The Space Complexity of Scannable Objects with Bounded Components

Sean Ovens
University of Toronto, Canada

Abstract

A fundamental task in the asynchronous shared memory model is obtaining a consistent view of a collection of shared objects while they are being modified concurrently by other processes. A scannable object addresses this problem. A scannable object is a sequence of readable objects called components, each of which can be accessed independently. It also supports the *Scan* operation, which simultaneously reads all of the components of the object. In this paper, we consider the space complexity of an n-process, k-component scannable object implementation from objects with bounded domain sizes. If the value of each component can change only a finite number of times, then there is a simple lock-free implementation from k objects. However, more objects are needed if each component is fully reusable, i.e. for every pair of values v, v', there is a sequence of operations that changes the value of the component from v to v'.

We considered the special case of scannable binary objects, where each component has domain $\{0, 1\}$, in PODC 2021. Here, we present upper and lower bounds on the space complexity of any n-process implementation of a scannable object O with k fully reusable components from an arbitrary set of objects with bounded domain sizes. We construct a lock-free implementation from k objects of the same types as the components of O along with $\lceil \frac{n}{b} \rceil$ objects with domain size 2^b. By weakening the progress condition to obstruction-freedom, we construct an implementation from k objects of the same types as the components of O along with $\lceil \frac{n}{b-1} \rceil$ objects with domain size b.

When the domain size of each component and each object used to implement O is equal to b and $n \leq b^k - bk + k$, we prove that $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ objects are required. This asymptotically matches our obstruction-free upper bound. When $n > b^k - bk + k$, we prove that $\frac{1}{2} \cdot (b^{k-1} - \frac{(b-1)k+1}{b})$ objects are required. We also present a lower bound on the number of objects needed when the domain sizes of the components and the objects used by the implementation are arbitrary and finite.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases space complexity, lower bound, shared memory, snapshot object

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.30

Funding Support is gratefully acknowledged from the Natural Sciences and Engineering Research Council of Canada under grant RGPIN-2020-04178 and the Ontario Graduate Scholarship (OGS) Program.

Acknowledgements I thank my advisor, Faith Ellen, for the many helpful discussions and proofreading throughout this project. I also thank the anonymous reviewers for their comments.

1 Introduction

A *scannable object* O consists of a sequence of objects $O[1], \ldots, O[k]$ called *components*, each of which stores a value from some domain and supports *Read* along with some other operations. The $Apply(i, op)$ operation applies the operation op to $O[i]$, where op is an operation supported by $O[i]$. A scannable object also supports the *Scan* operation, which returns a consistent view of $O[1], \ldots, O[k]$ at a point during the operation's execution interval.

A *snapshot object* [1, 2, 4] is a scannable object whose components support the *Read* and *Write* operations. Snapshot objects were formalized independently by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [1], Anderson [2], and Aspnes and Herlihy [4]. They have been used to simplify the description of obstruction-free consensus algorithms [10], approximate agreement algorithms [8], and implementations of large classes of objects [4, 16].

It is known that a k-component snapshot object implemented from read/write registers requires at least k registers [13]. There are many known implementations that match this lower bound, but all of them either use objects that are large enough to store the result of a *Scan* [1, 2, 17, 19, 20] or use unbounded sequence numbers [1, 9, 14, 15]. There are also implementations that use significantly more than k base objects [3, 18, 7, 25]. Other implementations use unbounded version lists [24]. Prior to our work, it was not well understood how the number of base objects required to implement a scannable object is related to the domain sizes of the base objects and the components. In this paper, we investigate the space complexity of scannable objects with bounded components that are implemented from objects with bounded domains.

Last year, we considered the space complexity of *scannable binary objects* (i.e. scannable objects whose components have domain $\{0, 1\}$) implemented from objects with domain $\{0, 1\}$. In some circumstances, it is possible to implement a scannable binary object from only k objects. For example, consider a scannable binary object O whose components are test-and-set (TAS) objects. A TAS object supports *Read* and *TAS*, which changes the value of the object to 1 and returns its previous value. There is a simple wait-free implementation of O from k TAS objects $T_1, \ldots, T_k$. The object T_i stores the value of component $O[i]$, an *Apply*(i, *Read*) operation reads T_i, and an *Apply*(i, *TAS*) operation applies *TAS* to T_i. A *Scan* repeatedly collects the values in $T_1, \ldots, T_k$ (i.e. reads them one at a time) until it observes the same sequence of values twice in a row. When this happens, the *Scan* returns this sequence of values. Since the value of each component can change at most once, a *Scan* operation will terminate after performing at most $k + 2$ collects. The implementation is correct because the value of a component cannot change from v to a different value v' and then back to v. Hence, the sequence of values returned by the *Scan* must be the actual value of the scannable object at some point during the execution interval of the *Scan*.

More generally, we gave a lock-free, n-process implementation of any k-component scannable binary object from k objects with the same types as the components of the object along with n binary registers. If the components of the scannable binary object are non-monotonic (i.e. their value can be changed from 0 to 1 and from 1 to 0), we show that more than k objects are required. Specifically, any obstruction-free, n-process implementation of a scannable binary object with k non-monotonic components requires at least $n + k - r - 2$ objects with domain $\{0, 1\}$, where $k \geq 2$ and $2^k - 2^{k-r} < n - 2 \leq 2^k - 2^{k-r-1}$. This lower bound applies to single-updater implementations, where only a single process (called the updater) is allowed to change the value of any component. Since the lower bound applies to obstruction-free and single-updater implementations, it applies to lock-free and wait-free implementations that support multiple updaters as well.

In this paper, we generalize our previous results significantly to obtain new upper and lower bounds on the space complexity of scannable object implementations from objects with arbitrary bounded domain sizes. As discussed with scannable binary objects, a k-component scannable object has a wait-free implementation from k objects with the same domain sizes as the components if each component's value can change only a finite number of times. For example, consider a k-component scannable object O consisting of b-bounded counters. A b-bounded counter supports *Read* and Inc_b, which increases the value of the counter by 1

if its current value is less than $b-1$ and does nothing otherwise. The scannable object O can be implemented from k b-bounded counters, each of which stores the value of one of the components. A *Scan* repeatedly collects the values of the objects until it obtains the same sequence of values twice in a row. Since the value of each component can increase at most $b-1$ times, a *Scan* will terminate after performing at most $k(b-1)+2$ collects.

We show how our lock-free, n-process implementation of any k-component scannable binary object from $n+k$ objects can be generalized to obtain a lock-free, n-process implementation of a k-component scannable object from k objects with the same domain sizes as the components of the scannable object along with $\lceil \frac{n}{b} \rceil$ objects that have domain size 2^b, for any $b \geq 1$. We also construct an obstruction-free implementation from k objects with the same domain sizes as the components of the scannable object along with $\lceil \frac{n}{b-1} \rceil$ read/write registers that have domain size b. We generalize the notion of non-monotonic binary objects to objects with larger domain sizes: An object is *fully reusable* if, for every pair of values v, v' in its domain, there is a sequence of operations that changes its value from v to v'. This is a natural condition that includes many common objects like registers, compare-and-swap objects, and modulo-b counters. A b-bounded counter is not fully reusable, since there is no sequence of operations that changes its value from 1 to 0, for example.

We show that any obstruction-free, n-process implementation of a scannable object with k fully reusable components that have domain size b requires at least $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ objects with domain size b when $n \leq b^k - bk + k$. When $n > b^k - bk + k$, we show that $\frac{1}{2} \cdot \left(b^{k-1} - \frac{(b-1)k+1}{b}\right)$ objects with domain size b are required. We also prove a lower bound on the number of objects required by any obstruction-free, n-process implementation of a scannable object with k fully reusable components when the domain sizes of the components and the objects used by the implementation are arbitrary, finite values. Just like our lower bound for scannable binary objects, our lower bound in this paper applies to obstruction-free, single-updater implementations, so it applies to lock-free and wait-free implementations that support multiple updaters as well.

