst-case step complexity of obstruction-free implementations from historyless primitives (e.g. read,write, test&set) of a large class of shared objects that includes exact counters. A wait-free exact counter with optimal worst case step complexity can be constructed easily by using a *wait-free atomic snapshot* : to increment the counter, a process simply increments its component of the snapshot, and to read the counter's value, it 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, with worst-case step complexity linear in n, e.g. [7], so can counters.

Aspnes, Attiya and Censor-Hillel [8] show the possibility of implementing exact counting algorithms whose step complexity is sub-linear when the number of operations is bounded. In particular, they presented a wait-free exact counter for which the step complexities of $CounterIncrement$ and $CounterRead$ 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, for executions in which the number of $CounterIncrement$ operations

is exponential in n, both the worst-case and the amortized step complexities of their constuction become linear in n.

In [9] Baig et al. present the first wait-free read/write exact counter whose amortized step complexity is polylogarithmic in n, $O(\log^2 n)$, in executions of arbitrary length and prove that their algorithm is optimal in terms of amortized step complexity up to a logarithmic factor.

Approximate counting has many applications (e.g.; [10], [11]) and there is a large literature in approximate probabilistic counting, both in the sequential (e.g. [12], [13]) and concurrent setting (e.g. [14], [15]). In [14], Aspnes and Censor present a randomized algorithm to implement an approximate counter that requires sublinear step complexity, and where any $CounterRead$ operation has a *high probability* of returning a value which is at most a fraction of δ less than the number of increments that have finished before the read started, and at most a fraction of δ more than the number of increments that have started before the read finished. Similarly to their counter, the k-multiplicative accurate counter we study in this paper, allows a multiplicative error of the exact value. However, we study approximate *deterministic counting*, where the value returned by $CounterRead$ operations is always ensured to be within the given approximation range.

Aspnes et al. also study k-additive counters that allow some additive error for the value returned by $CounterRead$ operations, [8]. In particular they prove a lower bound of $\Omega(min(n-1, \log m - \log k))$ on the worst-case step complexity of any deterministic asynchronous linearizable implementation of a k-additive counter, where m is the number of states of the counter and n is the number of processes. No matching upper bound is given.

The exact max register object has been proposed in [8] where Aspnes et al. present a bounded variant of this object to beat the linear lower bound on the worst case step complexity by Jayanti et al. In particular, their algorithm has $O(\log m)$ worst case step complexity for both $Read()$ and $Write(v)$ operations provided that the value written in the max register does not exceed the value m. Their algorithm is optimal [5]. Baig et al. [9] presented an unbounded deterministic max register implementation with polylogarithmic *amortized* step complexity in executions of arbitrary length.

B. Our Contribution

k-multiplicative-accurate counter: To the best of our knowledge we present the first deterministic approximate counter with *constant amortized complexity*. More precisely, we present a wait-free linearizable k-multiplicative-accurate counter for $k \geq \sqrt{n}$ where n is the number of processes, with *constant* amortized step complexity for executions of arbitrary length. Then, by extension of the lower bound of Attiya and Hendler, [16], we prove that any n-process solo-terminating implementation of a k-multiplicative-accurate counter from read/write and conditional primitive operations (including k-word compare-and-swap) has amortized step complexity of $\Omega(\log(n/k^2))$, for $k \leq \sqrt{n}/2$. Our results together with the upper and lower bound on exact counting proved in [9] show that when the approximation parameter k does not depend on n, relaxing the counter semantics by allowing a multiplicative error cannot asymptotically reduce the amortized step complexity by more than a logarithmic factor.

We also prove that the *worst-case step complexity* of obstruction-free implementations of m-bounded k-multiplicative-accurate counters from *historyless* primitives is $\Omega(\min(n, \log_2 \log_k m))$, where n is the number of processes and m is a bound on the number of $CounterIncrement$ operation instances that can be performed on the counter. This implies that for unbounded k-multiplicative-accurate counters, the worst case step complexity is in $\Omega(n)$, and we fall back to the linear lower bound by Jayanti, Tan and Toueg [6].

k-multiplicative-accurate max register: We prove that relaxing the semantics of the bounded max register by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity. In particular, we prove that the worst-case step complexity of obstruction-free read/write implementations of m-bounded k-multiplicative-accurate max registers is $\Omega(\min(n, \log_2 \log_k m))$, where n is the number of processes. A max register is m-bounded, if it can only represent values in $\{0, \ldots, m-1\}$. Then, we present a novel m-bounded k-multiplicative-accurate max register algorithm whose worst-case step complexity matches this lower bound. We can easily "plug-in" our bounded k-multiplicative-accurate max register into the construction proposed by Baig et al. [9] to obtain an unbounded k-multiplicative-accurate max register with sub-logarithmic amortized step complexity (omitted due to space constraints).

II. Model and Preliminaries

We consider an asynchronous shared memory system, where a set $\mathcal{P}$ of n crash-prone processes communicate by applying operations to shared objects. An *object* is an instance of an abstract data type. It is characterized by a domain of possible values and by a set of *operations* that provide the only means to manipulate it. An *implementation* of a shared object provides a specific data-representation for the object from a set of shared *base objects*, each of which is assigned an initial value; the implementation also provides algorithms for each process in $\mathcal{P}$ to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects *primitives* and reserve the term *operations* for the objects being implemented.

An *execution fragment* is a (finite or infinite) sequence of steps performed by processes as they follow their algorithms. In each step, a process applies at most a single primitive

to a base object (possibly in addition to some local computation). We consider read, write and test&set primitives. An *execution* is an execution fragment that starts from the *initial configuration*. This is a configuration in which all base objects have their initial values and all processes are in their initial states.

A set of primitives is *historyless* if all the nontrivial primitives in the set overwrite each other; we also require that each such primitive overwrites itself. A primitive is *nontrivial* if it may change the value of the base object to which it is applied.

Operation op_1 *precedes* operation op_2 in an execution E, if op_1's response appears in E before op_2's invocation. We consider only *deterministic implementations*, in which the next step taken by a process depends only on its state and the response it receives from the event it applies.

Roughly speaking, an implementation is *linearizable* [3] if each operation appears to take effect atomically at some point between its invocation and response; it is *wait-free* [17] if each process completes its operation if it performs a sufficiently large number of steps; it is *obstruction-free* (also called *solo-terminating*) [18] if each process completes its operation if it performs a sufficiently large number of steps when running solo.

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. 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:

$$AmtSteps(A) = \max_E \frac{\sum_{op \in Ops(E)} Nsteps(op, E)}{|Ops(E)|}$$

III. Unbounded Approximate k-multiplicative-accurate Counter

We present a wait-free linearizable unbounded k-multiplicative-accurate counter with $k \geq \sqrt{n}$ whose amortized step complexity is constant (Algorithm 1).

The algorithm uses an unbounded sequence of bits initially equal to 0, denoted $switch_0, switch_1, \ldots$ to approximately keep track of the number of increments that have been performed by the processes. For each $i \geq 0$, $switch_i$ can be accessed by $test\&set$ and $read$ operations. $switch_i.test\&set()$ sets the value of $switch_i$ to 1 and returns its previous value. A $read$ simply returns the value of $switch_i$.

In a nutshell, each process locally keeps an accurate count of the number of $CounterIncrement$ operations it performs and that are not yet known by the other processes. When this count reaches a certain threshold, the process tries to inform other processes of the number of increments it has performed locally, by attempting to set to 1 a switch in an appropriate bounded range. When a process succeeds in setting a switch to 1, it will restart the local count from 0. $switch$ bits are set in increasing order with regards to their index, one after the other.

In particular, the initial value of the threshold is 1 and after their first call to $CounterIncrement$, each process will attempt to set $switch_0$. Afterwards, the sequence of $switch_i$ with $i \geq 1$ is partitioned into consecutive intervals of size k. For any such interval $[qk+1, (q+1)k]$, where k is an integer, and for any $j \in [qk+1, (q+1)k]$, $switch_j$ equals to 1 indicates that k^{q+1} instances of $CounterIncrement$ have been performed by some process. In other words, a process p locally performs k^{q+1} instances of $CounterIncrement$ before attempting to set a switch in the interval $[qk+1, (q+1)k]$ and it increments its local threshold only if it knows that the last switch in this interval is set to 1 (i.e.; at least $k \cdot k^{q+1}$ instances of $CounterIncrement$ have been performed). The threshold is incremented by a factor k. There is no guarantee that p will succeed in setting to 1 one of the switches. But in this case, sufficiently many increments have been performed by the processes so that a $CounterRead$ operation can safely ignore the increments kept locally by p_i and still returns a value within a bounded factor of the actual number of increments.

By using test&set to modify a $switch$ from 0 to 1, we ensure that the $CounterIncrement$ instances accounted for by $switch_j$ are distinct from those accounted for by $switch_{j'}$, for any $j' \neq j$.

Performing an instance of a $CounterRead$ operation op consists in traversing the sequence of switches until 0 is found. An approximation of the total number of $CounterIncrement$ is then deduced from the index of the last switch j that op found equal to 1. The value returned is the sum of the $CounterIncrement$ operations represented by each switch from $switch_0$ to $switch_j$. In particular, $switch_0$ counts for one $CounterIncrement$, and each $switch_i$ in an interval $[qk+1, (q+1)k]$ for some integer $q \leq q_j$ counts for k^{q+1} $CounterIncrement$ operations, where $switch_j$ belongs to the interval $[q_j k+1, (q_j+1)k]$.

A. The $CounterIncrement$ operation

Each process i is equipped with two persistent local variables, $\texttt{lcounter}_i$ and $\texttt{limit}_i$. The former stores the number of $CounterIncrement$ instances performed by process i not yet announced to the other processes; and the latter stores the treshold on the number of $CounterIncrement$ that can be performed by process i without informing the other processes.

When a $CounterIncrement$ operation is invoked by a process i, $\texttt{lcounter}_i$ is first incremented (line 11). To ensure that a $CounterRead$ operation instance returns a

value that is within a multiplicative factor k of the actual number of increments, when $\texttt{lcounter}_i$ reaches a certain threshold stored in $\texttt{limit}_i$, process i tries to inform the other processes of the number of increments it has performed locally (lines 12). The value of $\texttt{limit}_i$ is initially 1 and is multiplied by k each time it is modified (line 21 and line 28). When $\texttt{lcount}_i = \texttt{limit}_i = k^{q+1}$ for some integer q, process i tries to set to 1 one of the k $switch_j$ whose index j is in the corresponding range $[qk+1,(q+1)k]$ (lines 15- 23). If it succeeds, it resets the local counter $\texttt{lcounter}_i$. The number of $CounterIncrement$ instances it has performed locally has been announced to the other processes, and thus will be taken into account by future $CounterRead$ operations.

Additionally, process i writes the index of the switch it sets together with a sequence number into a shared variable $H[i]$ (lines 17 and 18). As explained later this pair is intended to help $CounterRead$ operation instances to complete. Finally, the process will also update the value of the local persistant variable l_0 to indicate the index of the switch it managed to set within the interval (line 22). By doing so, we ensure that the process will avoid attempting to reset the same switches every time it reaches the treshold of $limit_i$ in the current interval by starting from the index $qk + l_0$ in the next attempt. If it does not succeed, every $switch_j$, where $j \in [qk+1,(q+1)k]$ is set. We show in the proof that for $k \geq \sqrt{n}$, this number is sufficiently large for allowing $CounterRead$ operations to return values within a factor k of the total number of $CounterIncrement$ instances (Section III-C). The threshold $\texttt{limit}_i$ is then incremented by a factor k (line 28) and the value of l_0 is reset to 1 (line 24).

B. The $CounterRead$ operation

When a $CounterRead$ operation is invoked, process i scans the first and last $switch$ of each interval of k switches, looking for the first one that is not yet set to 1. When such a switch is found, the index h of the last switch read that was equal to 1 is stored in the persistent local variable $\texttt{last}_i$ to avoid scanning the sequence from the beginning each time. We compute the value ret returned by the $CounterRead$ operation in the function $ReturnValue(p,q)$ where $h = q \cdot k + p$ (line 30). First, we consider the required increments needed to set all the switches in the current interval $[qk+1,(q+1)k]$ by adding to ret the value $p \cdot k^{q+1}$ (line 31). Next, we add 1 to ret to account for the first $switch_0$ (line 31), and then for each previous interval $[(l-1)k+1, lk]$ where $1 \leq l \leq q$, we add k^{l+1} to ret (line 33). Finally, we return the computed value ret multiplied by a factor k to ensure ret falls in the approximation range of the k-multiplicative-accurate counter.

However, it may be the case that the condition at line 37 is never verified, as other processes may concurrently keep executing $CounterIncrement$ operations. Thus, to ensure wait-freedom, we employ the following helping mechanism : a $CounterIncrement$ operation by a process i that succeeds to set a $switch_j$, writes the index j of this switch together with a sequence number in the shared register $H[i]$ (lines 17 and 18). A $CounterRead$ operation op that fails to find a switch to 0 after $\theta(n)$ steps, reads all the n shared registers $H[i]$ with $i \in 1,\ldots,n$. If a consistent value is found, then it returns at line 55. Otherwise, it executes another $\theta(n)$ steps. The first time op scan the array H, it stores the sequence number read in each $H[j]$, denotes sn_j. When scanning H again, op will select a pair whose timestamp is greater than or equal to sn_j +2. This ensures, that the corresponding switch has been set by process j in the execution interval of op.

C. Proof

1) Wait-freedom and technical lemmas: Let E be an execution of the k-multiplicative-accurate unbounded counter implemented in Algorithms 1.

Lemma III.1. *Operations $CounterIncrement()$ and $CounterRead()$ are wait-free.*

Proof: Let op_r and op_w denote a $CounterRead$ and $CounterIncrement$ instance respectively in E. The number of steps taken during op_w is bounded since at most the process will attempt to set k switches during a call to $CounterIncrement$ and there are no other loops or function calls in the $CounterIncrement$ operation.

Suppose by contradiction that op_r does not terminate. Meaning that every bit $switch_\ell$ it reads has been set to 1. Since the bits are initially 0, there is at least one process q that infinitely often performs a successful test&set operation on these bits. Note that each time this occurs, q increments its sequence number $\texttt{sn}_q$ and reports the new value in the helping array H (lines 17- 18). As every n iterations of the **while** loop, op_r scans the array H, it will eventually detect that the sequence number of q has been incremented at least twice, hence op_r terminates via the helping mechanism (lines 50-55). Therefore, operations $CounterIncrement$ and $CounterRead$ are wait-free. ■

We continue with a few technical lemmas.

Lemma III.2. *Switches are set to 1 in E in increasing order of their index, starting from* switch_0.