Our original lower bound proof for scannable binary objects involves inductively constructing an unordered set of k-component binary vectors $\{V_1, \ldots, V_\ell\}$ and a configuration C_ℓ, for all $\ell \leq min(n-2, 2^{k-1})$, that satisfy the following property: For any execution α from C_ℓ that does not involve the last ℓ scanners, if the scannable binary object does not contain any of the vectors $V_1, \ldots, V_\ell$ during α, then there is a set of ℓ objects that do not change during α. We show how to obtain an $(n-\ell)$-process implementation of a $(k-1)$-component scannable binary object by discarding these ℓ objects. This can be applied repeatedly until we have a 2-process implementation of a scannable binary object with $k' < k$ components, which we show requires at least $k'+1$ objects.

Our technique in this paper builds on these ideas. However, having objects with domain sizes larger than 2 presents several challenges. First, it is not possible to show that some objects stop changing entirely in certain executions from C_ℓ. Instead, we show that there is a set of forbidden values for each object in certain executions from C_ℓ. Second, since it is not possible to obtain a set of objects that stop changing, a new implementation cannot be obtained by discarding objects. Finally, in this paper we need to construct a sequence of vectors $\langle V_1, \ldots, V_\ell \rangle$ rather than a set of vectors. We will explain how our technique differs from our scannable binary object lower bound in more detail in Section 5.

In Section 2, we briefly survey some other related work. We define our model of computation in Section 3. In Section 4, we present our implementations of scannable objects from objects with bounded domain sizes. We prove our space complexity lower bound in Section 5. Finally, we discuss some possible future research directions in Section 6.

2 Related work

Scannable objects, like snapshot objects, have been used to simplify the description of many distributed algorithms and implementations. Aspnes, Attiya, Censor-Hillel, and Ellen [3] described an implementation of a 2-*component max array*, which is a scannable object that consists of 2 max registers. A *max register* supports *MaxWrite*(x), which changes the value of the object to x if and only if the current value of the max register is less than x, and *MaxRead*, which returns the current value of the max register. The authors used a 2-component max array to implement a limited-use snapshot object whose *Scan* and *Update* operations both have polylogarithmic step complexity. Ellen, Gelashvili, Shavit, and Zhu [14] classified some objects by the number of instances required to solve obstruction-free consensus. In certain cases, they showed that it is possible to solve obstruction-free consensus using a scannable object. For example, obstruction-free consensus can be solved among n processes using a scannable object with $n-1$ components that each support *Read* and *Swap*(v), which changes the value of the component to v and returns its previous value.

Consider a k-component scannable object O whose components have domains $D_1, \ldots, D_k$. Then O has a wait-free, single-updater implementation from one single-writer register with domain $D_1 \times \ldots \times D_k$. A *Scan* simply reads the register. The updater locally stores the current value of O in a variable V. When the updater performs an *Apply*(i, op) operation, it locally applies op to V and then writes the resulting vector to the register. There is a known wait-free implementation of a single-writer register with any finite domain size d from d single-writer binary registers [23]. Hence, O has a wait-free, single-updater implementation from $\prod_{i=1}^{k} |D_i|$ single-writer binary registers. Chen and Wei [12] gave an implementation of an s-bit single-writer register from $\Theta(\frac{ns}{t})$ instances of t-bit single-writer registers.

Jayanti [20] defined a generalization of a scannable object called an f*-array*, where f is a function. Like a scannable object, an f-array consists of a sequence of components, each of which has its own domain. The domain of an f-array is the cross product of the domains of its components. The function f maps the domain of the f-array to some arbitrary set of values. An f-array supports a generalization of *Scan*, which we call f*-Scan*, that returns the result of applying f to the value of the object. When f is the identity function, f*-Scan* is the same as *Scan*. Jayanti gives a wait-free implementation of a k-component f-array from k objects of the same types as the components of the f-array along with a single LL/SC object large enough to store the result of an f*-Scan*.

Wei, Ben-David, Blelloch, Fatourou, Ruppert, and Sun [24] described an approach for implementing a scannable object whose components are compare-and-swap objects. Their approach uses a versioned compare-and-swap object to store the value of a component. A versioned compare-and-swap object also stores an unbounded version list, which has a complete history of all the successful *CAS* operations applied to the object. Each element of the version list also stores a timestamp. To *Scan* the scannable object, a process first obtains a new timestamp ts and then traverses the version list of each component to find the value with the latest timestamp that does not exceed ts.

Ellen, Fatourou, and Ruppert [13] proved that, for all $n > k$, an n-process, k-component snapshot implementation requires at least k registers. Jayanti, Tan, and Toueg [21] proved that $n-1$ registers are required to implement a snapshot object with n components, where each component can be modified by only a single process. Both of these lower bound proofs used covering arguments, which were originally introduced by Burns and Lynch [11].

Covering arguments are a standard technique for proving space complexity lower bounds for implementations that use historyless objects. A *historyless* object can support two kinds of operations: *trivial* operations never change the value of the object, and *historyless* operations

set the object to some fixed value that does not depend on the old value of the object. A set of processes $\mathcal{P}$ *covers* a set of historyless objects $\mathcal{B}$ if $|\mathcal{P}| = |\mathcal{B}|$ and, for every $B \in \mathcal{B}$, there is a process in $\mathcal{P}$ that is poised to apply a nontrivial, historyless operation to B in its next step. If each of the processes in $\mathcal{P}$ takes its next step, then the information in $\mathcal{B}$ is overwritten. In order to prove a lower bound of m on the space complexity of an implementation, it suffices to construct a configuration of the implementation in which a set of processes $\mathcal{P}$ covers a set of objects $\mathcal{B}$ with $|\mathcal{B}| = m$. Therefore, the best space lower bound that can be obtained by a covering argument is n, the number of processes. Hence, to obtain our lower bound, we need to use different techniques. Furthermore, our lower bound applies to implementations that use non-historyless objects.

3 Model

We use a standard asynchronous shared memory model in which n processes communicate using shared *objects*. An object has a *domain* of possible *values*, a set of *invocations* that can be applied to it, and a set of possible *responses* to each invocation. The *sequential specification* of an object O defines, for each value v and each invocation Inv of O, the resulting value of O and the response to Inv when Inv is applied to O.

An object O is *fully reusable* if, for all distinct values v, v' of O, there is a sequence of invocations on O that changes its value from v to v'. An example of a fully reusable object with domain $\{0, \ldots, b-1\}$ supports a single invocation that returns its current value x and then changes its value to $x + 1 \mod b$. An example of an object with the same domain that is not fully reusable supports a single invocation that returns its current value x and then changes its value to $min(b-1, x+1)$. For example, there is no sequence of invocations that would change this object's value from 1 to 0.

In this paper, we implement new objects from a set of *base objects*, which are provided by the system. An *implementation* of an object defines a set of base objects and an algorithm for each process to follow for every invocation of the object. For the sake of clarity, we call the invocations of base objects *primitives*, and we call the invocations of implemented objects *operations*.

A *configuration* of an implementation consists of a value for every base object and a state for every process. We use $value(B, C)$ to denote the value of base object B in configuration C. A *step* by a process consists of a primitive applied to some base object and a response to that primitive, followed by a finite amount of local computation by that process, which may then change its state. In an *initial configuration* of an implementation, no processes have taken steps. An *execution* is an alternating sequence of configurations and steps that begins with a configuration. If an execution is finite, then it ends with a configuration. If C is a configuration and α is a finite execution starting with C, then $C\alpha$ denotes the final configuration of α. If an execution α only contains steps by processes in some set P, then we say α is P-only. If P contains exactly one process p_j, then we say α is p_j-only.

Executions are produced by a *scheduler* that decides the order in which processes take steps and the operations on the implemented object that they perform. In every initial configuration of an implementation, every process is idle. When an idle process p_i is chosen to take a step by the scheduler, the scheduler specifies an instance of an operation to p_i, and then the process takes the first step of its algorithm for that operation. When p_i's algorithm terminates, it returns a response to this operation instance, which is now *complete*, and then p_i becomes idle again. An instance of an operation by p_i is *ongoing* in any configuration that occurs after it is given the operation instance and before p_i returns its response. When

the scheduler chooses a process that is not idle, the process only performs the next step of its algorithm. Thus, a process can have at most one ongoing operation in any configuration. When a process takes a step, the response to the primitive that it applies is determined by the value of the base object to which the primitive is applied along with the sequential specification of that base object. A configuration C is called P-idle, where P is a set of processes, if all of the processes in P are idle in C. If $P = \{p_i\}$, then we say C is p_i-idle.

Two configurations C_1, C_2 are *indistinguishable* to a set of processes P if every process in P has the same state in C_1 and C_2. We use $C_1 \overset{P}{\sim} C_2$ to denote this. Suppose that α_1, α_2 are executions beginning with C_1, C_2, respectively. Then α_1 and α_2 are *indistinguishable* to a set of processes P if $C_1 \overset{P}{\sim} C_2$ and every process in P performs the same sequence of steps (and receives the same responses to each of the primitives applied) in α_1 and α_2. We use $\alpha_1 \overset{P}{\sim} \alpha_2$ to denote this. If γ_1 is a P-only execution from C_1, the base objects accessed by P during γ_1 have the same values in C_1 and C_2, and $C_1 \overset{P}{\sim} C_2$, then there is a P-only execution γ_2 from C_2 such that $\gamma_1 \overset{P}{\sim} \gamma_2$ [6].

An object is *readable* if it supports the *Read* invocation, which returns the value of the object. A *scannable object* O is a sequence of readable objects called *components*. We use $O[i]$ to denote the i-th component of the scannable object O. The value of a k-component scannable object O is a vector in $D_1 \times \ldots \times D_k$, where D_i is the domain of component $O[i]$. The object O supports the invocation $Apply(i, op)$, which applies the invocation op to $O[i]$. The object O also supports *Scan*, which reads every component of the object simultaneously.

A *single-updater* implementation of a scannable object allows only one process, called the *updater*, to perform *Apply* invocations. The other processes, called *scanners*, can only perform *Scan* invocations. Note that this is different from a *single-writer* implementation [1, 2, 5], in which process p_i can only perform *Apply* invocations on component i.

An execution α from an initial configuration of an implementation of an object O is *linearizable* if there exists a sequence Π of operation instances and responses that satisfies the following three properties.

- **(i.)** Π contains every complete operation instance in α immediately followed by its response. It also contains some subset of the remaining operation instances in α, each of which is immediately followed by some response.
- **(ii.)** If the response to an operation instance op_1 appears before an operation instance op_2 in α and op_2 is in Π, then op_1 appears before op_2 in Π.
- **(iii.)** Π satisfies the sequential specification of O.

The sequence Π is a *linearization* of α. An implementation is *linearizable* if every execution from every initial configuration of the implementation is linearizable.

If an execution α is linearizable, then every complete operation instance in α can be assigned a *linearization point* at which it appears to take effect. Each linearization point must occur at or after the step containing the operation instance and at or before the step containing its response. Operation instances that are not complete in α may also be assigned linearization points, which must occur at or after the step containing the operation instance. An operation instance that has been assigned a linearization point is said to be *linearized*. If there is a sequence of operation instances and responses in which the linearized operation instances are arranged according to their linearization points, each complete operation instance in α is immediately followed by its response, each operation instance that is not complete in α is immediately followed by some response, and Π satisfies the sequential specification of O, then Π is a linearization of α.

An implementation is called *wait-free* if every operation instance by every process completes within a finite number of steps by that process. An implementation is called *lock-free* if every infinite execution of the implementation contains an infinite number of complete

operation instances. An implementation is called *obstruction-free* if every operation instance by every process completes within a finite number of consecutive steps by that process. Note that every wait-free implementation is also lock-free, and every lock-free implementation is also obstruction-free.

4 Upper bound

In this section, we discuss two scannable object implementations from base objects with bounded domain sizes. Throughout this section, let O be a scannable object with k components. Let $\{p_0, \ldots, p_{n-1}\}$ be the set of processes. First, we argue that our lock-free implementation of a scannable binary object from [22] can be generalized to use fewer base objects with larger domain sizes. We obtain a lock-free implementation of O from k objects with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b} \rceil$ base objects with domain equal to the set of all binary strings of length b, each of which supports *Read* and *Set-bit*(i, v). The *Set-bit*(i, v) invocation changes the i-th bit of the object's value to $v \in \{0, 1\}$. Second, by weakening our progress requirement to obstruction-freedom, we show how to obtain an implementation from base objects with smaller domains. We construct an obstruction-free implementation of O from k objects with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b-1} \rceil$ multi-reader, multi-writer registers with domain size b.

4.1 Lock-free implementation

We presented a lock-free, n-process implementation of a k-component scannable binary object S from k objects $B_1, \ldots, B_k$ with the same types as $S[1], \ldots, S[k]$ along with n binary registers $R_1, \ldots, R_n$ [22]. The base objects $B_1, \ldots, B_k$ are used to store the values of $S[1], \ldots, S[k]$, and the registers $R_1, \ldots, R_n$ are used by scanning processes to detect concurrent *Apply* operations.

To perform an *Apply*(ℓ, op) operation, a process writes 0 to all of the registers $R_1, \ldots, R_n$ and then applies op to B_ℓ. To *Scan*, process p_i first collects the values in $B_1, \ldots, B_k$ and then writes 1 to R_i. Then p_i repeatedly collects the values in $B_1, \ldots, B_k, R_i$. When it sees the same sequence of values in $B_1, \ldots, B_k$ and reads the value 1 from R_i n times in a row, process p_i returns the sequence of values it read from $B_1, \ldots, B_k$. If the value of some base object B_ℓ changed since p_i's last collect or $R_i = 0$, then p_i writes 1 to R_i and restarts its sequence of collects.

More generally, there is a lock-free, n-process implementation of O from k objects $B_1, \ldots, B_k$ with the same types as $O[1], \ldots, O[k]$ along with $\lceil \frac{n}{b} \rceil$ base objects $R_1, \ldots, R_{\lceil n/b \rceil}$ with domain equal to the set of all binary strings of length b, each of which supports *Read* and *Set-bit*(i, v). To perform an *Apply*(ℓ, op) operation, a process sets all of the bits of $R_1, \ldots, R_{\lceil n/b \rceil}$ to 0. Then, the process applies op to B_ℓ. To *Scan*, process p_i first collects the values in $B_1, \ldots, B_k$ and then applies *Set-bit*$(i \mod b, 1)$ to $R_{\lceil (i+1)/b \rceil}$. The remainder of the implementation is similar to our implementation of a scannable binary object, except that p_i uses the $(i \mod b)$-th bit of $R_{\lceil (i+1)/b \rceil}$ to detect concurrent *Apply* operations.