Proof: For each process p the initial value of $limit_p$ is 1 and of $counter_p$ is 0, thus the first $CounterIncrement$ operation by process p applies a test&set primitive to $switch_0$ according to lines 11, 12, 13, and 27. We now prove that for any given process p and for any $j \geq 1$, p applies a test&set primitive (if any) on each of the switches with indexes in the interval $[(j-1)\cdot k+1,\ldots j\cdot k]$ in an increasing order of their index, starting from $switch_{(j-1)\cdot k+1}$. First observe that for any process p, the initial value of l_0 is 1, and l_0 is set to 1 iff the value of $limit_p$ is multiplied by a factor k

Algorithm 1: k-multiplicative-accurate unbounded counter, code for process i.

```
1  Shared variables
2      switch_j ∈ {0,1} : for each j ∈ ℕ, a 1-bit register that supports test&set and read primitives, initially all 0
3      H[n] : an array of n integer pairs (val, sn)
4  Persistent local variables
5      last_i ∈ ℕ_0 : largest index of a switch accessed by i, initially 0
6      lcounter_i: number of unannounced CounterIncrement by process i, initially 0
7      limit_i : number of CounterIncrement that process i can perform locally, initially 1
8      sn_i: number of switches set to 1 by process i, initially 0
9      l_0: index of last switch accessed by the process i in the current set of switches, initially 1
10 Function CounterIncrement()
11     lcounter_i ← lcounter_i + 1
12     if lcounter_i = limit_i then
13         j ← log_k(lcounter_i)
14         if j > 0 then
15             for ℓ ← (j − 1)k + l_0, ..., j · k do
16                 if switch_ℓ.test&set() = 0 then
17                     sn_i ← sn_i + 1
18                     H[i] ← (ℓ, sn_i)
19                     lcounter_i ← 0
20                     if ℓ = jk then
21                         limit_i ← k · limit_i
22                     l_0 ← 1 + ℓ mod k
23                     return
24             l_0 ← 1
25         else
26             if switch_0.test&set() = 0 then
27                 lcounter_i ← 0
28         limit_i ← k · limit_i
29     return
30 Function ReturnValue(p,q)
31     ret ← 1 + p · k^(q+1)
32     if q ≥ 1 then
33         ret ← ret + Σ_{l=1}^{q} k^(l+1)
34     return k · ret
35 Function CounterRead()
36     c ← 0
37     while switch_last_i ≠ 0 do
38         p ← last_i mod k
39         q ← ⌊last_i / k⌋
40         if last_i mod k = 0 then
41             last_i ← last_i + 1
42         else
43             last_i ← last_i + k − 1
44         c ← c + 1
45         if c mod n = 0 then
46             if c = n then
47                 for j ← 1, ..., n do
48                     help_i[j] ← H[j].sn
49             else
50                 for j ← 1, ..., n do
51                     (val, sn) ← H[j]
52                     if sn − help_i[j] ≥ 2 then
53                         p ← val mod k
54                         q ← ⌊val / k⌋
55                         return ReturnValue(p, q)
56     if last_i = 0 then
57         return 0
58     return ReturnValue(p, q)
```

(lines 24,28 and lines 21,22). This implies that when a new j is computed at line 13, the value of l_0 is 1.

Then the first iteration of the **for loop** at line 15 starts at $l = (j-1)\cdot k+1$. Also, the value of l is incremented by one at each iteration of the for loop at line 15 unless p succesfully sets a $switch_{(j-1)k+i}$ with $i \in \{1, \ldots, k\}$. In this latter case, the value of l_0 is modified at line 22 and takes the value $i+1$ if $i < k$, or 1 otherwise (we reach the end of the set). If l_0 takes a value different from 1, that is $l \neq j \cdot k$, (otherwise the claim is proved), then the *CounterIncrement* operation returns at line 23 without modifying the value of $limit_p$. Thus, in the execution of a successive *CounterIncrement* operation (if any), process p will apply the next test&set primitive (if any) to $switch_{jk+i+1}$ (because of lines 12, 13, 15).

The value of $limit_i$ is multiplied by k (and than the value of j is incremented by one) only after a process has applied a test&set primitive (both successfully or not) to the last switch in the current interval $[(j-1)\cdot k+1, \ldots, j \cdot k]$ with $log_k(limit_i) = j$ (lines 21, 28). This completes the proof. ■

Lemma III.3. *For any given execution E, if a CounterRead operation op returns the value computed in $ReturnValue(p, q)$ at line 55, then $switch_{q\cdot k+p}$ was equal to 0 before the invocation of op and the test&set primitive that sets $switch_{q\cdot k+p}$ to 1 is applied during the execution interval of op.*

Proof: At line 51, op reads a pair (val, σ) from an entry $H[p']$ of the helping array H where $val = q\cdot k+p$. According

to lines 16, 17, and 18, a unique process p' sets to 1 the $switch_{val}$ and associates with val the sequence number σ computed at line 17, before writing the pair (v, σ) to $H[p']$ in the execution of a $CounterIncrement$ operation op'.

Let p be the process that executes the $CounterRead$ operation op. Denote by σ' the value of $H[p'].sn$ read by p at line 48 in the execution of op. According to line 52, $\sigma - \sigma' \geq 2$. This means that process p' executes line 17 at least twice during the execution interval of op. In particular p' executes the step that set $switch_{val}$ to 1 after op was invoked by p. This proves the claim. ■

2) Linearizability: We next define the linearization L of the operations in E by first removing any $CounterRead$ operation that did not complete and any incomplete $CounterIncrement$ operation that has not successfully executed line 16.

Let OP_W be the set of (complete and incomplete) $CounterIncrement$ operations that sucessfully set a switch while executing line 16. Let OP_{LO} be the remaining complete $CounterIncrement$ operations in E and OP_R be the set of complete $CounterRead$ operations in E. Observe that each $CounterIncrement$ operation succesfully sets at most one switch, and each switch is succesfully set by at most one process. Thus we can univocally associate each operation in OP_W with the switch it sets. We order the operations in $OP_W \cup OP_{LO} \cup OP_R$, according to the following rules :

1) We linearize each operation in OP_W at the step where it sets its corresponding switch. By claim III.2, operations in OP_W are totally ordered and this order respect the real-time order. In the following we denote opw_i the i-th operation in OP_W according to our linearization order with $i \geq 0$.
2) We linearize a $CounterRead$ operation opr according to whether it returns normally or through the helping mechanism:
   a) If opr returns $ReturnValue(p, q)$ normally at line 58, then it is linearized at the step where it reads the value 1 of $switch_{q \cdot k+p}$ at line 37 . This is well-defined because this read primitive exists and it is unique (it is easy to check from the pseudo-code).
   b) If opr returns $ReturnValue(p, q)$ via the helping mechanism at line 55, then the operation is linearized immediately after $opw_{q \cdot k+p}$.
3) Let L_{WR} denote the linearization of all operations in $OP_W \cup OP_R$ according to rule 1 and 2, we linearize an operation op in OP_{LO} immediately before the first operation op' in L_{WR} that follows op in the real-time order or at the end of L_{WR} if op' does not exist.

$CounterRead$ operations that returns 0 after reading $switch_0 = 0$ are linearized before opw_0. If several operations are ordered at the same position, they are ordered respecting their real time order.

Linearization rule 2 and Lemma III.3 implies the following claim.

Claim III.4. *Let opr be a CounterRead operation. We have that opr is linearized at some point after its invocation.*

Lemma III.5 (Linearizability)**.** *Algorithm 1 is a linearizable implementation of a k-multiplicative-accurate unbounded counter.*

Proof: Let op_1 and op_2 be two operations in E such as op_1 ends before op_2 is invoked. We prove that the linearization order L respects the real-time order, thus op_1 precedes op_2 in L. First, we have the following claim:

- Let op_1 and op_2 be two $CounterIncrement$ operations. If at least one of these operations is in OP_{LO}, the claim trivially follows from rule 3. Otherwise it is already proved in rule 1.
- Let op_1 and op_2 be two $CounterRead$ operations. If both op_1 return normally the claim trivially holds from rule 2a and claim III.4. So consider that op_1 returns through the helping mechanism and let $h_1 = q \cdot k + p$ be the index of the switch read by op_1 at line 54, the last time before returning. According to rule 2b, op_1 is linearized immediately after opw_{h_1}. Also, by Lemma III.3 and rule 2, op_2 is linearized after opw_{h_1}. The claim follows since according our linearization rules, If several operations are ordered at the same position, they are ordered respecting their real time order.
- Consider that op_1 is a $CounterIncrement$ and op_2 is a $CounterRead$ operation. The claim follows from rules 1 and 2 and claim III.4 (the reverse follows a similar reasoning).

Next claim will be useful for proving that the ordering L is consistent with the sequential specification of the k-multiplicative-accurate counter.

Claim III.6. *Let op be a CounterRead operation invoked by a process p_i that returns $ReturnValue(p, q)$. Then, the number of CounterIncrement operations linearized before op in L, denoted v, is at least $u_{min} = 1 + \sum_{l=1}^{q} k^{l+1} + p \cdot k^{q+1}$ and at most $u_{max} = 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1} - 1)$ where n is the number of processes.*

Proof: Let op be a $CounterRead$ operation invoked by a process p_i that returns $ReturnValue(p, q)$ and let $h = q \cdot k+p$ with $p \geq 0$. Consider the $CounterIncrement$ operation by p_j that set to 1 the switch$_h$, denoted opw_h.

op is linearized at the step where it reads $switch_h$ if it returns normally, or immediately after opw_h. Thus, from our linearization rules, the minimal number of $CounterIncrement$ operations that are linearized before op includes each opw_i in OP_W with $0 \leq i \leq h$, and every $CounterIncrement$ in OP_{LO} linearized before op.

We have by construction that each $switch_s$ in the $(l+1)$-th set of k switches indexed in the interval $[l \cdot k + 1 \ldots (l +$

$1)k]$ with $l \geq 0$, requires a process to perform k^{l+1} $CounterIncrement$ operation instances before attempting to set $switch_s$ to 1. In other words a process p_i needs its local variable $lcounter_i$ to be equal to k^{l+1} before it can attempt to set any $switch_s$ in $[l \cdot k+1 \ldots (l+1)k]$ (line 12). Since the value of $lcounter_i$ is reset to 0 after a succesful $test\&set$ primitive is applied on a switch (line 19), the sets of $CounterIncrement$ operation instances associated with any pair of succesful $test\&set$ primitives are disjoint. Thus, $u_{min} = 1+k\sum_{l=0}^{q-1} k^{l+1}+p\cdot k^{q+1} = 1+k\sum_{l=1}^{q} k^{l}+p\cdot k^{q+1}$ since we account for, in addition to the p switches in the $(q+1)$-th set and $switch_0$, all k switches in each of the sets indexed from 1 to q.

Similarily, we compute an upper bound u_{max} on the maximum number of $CounterIncrement$ linearized before op. First suppose that op returns normally. As already said, op is linearized at the step where it reads $switch_h$ with $h = qk + p$. We have two possible cases either p is equal to 0 or it is equal to 1 because the process checks the first and last switch of each set during the $CounterRead()$ instance. These two cases are depicted in Figure 1 a) and b) respectively. If p is equal to 0, then process p_i read $switch_{kq+1} = 0$ in the execution of op, and according to our linearization rules opw_{kq+1} is linearized after op. In a similar way, if p is 1 p_i read $switch_{(q+1)k} = 0$ and $opw_{(q+1)k}$ is linearized after op. However, in this second case, all the $k-1$ switches$_j$ with $j \in [q \cdot k+2 \ldots (q+1)k-1]$ may have been set to 1 before op applied its read to switch$_{qk+1}$, and all the corresponding opw_j may be linearized before op. Thus, the number of opw linearized before op is smaller than or equal to $1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1}$. It remains to count the number of $CounterIncrement$ in OP_{LO} linearized before op. For every process p_i the value of $lcounter_i$ is smaller than k^{q+1} immediately before p read either $switch_{kq+1} = 0$ or $switch_{(q+1)k} = 0$ in the execution of op. Since a process resets the value of its local counter only when it succeeds to set a switch to 1 (line 19), $lcounter_i$ defines the number of $CounterIncrement$ by p_i in OP_{LO} that are linearized before op. Therefore, $u_{max} = 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1}-1)$ where n is the number of processes. If op returns via the helping mechanism, then according to rule 2b, it is linearized immediately after $opw_{q\cdot k+p}$ with $0 \leq p < k$. Thus, $1 + \sum_{l=1}^{q} k^{l+1} + pk^{q+1}$ is the number of $CounterIncrement$ in OP_W linearized before op. Since $p < k$ the local counter of every process immediately after $opw_{q\cdot k+p}$ sets the corresponding switch is smaller than k^{q+1}. Since $k > 1$, the claim follows. ■

Let op be a $CounterRead$ operation and let $v_{op} = ReturnValue(p, q)$ be the value it returns. According to lines 31, 33 and 34 of Algorithm 1, $v_{op} = k(1+\sum_{l=1}^{q} k^{l+1} + p\cdot k^{q+1})$; that is $v_{op} = k \cdot u_{min}$. According to claim III.6, the number of $CounterIncrement$ operations linearized before op in L, denoted u, is at least $u_{min} = 1 + \sum_{l=1}^{q} k^{l+1} + p \cdot k^{q+1}$ and at most $u_{max} = 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1}-1)$ (where n is the number of processes). And we have:

$$\frac{u_{max}}{k} = \frac{1}{k} + \sum_{l=1}^{q} k^l + p\frac{k-1}{k}k^{q+1} + \frac{n}{k}(k^{q+1}-1)$$

$$\frac{u_{max}}{k} \leq \sum_{l=1}^{q} k^l + p \cdot k^{q+1} + n \cdot k^q$$

$$\text{And } v_{op} = k(1 + k\sum_{l=1}^{q} k^l + p \cdot k^{q+1})$$

$$= k(1 + k\sum_{l=1}^{q-1} k^l + k^{q+1} + p \cdot k^{q+1})$$

$$= k + k\sum_{l=2}^{q} k^l + p \cdot k^{q+2} + k^{q+2}$$

Thus, for $k \geq \sqrt{n}$, $\frac{u_{max}}{k} \leq v_{op}$. Therefore, since $p < k$, we have $\frac{u}{k} \leq \frac{u_{max}}{k} \leq v_{op} \leq k \cdot u_{min} \leq k \cdot u$. This completes the proof. ■

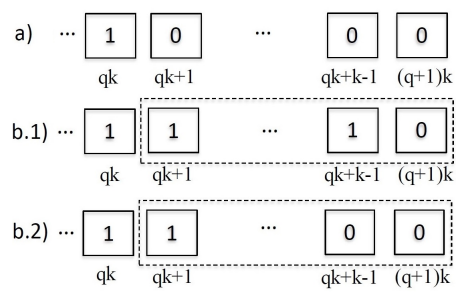

Figure 1. Switches state for the proof of claim III.6. The dotted line indicate the $q+1$th interval of consecutive switches.When $p = 1$, op does not distinguish between cases b.1) and b.2)

3) Complexity analysis:

Lemma III.7. *If process p applies a $test\&set()$ primitive to a switch$_\alpha$ with $i\cdot k+1 \leq \alpha \leq (i+1)\cdot k$ for some integer $i \geq 0$, then p has performed at least k^{i+1} $CounterIncrement()$ operations.*

Proof: Suppose that p has executed a $test\&set()$ primitive to a switch$_\alpha$ with $i\cdot k+1 \leq \alpha \leq (i+1)\cdot k$ in the execution of a $CounterIncrement()$ operation op. According to line 15, j was equal to $i+1$ when computed at line 13, meaning that $\texttt{lcounter}_p$ was equal to k^{i+1}. The claim holds because $\texttt{lcounter}_p$ is incremented only at line 11, that is once for each $CounterIncrement()$ operation performed by p. ■

Lemma III.8 (Amortized complexity)**.** *For $k \geq \sqrt{n}$, the amortized complexity of Algorithm 1 is constant.*

Proof: Let E be a finite execution of the unbounded k-multiplicative-accurate counter object implemented in Algorithm 1. Let r denote the number of $CounterRead()$ instances in E and s be the number of $CounterIncrement()$

instances in E. We additionally denote $Ops_W(E)$ the set of $CounterIncrement()$ operations that execute at least one step in E, and $Ops_R(E)$ the set of $CounterRead()$ operations in E. We want to compute

$$AmtSteps(E) = \frac{\sum_{op \in Ops_W(E) \cup Ops_R(E)} Nsteps(op, E)}{r+s}$$

where $Nsteps(op, E)$ is the number of steps executed by op in E.