4.2 Obstruction-free implementation

Implementation 1 is a linearizable, obstruction-free implementation of O. It uses k objects $B_1, \ldots, B_k$ with the same types as the components of O, along with $\lceil \frac{n}{b-1} \rceil$ multi-reader, multi-writer registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ with domain $\{0, \ldots, b-1\}$. The base objects $B_1, \ldots, B_k$ are used to store the values of the components. The registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ are used by scanning processes to determine whether other processes are concurrently taking steps.

Each process p_i stores a pair of constants $j = \lceil (i+1)/(b-1) \rceil$ and $v = \big(i \mod (b-1)\big)+1$. The constant j denotes the index of the register that p_i accesses during its *Scan* operations, and v is the value that p_i writes to that register. Notice that $v > 0$ and process p_i is the only process that can write the value v to the register R_j. Hence, if p_i writes the value v to R_j and then p_i later reads the value v from R_j, then p_i knows no other process has modified R_j since p_i last wrote to it.

Process p_i begins a *Scan* operation by collecting the values in $B_1, \ldots, B_k$ on line 7 and then writing the value v to R_j on line 9. Then process p_i begins repeatedly collecting the values in the base objects $B_1, \ldots, B_k, R_j$ on lines 11-12. Once p_i sees the same sequence of values in $B_1, \ldots, B_k$ and it sees the value v in R_j n times in a row, p_i returns the sequence of values it saw in $B_1, \ldots, B_k$. If p_i sees that the value of some base object B_ℓ has changed since p_i's last collect, or the register R_j contains a value other than v, then p_i enters the block on line 13, writes the value v to R_j, and restarts its sequence of collects. Notice that, unlike in the lock-free implementation, it is possible for a pair of scanning processes who share the same register R_j to repeatedly interrupt each other and prevent progress. Hence, this implementation is not lock-free.

An *Apply*(ℓ, op) operation simply writes 0 to all of the registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ and then applies the operation op to B_ℓ.

Implementation 1 A linearizable, obstruction-free implementation of a k-component scannable object from $k + \lceil \frac{n}{b-1} \rceil$ base objects.

shared:
$B_1, \ldots, B_k$, each initially 0, where B_ℓ is the same type as $O[\ell]$
registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ with domain $\{0, \ldots, b-1\}$, each initially 0

local constants for process p_i:
$j \coloneqq \lceil (i+1)/(b-1) \rceil$
$v \coloneqq \big(i \mod (b-1)\big) + 1$

```
1  Apply(ℓ, op) by process p_i:
2      for r ∈ {1, ..., ⌈n/(b−1)⌉} do
3          Write(R_r, 0)
4      end
5      return op(B_ℓ)
```

```
6  Scan by process p_i:
7      S ← collect(B_1, ..., B_k)
8      c ← 0
9      Write(R_j, v)
10     while c < n do
11         S' ← collect(B_1, ..., B_k)
12         v' ← Read(R_j)
13         if S ≠ S' or v ≠ v' then
14             S ← S'
15             c ← 0
16             Write(R_j, v)
17         else
18             c ← c + 1
19         end
20     end
21     return S
```

A process p_i performing a *Scan* operation aims to perform a collect during which none of the objects $B_1, \ldots, B_k$ are modified. This way, p_i can safely return the sequence of values it returned by this collect. If an *Update* operation writes 0 to R_j before p_i has read R_j for the last time on line 12, then p_i will restart its sequence of collects after it next reads R_j. It is possible that an *Apply* operation finishes setting all of the registers $R_1, \ldots, R_{\lceil n/(b-1) \rceil}$ to 0 just before a scanning process p_i sets R_j to v. In this case, the *Apply* might change one of the base objects $B_1, \ldots, B_k$ during a collect by p_i. However, in Lemma 1, we will argue that, for every complete instance of a *Scan* operation sc, at least one of the last n collects performed by sc does not overlap with any application of a primitive to $B_1, \ldots, B_k$ on line 5.

▶ **Lemma 1.** *Let α be an execution from the initial configuration of Implementation 1. For any complete instance sc of a* Scan *operation in α, there is at least one collect among the last n collects performed by sc during which no* Apply *operation applies a primitive to any base object $B_1, \ldots, B_k$.*

Proof. Suppose that sc is performed by process p_i. Let cl_1 be the first of the final n collects of $B_1, \ldots, B_k$ performed by p_i during sc, and let cl_n be the last. Process p_i does not enter the block on line 14 between cl_1 and cl_n, as this would cause p_i to perform at least n more collects before returning from sc. Hence, p_i does not write v to R_j after cl_1 begins during sc. Furthermore, immediately after each of the last n collects performed by p_i during sc, process p_i reads the value v from R_j. Since p_i is the only process that can write v to R_j, this implies that no process writes to R_j after cl_1 begins and before cl_n ends (i.e. after p_i reads R_1 during cl_1 and before p_i reads $R_{\lceil n/(b-1)\rceil}$ during cl_n). Every *Apply* operation writes 0 to R_j on line 3 before applying a primitive to one of the base objects $B_1, \ldots, B_k$. Therefore, every *Apply* operation that applies a primitive to a base object $B_1, \ldots, B_k$ during one of the final n collects performed by p_i during sc must have written 0 to R_j before cl_1 began. Hence, at most $n-1$ *Apply* operations apply a primitive to a base object $B_1, \ldots, B_k$ during one of the final n collects of sc. ◀

▶ **Theorem 2.** *Implementation 1 is an obstruction-free, linearizable implementation of O.*

Proof. Consider some execution α of Implementation 1. By Lemma 1, there is at least one collect among the final n collects performed by any complete *Scan* operation sc during which no *Apply* operation applies a primitive to any base object $B_1, \ldots, B_k$. We can linearize sc at the beginning of this collect. All *Apply* operations in sc can be linearized when they apply the primitive on line 5.

By inspection of the code, every complete *Apply* operation applies exactly $\lceil n/(b-1)\rceil + 1$ primitives. A process performing a *Scan* operation by itself will execute at most n iterations of the loop on line 10 before terminating. Hence, Implementation 1 is obstruction-free. ◀

5 Lower bound

In this section, we present a lower bound on the number of objects with bounded domain sizes that are required to implement a scannable object. First, we explain the proof technique that we used to obtain a space complexity lower bound for scannable binary objects, since our proof in this section builds on this technique. A key concept in our technique is the notion of a $\mathcal{W}$-absent execution. Let $\mathcal{I}$ be an obstruction-free, single-updater, n-process implementation of a scannable object O, where process p_0 is the updater and processes $p_1, \ldots, p_{n-1}$ are the scanners. Let $\mathcal{W}$ be some set of values of O. Since $\mathcal{I}$ is a single-updater implementation of O, we note that in any p_0-idle configuration of $\mathcal{I}$, the value of the scannable object O is well-defined. Let α be an execution from some p_0-idle configuration C of $\mathcal{I}$. If p_0 is idle in $C\alpha$, then α is $\mathcal{W}$-absent if, for every p_0-idle configuration C' in α, the value of O in C' is not in the set $\mathcal{W}$. If p_0 is not idle in $C\alpha$, then α is $\mathcal{W}$-absent if $\alpha\alpha'$ is $\mathcal{W}$-absent, where α' is the p_0-only execution from $C\alpha$ in which p_0 finishes its ongoing operation in $C\alpha$. An execution β from $C\alpha$ is called a *$\mathcal{W}$-absent extension* of α if $\alpha\beta$ is $\mathcal{W}$-absent. The following observation is from [22].