Let $Ops_{W_p}(E)$ denote the $CounterIncrement()$ operations in $Ops_W(E)$ executed by process p and s_p denote the total number of $CounterIncrement()$ operations executed by process p. Let α_p be the index of the furthest switch accessed by a process p when executing any of the $CounterIncrement()$ operations in $Ops_{W_p}(E)$. We have that $i_p \cdot k + 1 \leq \alpha_p \leq (i_p + 1) \cdot k$ for some integer $i_p \geq 0$ (the case where $\alpha_p = 0$ is trivial).

In the worst case, process p applies a $test\&set()$ primitive to switch$_h$ for every $h \in [0, \ldots \alpha_p]$ and one additional step to write into $H[p]$ (line 18) each time p successfully set one of those switches. On the other hand, by Lemma III.7 if process p applies a $test\&set()$ primitive to the switch$_{\alpha_p}$, then it has performed at least $k^{i_{p+1}}$ $CounterIncrement()$ operations. Therefore,

$$\sum_{op \in Ops_{W_p}(E)} Nsteps(op) \leq 2 \cdot (i_p + 1)k + 1$$
$$\text{And } s_p \geq k^{i_p+1}$$

Thus, the total number of steps executed by the set of all processes $\mathcal{P}$ in order to perform the $CounterIncrement()$ operations in E is :

$$\sum_{op \in Ops_W(E)} Nsteps(op) = \sum_{p \in \mathcal{P}} \sum_{op \in Ops_{W_p}(E)} Nsteps(op)$$
$$\leq \sum_{p \in \mathcal{P}} 2 \cdot (i_p + 1)k + 1$$
$$\text{And } s = \sum_{p \in \mathcal{P}} s_p \geq \sum_{p \in \mathcal{P}} k^{i_p+1}$$

Now we consider the number of steps applied by each process to perform $CounterRead$ operations. Let α be the index of the furthest switch set to 1 by any process in $\mathcal{P}$. If $\alpha = 0$ then the claim follows. Then suppose $i \cdot k + 1 \leq \alpha \leq (i+1) \cdot k$ for some integer $i \geq 0$. For any sequence of switches with the index in $[j \cdot k + 1, \ldots, (j+1) \cdot k]$ with $0 \leq j \leq i$ a process p only reads the first and the last switch in such interval (i.e., $switch_{j \cdot k+1}$ and $switch_{(j+1) \cdot k}$). This is because at the beginning $last_p$ is equal to 0 and it is incremented by 1 if it is a multiple of k (at line 41), by $k - 1$ otherwise (line 43). Also, $last_p$ is a persistent variable, thus a process p reads a given switch that has been set to 1 at most once. This implies that the total number (in all its $CounterRead$ operations) of read primitives applied by a process p to the switches is less or equal to $2(i+2)$ (2 per each of the $i+1$ intervals, plus $switch_0$ and $switch_{\alpha+1}$). Furthermore, any $CounterRead()$ operation executes $O(n)$ steps of the **for** loop at line 47 or line 50 once every n iterations of the **while** loop (when the condition of line 45 is satisfied). This means that the total number of steps executed by a process p when performing its $CounterRead()$ operations is less or equal to $4(i+2)$. Thus,

$$\sum_{op \in Ops_R(E)} Nsteps(op) \leq \sum_{p \in \mathcal{P}_r} 4(i+2) \quad \leq 4(i+2) \cdot n_r$$

where $\mathcal{P}_r$ is the set of processes who have invoked at least one $CounterRead()$ operation and n_r is the cardinality of $\mathcal{P}_r$. Consider $n_r > 0$, the other case is trivial. Therefore:

$$AmtSteps(E) \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1)k + 1}{\sum_{p \in \mathcal{P}} k^{i_p+1} + r} + \frac{4(i+2) \cdot n_r}{s+r}$$

Furthermore, by lemma III.7 the minimum number of instances of the $CounterIncrement()$ operation executed to set the switch α is k^{i+1}. Thus,

$$AmtSteps(E) \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} k^{i_p} + \frac{r}{k}} + \frac{4(i+2) \cdot n_r}{k^{i+1} + r}$$

We have $k^x \geq x + 1$ for $k \geq e$ and $\forall x \in \mathbf{R}$, it follows:

$$\frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} k^{i_p} + \frac{r}{k}} \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} (i_p+1)}$$

If $i = 0$, and since $r \geq n_r$ we have:

$$\frac{4(i+2) \cdot n_r}{k^{i+1} + r} \leq \frac{8 \cdot n_r}{k+r} \quad \leq 8$$

If $i \geq 1$, because $n_r \leq n$ and $k^{i+1} \geq i \cdot k^2$ we have:

$$\frac{4(i+2) \cdot n_r}{k^{i+1} + r} \leq \frac{4(i+2) \cdot n}{i \cdot k^2 + r}$$

Resulting in an amortized complexity of $O(1)$ for $k \geq \sqrt{n}$. ■

From Lemma III.1, III.5 and III.8 we conclude:

Theorem III.9. *Algorithm 1 is a wait-free linearizable implementation of a k-multiplicative-accurate unbounded counter with a constant amortized complexity for $k \geq \sqrt{n}$.*

D. An Amortized Step Complexity Lower bound for k-multiplicative accurate Counters

In this section, we prove that the amortized step complexity of solo-terminating implementations of k-multiplicative accurate counters is $\Omega(\log_{q+1} \frac{n}{k^2})$ for $k \leq \sqrt{n/2}$, assuming the implementation uses base objects that support only read, write and either reading or regular conditional primitives of arity q or less. A primitive has arity q if it is applied atomically to a vector of q base objects. In all the paper but this section, we consider $q = 1$. Definitions and most of the technical lemmata hold from [16]. In the following we

provide the main idea of the lower bound and the Lemma that most differs from the original work. To this aim, in the following we remember some of the definitions formalized in [16].

Processes communicate with one another by issuing events that apply *read-modify-write* (RMW) primitives to vectors of base objects. We assume that a primitive is always applied to vectors of the same size and that all the base objects to which a primitive is applied are over the same domain. A RMW primitive, applied to a vector of k base objects over some domain D, is characterized by a pair of functions, $\langle g, h\rangle$, where g is the primitive's *update function* and h is the primitive's *response function*. The update function $g : D^k \times W \to D^k$, for some input-values domain W, determines how the primitive updates the values of the base objects to which it is applied.

Let e be an event, issued by process p after execution E, which applies the primitive $\langle g, h\rangle$ to a vector of base objects $\langle o_1, \ldots, o_k\rangle$. Then e atomically does the following: it updates the values of objects $o_1, \ldots, o_k$ to the values of the components of the vector $g(\langle v_1, \ldots, v_k\rangle, w)$, respectively, where $\overrightarrow{v} = \langle v_1, \ldots, v_k\rangle$ is the vector of values of the base objects after E, and $w \in W$ is an input parameter to the primitive. We call $\overrightarrow{v}$ the *object-values vector* of e after E. The RMW primitive returns a response value, $h(\overrightarrow{v}, w)$, to process p. If W is empty, we say that the primitive *takes no input*.

Next, we revise the concept of conditional synchronization primitives.

Definition III.1. *A RMW primitive $\langle g, h\rangle$ is* conditional *if, for every possible input w, $\left|\{\overrightarrow{v} | g(\overrightarrow{v}, w) \neq \overrightarrow{v}\}\right| \leq 1$. Let e be an event that applies the primitive $\langle g, h\rangle$ with input w. The* change point *of e is the unique vector $\overrightarrow{c_w}$ such that $g(\overrightarrow{c_w}, w) \neq \overrightarrow{c_w}$; any other vector is a* fixed point *of e.*

For example, k-CAS is a conditional primitive for any integer $k \geq 1$. The single change point of a k-CAS event with input $\langle old_1, \ldots, old_k, new_1, \ldots, new_k\rangle$ is the vector $\langle old_1, \ldots, old_k\rangle$. Read is also a conditional primitive, since read events have no change points.