▶ **Observation 3.** *Let α be a $\mathcal{W}$-absent execution from some p_0-idle configuration C of $\mathcal{I}$.*

(a) *If sc is an instance of a* Scan *operation in α whose response is also in α, then the response of sc is not equal to any vector in $\mathcal{W}$.*

(b) *Any execution from $C\alpha$ in which only the scanners $p_1, \ldots p_{n-1}$ take steps is a $\mathcal{W}$-absent extension of α.*

We originally considered the case in which O is a scannable binary object and $\mathcal{I}$ only uses binary base objects. We inductively construct, for all $\ell \leq min(n-2, 2^{k-1})$, a configuration C_ℓ and a set of ℓ binary k-component vectors $\{V_1, \ldots, V_\ell\}$, such that, for every $\{V_1, \ldots, V_\ell\}$-absent execution α from C_ℓ in which the last ℓ scanners take no steps, there is a set of ℓ

base objects that do not change during α. All of the vectors $V_1, \ldots, V_\ell$ have a 1 in their k-th component. We show that the ℓ base objects that stop changing can be discarded to obtain an implementation of an $(n-\ell)$-process, $(k-1)$-component scannable binary object. This idea is applied repeatedly until we have a 2-process implementation of a scannable binary object with $k' < k$ components, which we show requires at least $k'+1$ base objects.

When the base objects have larger domain sizes, it is not possible to show that a set of base objects stop changing after certain executions from C_ℓ. Hence, we cannot obtain a new implementation by discarding base objects. Instead, we will show how to construct a set of forbidden values for each base object. More precisely, consider an obstruction-free, n-process, single-updater implementation of a scannable object with k components that have domain sizes $c_1, \ldots, c_k$. We show that, for all $\ell \leq min(n-1, \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1)$, there is a sequence of distinct k-component vectors $\langle V_1, \ldots, V_\ell \rangle$, a configuration C_ℓ, and a function $\mathcal{X}_\ell$ that maps each base object to a set of forbidden values such that the following property is satisfied: For any $\{V_1, \ldots, V_\ell\}$-absent execution α from C_ℓ in which the last ℓ scanners take no steps, no base object B_x contains any of its forbidden values $\mathcal{X}_\ell(B_x)$ at any point during α. However, the updater is still able to change the object O to any vector other than $V_1, \ldots, V_\ell$ without using any forbidden value for any base object. This allows us to obtain a lower bound on the number of base objects that are needed by the implementation.

In our construction for the scannable binary object lower bound, the order of the vectors $V_1, \ldots, V_\ell$ does not matter. However, for our construction, the order of the vectors is crucial. In particular, for all $i \in \{1, \ldots, \ell\}$, it is important that every possible value of the scannable object $V' \notin \{V_1, \ldots, V_i\}$ can be reached without changing its value to any of the vectors $V_1, \ldots, V_i$. For example, consider a 2-component scannable object that consists of modulo-3 counters. Consider the sequence of vectors $\big\langle (0,1), (1,0) \big\rangle$. Notice that it is impossible to change the value of this scannable object from $(0,0)$ to $(2,2)$ without first changing its value to either $(0,1)$ or $(1,0)$. Hence, our construction would not work with this particular sequence of vectors.

Throughout the remainder of this section, we consider an obstruction-free, single-updater, n-process implementation $\mathcal{I}$ of a scannable object O with k fully reusable components that have bounded domain sizes. Let $p_0, \ldots, p_{n-1}$ be the processes using $\mathcal{I}$, where p_0 is the updater and $p_1, \ldots, p_{n-1}$ are the scanners.

Let $\mathcal{B}$ be the set of base objects used by the implementation $\mathcal{I}$. Let $B_1, \ldots, B_{|\mathcal{B}|}$ be the base objects in $\mathcal{B}$, and, for all $x \in \{1, \ldots, |\mathcal{B}|\}$, let b_x be the domain size of B_x. Without loss of generality, we assume that the domain of B_x is $\{0, \ldots, b_x - 1\}$. Let $b' = \frac{1}{|\mathcal{B}|} \sum_{x=1}^{|\mathcal{B}|} b_x$ be the average domain size of the base objects in $\mathcal{B}$. We assume that $b_x \geq 2$ for all $x \in \{1, \ldots, |\mathcal{B}|\}$. Thus, $b' \geq 2$.

For all $y \in \{1, \ldots, k\}$, let c_y be the domain size of the y-th component $O[y]$ of the implemented object O. We assume that $c_y \geq 2$ for all $y \in \{1, \ldots, k\}$. Let $h = min\big(n - 1, \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1\big)$. We will prove that $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\big)$. In particular, consider the case in which $b_1, \ldots, b_{|\mathcal{B}|}, c_1, \ldots, c_k$ are all equal to b. Our lower bound implies that,

1. if $n \leq b^k - bk + k$, then $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(k + \frac{n-1}{b} - \log_b n\big)$, and
2. if $n > b^k - bk + k$, then $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} - \frac{(b-1)k+1}{b}\big)$.

We will now show how to obtain the sequence of vectors discussed previously. Without loss of generality, we assume that, for all $y \in \{1, \ldots, k\}$, the domain of component $O[y]$ is $\{0, \ldots, c_y - 1\}$.

For all $y \in \{1, \ldots, k\}$ and all $u \in \{0, \ldots, c_y - 1\}$, define $L_y(u)$ as the length of the shortest sequence of operations that changes the value of $O[y]$ from 0 to u. Since $O[y]$ is fully reusable, $L_y(u)$ is well defined. Define a total order $\prec_y$ as follows: For all $u, v \in \{0, \ldots, c_y - 1\}$, $u \prec_y v$ if and only if either (a) $L_y(u) < L_y(v)$, or (b) $L_y(u) = L_y(v)$ and $u < v$. Consider any shortest sequence of operations σ that changes the value of $O[y]$ from 0 to u, for some $u \in \{0, \ldots, c_y - 1\}$. By definition of $\prec_y$, the sequence of values that $O[y]$ takes during σ does not contain any value $v \in \{0, \ldots, c_y - 1\}$ with $u \prec_y v$.

For all $y \in \{1, \ldots, k\}$ and all $u \in \{1, \ldots, c_y - 1\}$, define $L'_y(u)$ as the length of the shortest sequence of operations that changes the value of $O[y]$ from u to 0. Since $O[y]$ is fully reusable, $L'_y(u)$ is well defined. Define a total order $\prec'_y$ as follows: For all $u, v \in \{1, \ldots, c_y - 1\}$, $u \prec'_y v$ if and only if either (a) $L'_y(u) < L'_y(v)$, or (b) $L'_y(u) = L'_y(v)$ and $u < v$. Consider any shortest sequence of operations σ that changes the value of $O[y]$ from u to 0, for some $u \in \{1, \ldots, c_y - 1\}$. By definition of $\prec'_y$, the sequence of values that $O[y]$ takes during σ does not contain any value $v \in \{1, \ldots, c_y - 1\}$ with $u \prec'_y v$.

Let $\mathcal{U} = \{0, \ldots, c_1 - 1\} \times \ldots \times \{0, \ldots, c_k - 1\}$ be the set of values of the scannable object O. Let $\mathcal{V} \subsetneq \mathcal{U}$ be the set of all vectors in $\mathcal{U}$ that have at least two components with nonzero values. Note that $|\mathcal{V}| = \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1$. For all $j \in \{1, \ldots, k-1\}$, define $\mathcal{S}_j \subseteq \mathcal{V}$ as the set of all vectors in $\mathcal{V}$ whose first $j-1$ components contain the value 0 and whose j-th components contain a nonzero value. Notice that $\mathcal{S}_1, \ldots, \mathcal{S}_{k-1}$ is a partition of the set $\mathcal{V}$.