A RMW event is invisible if its object-values vector is a fixed point of the event when it is issued. A RMW event e that is applied by process p to an object vector is also invisible if, before p applies another event, a write event is applied to each object o_i that is changed by e before another RMW event is applied to o_i. If a RMW event e is not invisible in an execution E on some object o, we say that *e is visible in E on o*. If e is not invisible in E, we say that e is a *visible* event in E.

Lower bound: The key intuitions behind the following lower bound is that first, in any n-process execution of a k-multiplicative accurate counter implementation, 'many' processes need to be aware of the participation of 'many' other processes in the execution, and second, if processes only use read, write and conditional primitives, then a scheduling adversary can order events so that information about the participation of processes in the computation accumulates 'slowly'.

We formalize the first key intuition by proving Lemma III.10. The rest is straightforward from the lower bound in the original work [16].

The next definition captures the extent to which processes are aware of the participation of other processes in an execution. Intuitively, a process p is aware of the participation of another process q if p reads a shared-memory value that was either directly written by q or indirectly influenced by a value written by q. The following definitions formalize this notion.

Definition III.2. *Let e_q be an event by process q in an execution E, which applies a non-trivial primitive to a vector v of base objects. We say that an event e_p in E by process p is* aware *of e_q if e_p accesses a base object o such that at least one of the following holds:*

- *There is a prefix E' of E such that e_q is visible on o in E' and e_p is a RMW event that applies a primitive other than write to o, and it follows e_q in E', or*
- *there is an event e_r that is aware of e_q in E and e_p is aware of e_r in E.*

If an event e_p of process p is aware of an event e_q of process q in E, we say that p is aware *of e_q and that e_p is* aware *of q in E.*

The following definition quantifies the extent to which a process is aware of the participation of other processes in an execution.

Definition III.3. *Process p is* aware *of process q after an execution E if either $p = q$ or p is aware of an event of q in E. The* awareness set *of p after E, denoted $AW(E, p)$, is the set of processes that p is aware of after E.*

The following lemma proves a relation between the value returned by a *CounterRead* operation instance of a process in some execution and the size of that process' awareness set after that execution. This is our main technical contribution to the lower bound.

Lemma III.10. *Let E be an execution of a solo-terminating k-multiplicative accurate counter object implementation where each process executes one instance of the $CounterIncrement()$ operation followed by one instance of the $CounterRead()$ operation. If the $CounterRead()$ instance by a process p returns i in E then $|AW(E, p)| \geq \frac{i}{k}$.*

Proof: Assume, by way of contradiction, that there is an execution E where each process executes one instance of the $CounterIncrement()$ operation followed by one instance of the $CounterRead()$ operation, and a process p such that

a $CounterRead()$ instance by p, namely op, returns i and $|AW(E,p)| < \frac{i}{k}$.

We construct a new execution E' as follows: for any process $q \notin AW(E,p)$, we first remove all the events of q from E; then, for any process q', we remove all the events of q' that are aware of q. Note that if an event $e_{q'}$ of q' is aware of q, then all following events by q' are also aware of q and are removed. Also, no events of p are removed since p is aware only of processes in $AW(E,p)$.

We prove that E' is an execution, and that it is indistinguishable from E. We consider events in the order they appear in E'. Let e'_q be an event by process q' that appears in E', namely $E' = E'_1 e'_q E'_2$. Since e'_q is also in E, we can also write $E = E_1 e'_q E_2$. For the induction, assume that E'_1 is an execution and that it is indistinguishable to every process that appears in it from E_1. In particular, q' does not distinguish between E'_1 and E_1 and takes the same step after both of them. To see why q' obtains the same response in e'_q after E'_1 and after E_1, note that it can return a different response only if in E, e'_q is aware of an event e that was removed from E_1. This happens only if e is aware of some process $q \notin AW(E,p)$, meaning that in E, e'_q is also aware of q, contradicting the fact that e'_q was not removed. Hence $E'_1 e'_q$ is an execution and q' does not distinguish between $E'_1 e'_q$ and $E_1 e'_q$.

This implies that the $CounterRead()$ instance by p returns i also in E'; on the other hand, less than $\frac{i}{k}$ processes participate in E'. Let E" be the extension of E' in which the processes that participate in E' complete their operation instances, one at a time. This execution exists by solo-termination, and results in a quiescent execution. However, less than $\frac{i}{k}$ instances of $CounterIncrement()$ operations completed in E", and we have that p returns i when invoking op. Thus, the response of the op is not linearizable. In particular, consider any linearization L of E" and let v be the number of $CounterIncrement()$ instances linearized before op in L, we have that $\frac{v}{k} \leq i \leq k \cdot v < k \cdot \frac{i}{k} = i$. ■

Corollary III.10.1. *Let E be a quiescent n-process execution of a solo-terminating k-multiplicative counter implementation, where each process executes one instance of the $CounterIncrement()$ operation followed by one instance of a $CounterRead()$ operation. Then, the awareness sets of $\frac{n}{2}$ processes contain at least $\frac{n}{2k^2}$ other processes after E.*

Proof: Let L denote any linearization of E, and let op be the i-th $CounterRead()$ instance in L. Since op is the i-th instance of $CounterRead()$ in L, it returns v such that $v \geq \frac{i}{k}$. By considering the last $\frac{n}{2}$ processes linearized and by Lemma III.10, the claim follows. ■

Information about processes that participate in an execution is transferred through base objects. To complete the proof, we need to show that each of the $\frac{n}{2}$ processes has to apply $\Omega(\log_{q+1}(n/k^2))$ events to build its awareness set (proved in the Appendix).

The following theorem formalize our step complexity lower bound.

Theorem III.11. *Let A be an n-process solo-terminating implementation of a k-multiplicative counter from base objects that support only read, write and either reading or regular conditional primitives of arity q or less. Then A has an execution E that contains $\Omega(n \log_{q+1}(n/k^2))$ events for $k \leq \sqrt{n/2}$, in which every process performs a single $CounterIncrement()$ instance and a single $CounterRead()$ instance.*

IV. A k-MULTIPLICATIVE-ACCURATE BOUNDED MAX REGISTER

In this section, we present an implementation of a k-multiplicative-accurate max register. The algorithm is wait-free, and has asymptotically optimal worst-case step complexity. Indeed, we present in the next section a matching lower bound.

The key idea of our algorithm is to consider the k-base representation of values written to the register and have `Write` operations store only the index of the bit preceding (i.e., to the left of) the most significant bit (MSB) of their arguments. These indices are stored in an (accurate) $\big((\lfloor \log_k(m-1) \rfloor)+1\big)$-bounded max register implemented in a wait-free manner [8]. A $Read$ operation R reads the value p of the accurate max register. If it equals 0 (implying that it was not written to yet), R returns 0. Otherwise, p is the largest index written so far to the accurate max register and R returns k^p. The pseudocode is presented by Algorithm 2.

Algorithm 2: A k-multiplicative-accurate m-bounded max register

```
Shared variables M:
    ((⌊log_k(m − 1)⌋) + 1)-bounded max register
    initially 0
Function Read()
    p ⟵ M.read()
    if p=0 then return 0;
    else return k^p;
end
Function Write(v)
    p ⟵ ⌊log_k v⌋ + 1;
    M.write(p);
end
```

We now prove that Algorithm 2 is a correct wait-free implementation of a k-multiplicative-accurate max register.

Observation 1. *Algorithm 2 is a wait-free implementation of a k-multiplicative-accurate m-bounded max register.*

Proof: Follows directly from the wait-freedom of the max register algorithm of [8]. ■

Lemma IV.1. *Algorithm 2 is a linearizable implementation of a k-multiplicative-accurate m-bounded max register.*

Proof: Let M_m^k denote a k-multiplicative-accurate m-bounded max register implemented by Algorithm 2 and let

E be an execution of M_m^k. We now specify how operation instances on M_m^k in E are linearized. First, all the instances of `Read` that did not execute line 3 in E and all the instances of `Write` operations did not execute line 9 in E do not appear in the linearization. We say these are *removed operations*. Note that none of the removed operations has completed in E. For all remaining instances, we define the linearization point of a `Read` operation on M_m^k to be the linearization point of the $read$ operation it invoked on M in E (in line 3) and the linearization point of a `Write` operation on M_m^k as the linearization point of the $write$ operation it invokes on M (in line 9). Since each non-removed operation instance on M_m^k in E is linearized at a step it performs (hence during its execution interval), the linearization order we have define, denoted by L, respects the real-time order of the operation instances in E.

It remains to show that L satisfies the sequential specification of a k-multiplicative-accurate m-bounded max register. First note that since values written to M_m^k are from $\{1, ..., m-1\}$ and from lines 8-9, only values from $\{1, ..., \lfloor \log_k(m-1) \rfloor + 1\}$ are written to M. Let R denote a `Read` instance in L that returns 0 in line 4. Since only positive values are ever written to M, it follows that R is not preceded in L by any `Write` instance, hence the value of M_m^k when R is linearized is its initial value 0, so R returns the exact value of M_m^k.

Assume, then, that R *is* preceded in L by one or more `Write` instances and returns a positive value $x = k^p$ for some $p \geq 1$. We need to prove that $v/k \leq x \leq vk$ holds, where v is the maximum value written by any `Write()` instance linearized before R in L. Since M is linearizable and since we have linearized all non-removed instances applied to M_m^k in E according the order of the operations they applied to M (in line 3 or in line 9), there exists a `Write` operation that writes some value w and appears before R in L, such that $\lfloor \log_k w \rfloor = p - 1$ and p is the maximum value written to M by any `Write` instance that precedes R in L. Let $V = \{w \big| \lfloor log_k(w) \rfloor = p - 1\}$ be the set of all the values written to M_m^k in L before R whose MSB equals $p-1$. Let $v = max(V)$. It follows that v is the maximum value written to M_m^k by any `Write()` instance linearized in L before R. We have $v \in [k^{p-1}, k^p - 1]$ and $x = k^p$. Consequently, $v \leq x \leq v \cdot k$ and the sequential specification of the k-multiplicative m-bounded max register is satisfied. ■

Theorem IV.2. *Algorithm 2 is a wait-free linearizable implementation of a k-multiplicative-accurate m-bounded max register with worst case operation step complexity* $O\Big(min\big(\log_2(\log_k m), n\big)\Big)$.

Proof: Wait-freedom and linearizability follow from Observation 1 and Lemma IV.1, respectively. As for step complexity – the worst case operation step complexity of the wait-free implementation of an m-bounded max register of [8] is $O\big(min(\log m, n)\big)$ for both $Read$ and $Write$ operations. Each operation of Algorithm 2 applies a single operation on a $\big((\lfloor \log_k(m-1) \rfloor) + 1\big)$-bounded max register and a constant number of additional steps. The theorem follows. ■

V. A Lower bound on the Worst Case Step Complexity of k-multiplicative m-bounded Max Register and Counter

Aspnes et al. [5] proved a worst-case step complexity on the lower bound of a class of concurrent objects called *L-perturbable*, that includes objects such as max registers, counters and snapshots. L is called the *perturbation bound*. Roughly speaking, an object is L-perturbable if, for every implementation of the object, there exists an operation Op and an execution E, in the course of which Op is "perturbed" L times. An outstanding operation Op by process p is said to be perturbed by a process q, if a solo execution by q can change the response of a solo execution by p. They prove [5, Theorem 1] that any obstruction-free implementation of an L-perturbable object O from *historyless* primitives has an execution in which some process accesses $\Omega\big(min(\log_2 L, n)\big)$ distinct base objects during a single operation instance. Specifically, this implies that the worst-case step complexity of such implementations is $\Omega\big(min(\log_2 L, n)\big)$.

For the sake of presentation completeness, we restate the definition of an L-perturbable object from [5].

[5], Definition 2. *Let $\mathcal{I}$ be an obstruction-free implementation of an object. The set S_k of k-perturbing executions with respect to an operation instance op_n by process p_n is defined inductively as follows:*

1) *S_0 is the singleton set containing the empty sequence.*
2) *If $\alpha_{k-1}\lambda_{k-1}$ is in S_{k-1}, where λ_{k-1} consists of $n-1$ events, one by each of the processes $p_1, \ldots, p_{n-1}$, then $\alpha_{k-1}\lambda_{k-1}$ is in S_k. In this case, we say that $\alpha_{k-1}\lambda_{k-1}$ is* saturated.
3) *Suppose $\alpha_{k-1}\lambda_{k-1}$ is in S_{k-1}, no process has more than one event in λ_{k-1}, and there is a sequence γ of events by a process p_l different from p_n and the processes that have events in λ_{k-1}, such that the sequences of events by p_n as it performs op_n after $\alpha_{k-1}\lambda_{k-1}$ and $\alpha_{k-1}\gamma\lambda_{k-1}$ differ. Let $\gamma = \gamma' e \gamma''$, where e is the first event of γ such that the sequences of events taken by p_n as it performs op_n by itself after $\alpha_{k-1}\lambda_{k-1}$ and after $\alpha_{k-1}\gamma' e \lambda_{k-1}$ differ. Let λ be some permutation of the event e together with the events in λ_{k-1}, and let λ', λ'' be any two sequences of events such that $\lambda = \lambda'\lambda''$. Then the execution $\alpha_k\lambda_k$ is in S_k, where $\alpha_k = \alpha_{k-1}\gamma'\lambda'$ and $\lambda_k = \lambda''$.*

[5], Definition 3. *An obstruction-free implementation of an object is L-perturbable if there is an operation instance op_n*

such that the set S_L of L-perturbing executions with respect to op_n by p_n is nonempty.

An object $\mathcal{O}$ is *perturbable* if all its obstruction-free implementations are perturbable.

[5], Theorem 1. *Let A be an n-process obstruction-free implementation of an L-perturbable object $\mathcal{O}$ from historyless primitives. Then A has an execution in which some process accesses $\Omega(\min(\log_2 L, n))$ distinct base objects during a single operation instance.*

Lemma V.1. *A k-multiplicative-accurate m-bounded max register is $\Theta(log_k m)$-perturbable for $k > 1$.*

Proof: Let O be a k-multiplicative-accurate m-bounded max register and consider an obstruction-free implementation of O. We show that O is $(\frac{1}{2}log_k(m-1))$-perturbable for a `Read()` operation instance op_n by process p_n. We proceed by induction where the base case for $r = 0$ is immediate. Let $r < \frac{1}{2}log_k(m-1)$ and let $\alpha_{r-1}\lambda_{r-1}$ be an $(r-1)$-perturbing execution of O. If $\alpha_{r-1}\lambda_{r-1}$ is saturated, then it is also an r-perturbing execution. Otherwise, denote by v_{r-1} the maximum input to the `write()` operations linearized before op_n in the execution sequence $\alpha_{r-1}\lambda_{r-1}$. Since $\alpha_{r-1}\lambda_{r-1}$ is not saturated, there exists a process $p_l \neq p_n$ that does not take steps in λ_{r-1}. Let γ be the execution fragment by p_l where it finishes any incomplete operation in α and then performs a `write()` operation to the max register with the value $v_r = k^2 v_{r-1} + 1$. Then op_n must return a value x such that $kv_{r-1} < v_r/k \leq x \leq kv_r$ when run after $\alpha_{r-1}\gamma\lambda_{r-1}$. It follows that an r-perturbing execution can be constructed from $\alpha_{r-1}\lambda_{r-1}$ and γ as specified by [5], Definition 2. Because O is an m-bounded max register, during the rth step of the induction, the value written to the max register must satisfy $v_r \leq m-1$. Consequently it suffices to have:

$$v_r \leq (k+1)^{2r} \leq m-1 \implies r \leq \frac{1}{2}log_{k+1}(m-1) = \Theta(log_k m)$$