For all $j \in \{1, \ldots, k-1\}$, let Γ_j be the sequence of all vectors in $\mathcal{S}_j$ ordered first by decreasing lexicographical order of the final $k - j$ components with respect to $\prec_y$, and then in decreasing order by the value in the j-th component with respect to $\prec'_j$. For example, if O consists of $k = 3$ modulo-3 counters, then $2 \prec'_y 1$ and $0 \prec_y 1 \prec_y 2$ for all y. In this case, $\Gamma_1 = \langle [1,2,2], [2,2,2], [1,2,1], \ldots, [2,1,0], [1,0,1], [2,0,1] \rangle$ and $\Gamma_2 = \langle [0,1,2], [0,2,2], [0,1,1], [0,2,1] \rangle$. For all $i \in \{1, \ldots, |\mathcal{V}|\}$, define V_i as the i-th vector in the concatenation of $\Gamma_1, \ldots, \Gamma_{k-1}$. We use $[0, \ldots, 0]$ to denote the k-component all 0 vector.

▶ **Lemma 4.** *For any $i \in \{1, \ldots, |\mathcal{V}|\}$, any $U \in \mathcal{U} - \{V_1, \ldots, V_i\}$, and any p_0-idle configuration C in which the scannable object O contains $[0, \ldots, 0]$, there is a p_0-only, $\{V_1, \ldots, V_i\}$-absent execution λ from C such that $C\lambda$ is p_0-idle and the object O contains U in $C\lambda$.*

Proof. First suppose that $U \in \mathcal{U} - \mathcal{V}$. If $U = [0, \ldots, 0]$, then let λ be the empty execution. Otherwise, let the j-th component of U be a nonzero value. Let λ be some p_0-only execution from C in which p_0 changes the value of $O[j]$ from 0 to $U[j]$. In every p_0-idle configuration of λ, the value of O is a vector in $\mathcal{U} - \mathcal{V}$. Hence, λ is $\{V_1, \ldots, V_i\}$-absent.

Otherwise, $U = V_{i'}$, where $i < i' \leq |\mathcal{V}|$. Suppose that $V_{i'} \in \mathcal{S}_j$, for some $j \in \{1, \ldots, k-1\}$. Let λ_j be the p_0-only execution from C in which p_0 performs a shortest sequence of operations that changes the value of $O[j]$ from 0 to $V_{i'}[j]$. For all $y \in \{j+1, \ldots, k\}$, let λ_y be the p_0-only execution from $C\lambda_j \ldots \lambda_{y-1}$ in which p_0 performs a shortest sequence of operations that changes the value of $O[y]$ from 0 to $V_{i'}[y]$. Note that the sequence of values of $O[y]$ during λ_y are in increasing order with respect to $\prec_y$. Since $V_{i'} \in \mathcal{S}_j$, the first $j-1$ components of $V_{i'}$ contain the value 0. Hence, in $C\lambda_j \ldots \lambda_k$, the value of O is the vector $V_{i'}$.

During the execution λ_j, the updater p_0 changes the value of $O[j]$ from 0 to $V_{i'}[j]$. All of the other components of O contain 0 throughout λ_j. Hence, the object O only contains vectors in $\mathcal{U} - \mathcal{V}$ during λ_j. Thus, λ_j is $\{V_1, \ldots, V_i\}$-absent.

In the configuration $C\lambda_j$, component $O[j]$ contains the value $V_{i'}[j]$, and component $O[j]$ is not changed during $\lambda_{j+1} \ldots \lambda_k$. Furthermore, the first $j-1$ components of O contain the value 0 throughout $\lambda_j \ldots \lambda_k$. Hence, the value of the object O in every p_0-idle configuration that appears after $C\lambda_j$ in the execution $\lambda_{j+1} \ldots \lambda_k$ is a vector in $\mathcal{S}_j$. For all $y \in \{j+1, \ldots, k\}$,

every operation that p_0 applies to $O[y]$ increases the value of $O[y]$ with respect to the order $\prec_y$. Hence, the sequence of all values of O are in increasing lexicographical order with respect to $\prec_y$ and the final vector in this sequence is $V_{i'}$. Every vector V that appears before $V_{i'}$ in Γ_j with $V[j] = V_{i'}[j]$ is lexicographically larger than $V_{i'}$ with respect to $\prec_y$. Hence, $\lambda = \lambda_j \dots \lambda_k$ is $\{V_1, \dots, V_i\}$-absent. ◀

▶ **Lemma 5.** *For any $i \in \{1, \dots, |\mathcal{V}|\}$, any $U \in \mathcal{U} - \{V_1, \dots, V_i\}$, and any p_0-idle configuration C in which the scannable object O contains the value U, there is a p_0-only, $\{V_1, \dots, V_i\}$-absent execution τ from C such that $C\tau$ is p_0-idle and the object O contains $[0, \dots, 0]$ in $C\tau$.*

Proof. First suppose that $U \in \mathcal{U} - \mathcal{V}$. If $U = [0, \dots, 0]$, then let τ be the empty execution. Otherwise, let the j-th component of U be a nonzero value. Let τ be a p_0-only execution from C in which p_0 changes the value of $O[j]$ from $U[j]$ to 0. In every p_0-idle configuration of τ, the value of O is a vector in $\mathcal{U} - \mathcal{V}$. Hence, τ is $\{V_1, \dots, V_i\}$-absent.

Otherwise, $U = V_{i'}$, where $i < i' \leq |\mathcal{V}|$. Suppose that $V_{i'} \in \mathcal{S}_j$, for some $j \in \{1, \dots, k-1\}$. Let τ_j be the p_0-only execution from C in which p_0 performs a shortest sequence of operations that changes the value of $O[j]$ from $V_{i'}[j]$ to 0. For all $y \in \{j+1, \dots, k\}$, let τ_y be the p_0-only execution from $C\tau_j \dots \tau_{y-1}$ in which p_0 performs a shortest sequence of operations that changes the value of $O[y]$ from $V_{i'}[j]$ to 0. Since $V_{i'} \in \mathcal{S}_j$, the first $j-1$ components of $V_{i'}$ contain the value 0. Hence, in $C\tau_j \dots \tau_k$, the value of O is $[0, \dots, 0]$.

During the execution τ_j, the updater p_0 changes the value of $O[j]$ from $V_{i'}[j]$ to 0. None of the other components are modified during τ_j. Recall that $\mathcal{S}_j$ is ordered first in decreasing lexicographical order by the last $k-j$ components with respect to $\prec_y$, and then in decreasing order by the j-th component with respect to $\prec'_j$. Every operation by p_0 in τ_j decreases the value of $O[j]$ with respect to $\prec'_j$. In configuration $C\tau_j$, the value of O is a vector Y that contains 0 in its first $j+1$ components. Hence, either $Y \in \mathcal{S}_{j+1} \cup \dots \cup \mathcal{S}_{k-1}$ or $Y \in \mathcal{U} - \mathcal{V}$. Thus, τ_j is $\{V_1, \dots, V_i\}$-absent.