■

from Lemma V.1 and [5], Theorem 1 we have the following theorem:

Theorem V.2. *The worst-case step complexity of a k-multiplicative m-bounded max register is $\Omega\big(min(\log_2(\log_k m), n)\big)$*

Lemma V.3. *A k-multiplicative-accurate m-bounded counter is $\Theta(log_k(m))$-perturbable for $k > 1$.*

Proof: Let O be a k-multiplicative m-bounded counter and consider an obstruction-free implementation of O. We show that O is $(\frac{1}{2}log_k(m-1))$-perturbable for a $CounterRead()$ operation instance op_n by the process p_n. We proceed by induction where the base case for $r = 0$ is immediate. Let $\alpha_{r-1}\lambda_{r-1}$ be an $(r-1)$-perturbing execution of O. If $\alpha_{r-1}\lambda_{r-1}$ is saturated, then it is also an r-perturbing execution. Otherwise, let I_r denote the number of $CounterIncrement()$ operation instances performed by the perturbing process in iteration r. We have that $I_1 = 1$ in order for op_n to return a value greater than 0. For $r > 1$, if op_n runs after $a_{r-1}\lambda_{r-1}$ it can return a value that is as large as $k \cdot \sum_{j=1}^{r-1} I_j$. Therefore, we need the number of complete $CounterIncrement()$ operation instances after $a_{r-1}\gamma\lambda_{r-1}$ to be at least $k^2 \cdot \sum_{j=1}^{r-1} I_j + 1$ for op_n to return a value greater than $k \cdot \sum_{j=1}^{r-1} I_j$.

Besides the $CounterIncrement()$ operation instances in γ, at least $\sum_{j=1}^{r-1} I_j - (r-1)$ have finished, therefore setting $I_r = (k^2-1) \cdot \sum_{j=1}^{r-1} I_j + r$ implies that op_n returns at least $\frac{1}{k}(\sum_{j=1}^{r-1} I_j - (r-1) + I_r) = \frac{1}{k}(\sum_{j=1}^{r-1} I_j - (r-1) + (k^2-1) \cdot \sum_{j=1}^{r-1} I_j + r) = \frac{1}{k}(k^2 \cdot \sum_{j=1}^{r-1} I_j + 1)$ which is greater tha $k \cdot \sum_{j=1}^{r-1} I_j$ as needed.

$$\begin{aligned} I_r &= \sum_{i=0}^{r-1}(r-i)(k^2-1)^i = \sum_{i=1}^{r} i \cdot (k^2-1)^{r-i} \\ &= (k^2-1)^r \sum_{i=1}^{r} \frac{i}{(k^2-1)^i} \\ &= \frac{(k^2-1)((k^2-1)^r - 1) + r(2-k^2)}{(k^2-2)^2} \leq k^{2r} \leq m \\ &\implies r \leq \frac{1}{2}log_k(m) = \Theta(log_k m) \end{aligned}$$

■

From Lemma V.3 and [5], Theorem 1, we prove the following Theorem

Theorem V.4. *The worst-case step complexity of a k-multiplicative m-bounded counter is $\Omega\big(min(\log_2(log_k m), n)\big)$*

VI. Discussion

We have presented upper and lower bounds on the step complexity of a variant of deterministic approximate counters and max-registers. We have proved the possibly counter-intuitive[1] result that when the accuracy parameter k does not depend on n, relaxing counter semantics by allowing inaccuracy of a multiplicative factor cannot asymptotically reduce the step complexity of unbounded counters by more than a logarithmic factor. Then, we present a wait-free linearizable k-multiplicative-accurate counter for $k \geq \sqrt{n}$ with constant amortized step complexity.

The maximum improvement in the worst case step complexity of the bounded variant of k-multiplicative-accurate counters remains an open question. Also, when k is constant, it is unclear whether there exists a deterministic wait-free k-multiplicative-accurate counter implementation with $o(\log^2 n)$ amortized step complexity.

[1]Pun unintended.

We also show that relaxing the semantics of max registers by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity of the bounded variant and in the amortized step complexity of the unbounded one. Overall we provide theoretical evidence that worst-case time complexity does not indicate benefit from some common relaxation of unbounded counters and max-register while average complexity does. It is interesting to note that a similar result has been proved for relaxed queues in the message passing systems by Talmage and Welch [19].

ACKNOWLEDGMENT

Alessia Milani and Corentin Travers are supported by ANR projects Descartes and FREDDA. Adnane Khattabi is supported by UMI Relax. Danny Hendler is supported in part by ISF grant 380/18.

REFERENCES

[1] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *14th International Conference on Principles of Distributed Systems (OPODIS)*, ser. Lecture Notes in Computer Science, vol. 6490. Springer, 2010, pp. 395–410.

[2] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2013, pp. 317–328.

[3] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[4] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," in *33rd International Symposium on Distributed Computing, DISC*, ser. LIPIcs, vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 31:1–31:15.

[5] J. Aspnes, K. Censor-Hillel, H. Attiya, and D. Hendler, "Lower bounds for restricted-use objects," *SIAM J. Comput.*, vol. 45, no. 3, pp. 734–762, 2016.

[6] P. Jayanti, K. Tan, and S. Toueg, "Time and space lower bounds for nonblocking implementations," *SIAM J. Comput.*, vol. 30, no. 2, pp. 438–456, Apr. 2000.

[7] H. Attiya and A. Fouren, "Adaptive and efficient algorithms for lattice agreement and renaming," *SIAM Journal on Computing*, vol. 31, no. 2, pp. 642–664, 2001.

[8] J. Aspnes, H. Attiya, and K. Censor-Hillel, "Polylogarithmic concurrent data structures from monotone circuits," *J. ACM*, vol. 59, no. 1, pp. 2:1–2:24, 2012.

[9] M. A. Baig, D. Hendler, A. Milani, and C. Travers, "Long-lived counters with polylogarithmic amortized step complexity," in *33rd International Symposium on Distributed Computing, (DISC)*, ser. LIPIcs, vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 3:1–3:16.

[10] D. Dice, Y. Lev, and M. Moir, "Scalable statistics counters," in *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13*. ACM, 2013, pp. 43–52.

[11] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, "Cbtree: A practical concurrent self-adjusting search tree," in *Distributed Computing - 26th International Symposium, DISC. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7611. Springer, 2012, pp. 1–15.

[12] R. H. M. Sr., "Counting large numbers of events in small registers," *Commun. ACM*, vol. 21, no. 10, pp. 840–842, 1978.

[13] P. Flajolet, "Approximate counting: A detailed analysis," *BIT Comput. Sci. Sect.*, vol. 25, no. 1, pp. 113–134, 1985.

[14] J. Aspnes and K. Censor, "Approximate shared-memory counting despite a strong adversary," *ACM Trans. Algorithms*, vol. 6, no. 2, pp. 25:1–25:23, 2010.

[15] M. A. Bender and S. Gilbert, "Mutual exclusion with o(logˆ2 log n) amortized work," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, 2011, pp. 728–737.

[16] H. Attiya and D. Hendler, "Time and space lower bounds for implementations using k-cas," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 2, pp. 162–173, 2010.

[17] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[18] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *null*. IEEE, 2003, p. 522.

[19] E. Talmage and J. L. Welch, "Improving average performance by relaxing distributed data structures," in *28th International Symposium on Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 8784. Springer, 2014, pp. 421–438.