Notice that p_0 does not modify any of the first $j+1$ components of O after the configuration $C\tau_j$ in $\tau_{j+1} \dots \tau_k$. That is, in every configuration of $\tau_{j+1} \dots \tau_k$ after $C\tau_j$, the value of the object is a vector that contains 0 in its first $j+1$ components. Thus, $\tau = \tau_j \dots \tau_k$ is $\{V_1, \dots, V_i\}$-absent. ◀

We will now prove our main technical lemma, which constructs a set of forbidden values for each of the base objects in $\mathcal{B}$. Let C_0 be an initial configuration of $\mathcal{I}$ in which O contains the value $[0, \dots, 0]$. In the following lemma, we use induction to show that, for all $0 \leq \ell \leq h$, there is an execution α_ℓ from C_0 and a function $\mathcal{X}_\ell$ that maps each base object B_x to a proper subset of $\{0, \dots, b_x - 1\}$, where $\mathcal{X}_\ell(B_x)$ represents the set of forbidden values for the base object B_x. The first $n - \ell$ processes are idle and O contains $[0, \dots, 0]$ in the configuration

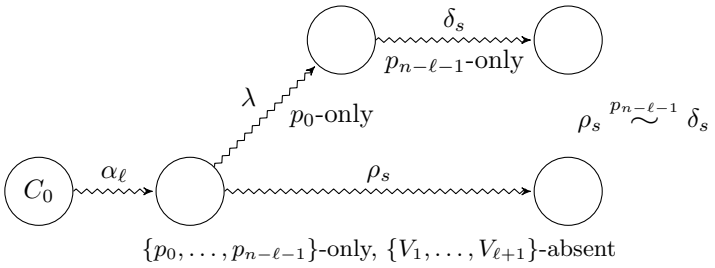

Figure 1 The executions δ_s and ρ_s in the proof of Lemma 6. Notice that δ_s starts from $C_0\alpha_\ell\lambda$ and ρ_s starts from $C_0\alpha_\ell$.

$C_0\alpha_\ell$. Furthermore, the number of forbidden values summed over all the base objects is exactly ℓ. For any $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution from $C_0\alpha_\ell$, none of the forbidden values are used during that execution. To complete the inductive step, we show how to obtain one more forbidden value by stalling the scanner $p_{n-\ell-1}$.

▶ **Lemma 6.** *For all ℓ such that $0 \leq \ell \leq h$, there is an execution α_ℓ from C_0 and a function $\mathcal{X}_\ell$ that maps each base object $B_x \in \mathcal{B}$ to a proper subset of $\{0, \ldots, b_x - 1\}$ such that*

(a) *$C_0\alpha_\ell$ is $\{p_0, \ldots, p_{n-\ell-1}\}$-idle,*
(b) *the object O contains the value $[0, \ldots, 0]$ in $C_0\alpha_\ell$,*
(c) *$\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_\ell(B_x)| = \ell$,*
(d) *for every $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution γ from $C_0\alpha_\ell$ and every $B_x \in \mathcal{B}$, we have $\text{value}(B_x, C_0\alpha_\ell\gamma) \notin \mathcal{X}_\ell(B_x)$.*

Proof. We use induction on ℓ. Let α_0 be the empty execution and let $\mathcal{X}_0(B_x) = \emptyset$ for all $B_x \in \mathcal{B}$. Since no processes have taken any steps in $C_0\alpha_0 = C_0$, part (a) holds. Since all of the components contain the value 0 in the configuration $C_0\alpha_0 = C_0$, this gives us part (b). Since $\mathcal{X}_0(B_x) = \emptyset$ for all $B_x \in \mathcal{B}$, we know that $\sum_{x=1}^{|\mathcal{B}|} \mathcal{X}_0(B_x) = 0$, which gives us part (c) and part (d). This concludes the proof of the base case.

Now let $0 \leq \ell < h$ and suppose the lemma holds for ℓ. Then there exists an execution α_ℓ from C_0 that satisfies parts (a)–(d) of the lemma statement. By Lemma 4 (with $i = \ell$, $U = V_{\ell+1}$, and $C = C_0\alpha_\ell$), there is a p_0-only, $\{V_1, \ldots, V_\ell\}$-absent execution λ from $C_0\alpha_\ell$ such that p_0 is idle in $C_0\alpha_\ell\lambda$ and the object O contains the value $V_{\ell+1}$ in $C_0\alpha_\ell\lambda$.

Process $p_{n-\ell-1}$ is idle in $C_0\alpha_\ell$ by property (a). Since $p_{n-\ell-1}$ takes no steps in λ, it is idle in $C_0\alpha_\ell\lambda$ as well. Let δ be the $p_{n-\ell-1}$-only execution from $C_0\alpha_\ell\lambda$ in which $p_{n-\ell-1}$ does a complete *Scan*. Since the value of O is $V_{\ell+1}$ in $C_0\alpha_\ell\lambda$, the *Scan* operation by $p_{n-\ell-1}$ in δ returns the vector $V_{\ell+1}$. Furthermore, the execution δ is a $\{V_1, \ldots, V_\ell\}$-absent extension of λ by Observation 3 (b). Let r be the number of steps by $p_{n-\ell-1}$ in δ. Define δ_s as the prefix of δ consisting of the first s steps by $p_{n-\ell-1}$. (In particular, δ_0 is empty and $\delta_r = \delta$.)

Let ρ_0 be the empty execution from $C_0\alpha_\ell$. Since $p_{n-\ell-1}$ takes no steps in λ, we know that $C_0\alpha_\ell \overset{p_{n-\ell-1}}{\sim} C_0\alpha_\ell\lambda$. Furthermore, since $p_{n-\ell-1}$ takes no steps in either ρ_0 or δ_0, we know that $\rho_0 \overset{p_{n-\ell-1}}{\sim} \delta_0$. Since O contains the value $[0, \ldots, 0]$ in $C_0\alpha_\ell\rho_0 = C_0\alpha_\ell$ by property (b), the execution ρ_0 is $\{V_1, \ldots, V_{\ell+1}\}$-absent.

Let ρ_r be any $\{p_0, \ldots, p_{n-\ell-1}\}$-only execution from $C_0\alpha_\ell$ such that $\rho_r \overset{p_{n-\ell-1}}{\sim} \delta_r$. Then $p_{n-\ell-1}$'s *Scan* operation in ρ_r returns the vector $V_{\ell+1}$. Hence, by the contrapositive of Observation 3 (a), the execution ρ_r is not $\{V_1, \ldots, V_{\ell+1}\}$-absent.

Let $s \in \{0, \ldots, r-1\}$ be the maximum value such that there is a $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution ρ_s from $C_0\alpha_\ell$ such that $\rho_s \overset{p_{n-\ell-1}}{\sim} \delta_s$. Then there is no $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent extension ρ' of ρ_s such that $\rho_s\rho' \overset{p_{n-\ell-1}}{\sim} \delta_{s+1}$. Suppose that $p_{n-\ell-1}$ is poised to access the base object B_w in $C_0\alpha_\ell\lambda\delta_s$ and $C_0\alpha_\ell\rho_s$. Let d be the last step of δ_{s+1}. If there is a $\{p_0, \ldots, p_{n-\ell-2}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent extension ρ' of ρ_s such that $value(B_w, C_0\alpha_\ell\rho_s\rho') = value(B_w, C_0\alpha_\ell\delta_s)$, then $\rho_s\rho'd \overset{p_{n-\ell-1}}{\sim} \delta_s d = \delta_{s+1}$. By Observation 3 (b), the execution $\rho_s\rho'd$ is $\{V_1, \ldots, V_{\ell+1}\}$-absent. This contradicts the definition of s. Hence,

$$value(B_w, C_0\alpha_\ell\rho_s\rho') \neq value(B_w, C_0\alpha_\ell\lambda\delta_s) \text{ for every } \{p_0, \ldots, p_{n-\ell-2}\}\text{-only, } \{V_1, \ldots, V_{\ell+1}\}\text{-absent extension } \rho' \text{ of } \rho_s. \quad (1)$$

Let σ_ℓ be the $\{p_0, \ldots, p_{n-\ell-2}\}$-only execution from $C_0\alpha_i\rho_s$ in which the processes $p_0, \ldots, p_{n-\ell-2}$ complete their pending operations in increasing order of their identifiers. Suppose that $\sigma_\ell = \sigma'_\ell\sigma''_\ell$, where σ'_ℓ is the prefix of σ_ℓ that contains all of p_0's steps in σ_ℓ.

Since p_0 does not begin any new *Apply* operations during σ'_ℓ, it is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of ρ_s. Furthermore, since σ''_ℓ only contains steps by the scanners $p_1, \ldots, p_{n-1}$, Observation 3 (b) implies that σ''_ℓ is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of $\rho_s\sigma'_\ell$. Thus, σ_ℓ is a $\{V_1, \ldots, V_{\ell+1}\}$-absent extension of ρ_s.

Let Y be the value of the object O in configuration $C_0\alpha_\ell\rho_s\sigma_\ell$. Since $\rho_s\sigma_\ell$ is $\{V_1, \ldots, V_{\ell+1}\}$-absent, we know that $Y \in \mathcal{U} - \{V_1, \ldots, V_{\ell+1}\}$. By Lemma 5 (with $i = \ell+1$, $U = Y$, and $C = C_0\alpha_\ell\rho_s\sigma_\ell$), there exists a p_0-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution τ_ℓ from $C_0\alpha_\ell\rho_s\sigma_\ell$ such that p_0 is idle in $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell$ and the object O contains $[0, \ldots, 0]$ in $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell$.

Let $\alpha_{\ell+1} = \alpha_\ell\rho_s\sigma_\ell\tau_\ell$. In configuration $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell = C_0\alpha_{\ell+1}$, the object O contains the value $[0, \ldots, 0]$. This gives us property (b) for $\ell+1$.

By definition of σ_ℓ, the configuration $C_0\alpha_\ell\rho_s\sigma_\ell$ is $\{p_0, \ldots, p_{n-\ell-2}\}$-idle. Since processes $p_1, \ldots, p_{n-\ell-2}$ take no steps during τ_ℓ, configuration $C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell = C_0\alpha_{\ell+1}$ is $\{p_1, \ldots, p_{n-\ell-2}\}$-idle. Furthermore, this configuration is p_0-idle by definition of τ_ℓ. This gives us property (a) for $\ell+1$.

For all $B_x \in \mathcal{B}$, define

$$\mathcal{X}_{\ell+1}(B_x) = \begin{cases} \mathcal{X}_\ell(B_x) \cup \{value(B_x, C_0\alpha_\ell\lambda\delta_s)\} & \text{if } B_x = B_w \\ \mathcal{X}_\ell(B_x) & \text{otherwise.} \end{cases}$$

Recall that $\lambda\delta_s$ is $\{V_1, \ldots, V_\ell\}$-absent. Hence, $value(B_w, C_0\alpha_\ell\lambda\delta_s) \notin \mathcal{X}_\ell(B_w)$ by property (d) for ℓ with $\gamma = \lambda\delta_s$. Thus, we have $|\mathcal{X}_{\ell+1}(B_w)| = |\mathcal{X}_\ell(B_w)| + 1$. Since $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_\ell(B_x)| = \ell$ by property (c) for ℓ, we have $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_{\ell+1}(B_x)| = \ell+1$. This gives us property (c) for $\ell+1$.

Let γ' be a $\{p_0, \ldots, p_{n-\ell-2}\}$-only, $\{V_1, \ldots, V_{\ell+1}\}$-absent execution from $C_0\alpha_{\ell+1}$. Then $\rho_s\sigma_\ell\tau_\ell\gamma'$ is a $\{p_0, \ldots, p_{n-\ell-1}\}$-only, $\{V_1, \ldots, V_\ell\}$-absent execution from $C_0\alpha_\ell$. By property (d) for ℓ with $\gamma = \rho_s\sigma_\ell\tau_\ell\gamma'$, for every $B_x \in \mathcal{B}$, we have $value(B_x, C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell\gamma') \notin \mathcal{X}_\ell(B_x)$. By (1) with $\rho' = \sigma_\ell\tau_\ell\gamma'$, we have $value(B_w, C_0\alpha_\ell\rho_s\sigma_\ell\tau_\ell\gamma') \neq value(B_w, C_0\alpha_\ell\lambda\delta_s)$. This completes the proof of property (d) for $\ell+1$. Hence, by induction, the lemma holds for all $\ell \in \{0, \ldots, h\}$. ◀

We apply Lemma 6 with $\ell = h$ to obtain an execution α_h and h forbidden values for the base objects. We apply Lemma 4 to obtain p_0-only, $\{V_1, \ldots, V_h\}$-absent executions from $C_0\alpha_h$ in which p_0 changes the value of O to the vectors in $\mathcal{U} - \{V_1, \ldots, V_h\}$. None of the forbidden values of any base objects can be used in these executions. This allows us to obtain a lower bound on the number of base objects in $\mathcal{B}$. We provide a sketch of the proof in the following theorem, and complete the proof in Appendix A.

▶ **Theorem 7.** $|\mathcal{B}| \geq \frac{1}{2} \cdot \left(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\right)$.

Proof sketch. Apply Lemma 6 with $\ell = h$ to obtain an execution α_h and a function $\mathcal{X}_h$ that satisfy (a)–(d). By property (c), we have $\sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_h(B_x)| = h$. Since $|\mathcal{X}_h(B_x)| \leq b_x - 1$ for all $B_x \in \mathcal{B}$, we have $\sum_{x=1}^{|\mathcal{B}|}(b_x - 1) \geq h$. Hence, $\frac{1}{|\mathcal{B}|} \cdot \sum_{x=1}^{|\mathcal{B}|}(b_x - 1) \geq \frac{h}{|\mathcal{B}|}$. Therefore, $|\mathcal{B}| \geq \frac{h}{b'-1}$.

Let $\mathcal{V}'$ be the set of all values of O except for $V_1, \ldots, V_h$. Then $|\mathcal{V}'| = \prod_{y=1}^{k} c_y - h$. By Lemma 6 (b), the object O contains the value $[0, \ldots, 0]$ in $C_0\alpha_h$. For all $V' \in \mathcal{V}'$, there exists a p_0-only, $\{V_1, \ldots, V_h\}$-absent execution $\gamma_{V'}$ from $C_0\alpha_h$ such that p_0 is idle in $C_0\alpha_h\gamma_{V'}$ and O contains V' in $C_0\alpha_h\gamma_{V'}$ by Lemma 4 (with $i = h$, $U = V'$, and $C = C_0\alpha_h$).

Let V'_1, V'_2 be two distinct vectors in $\mathcal{V}'$. Consider the p_1-only executions from $C_0\alpha_h\gamma_{V'_1}$ and $C_0\alpha_h\gamma_{V'_2}$ in which p_1 finishes its ongoing *Scan* operation (if it has one) and then performs a complete *Scan*. The complete *Scan* operation in p_1's solo execution from $C_0\alpha_h\gamma_{V'_1}$ returns the vector V'_1 and the complete *Scan* operation in p_1's solo execution from $C_0\alpha_h\gamma_{V'_2}$ returns the vector $V'_2 \neq V'_1$. Since p_1 takes no steps in $\gamma_{V'_1}$ or $\gamma_{V'_2}$, it must be true that $C_0\alpha_h\gamma_{V'_1} \overset{p_1}{\sim} C_0\alpha_h\gamma_{V'_2}$. Therefore, at least one base object must have different values in $C_0\alpha_h\gamma_{V'_1}$ and $C_0\alpha_h\gamma_{V'_2}$.

Lemma 6 (d) implies that, for every p_0-only, $\{V_1, \ldots, V_h\}$-absent execution γ from C_0 and every $B_x \in \mathcal{B}$, we have $value(B_x, C_0\alpha_h\gamma) \notin \mathcal{X}_h(B_x)$. Thus, there are at most $b_x - |\mathcal{X}_h(B_x)|$ possible values for any base object B_x in $C_0\alpha_h\gamma$. This means that the shared objects can hold $\prod_{x=1}^{|\mathcal{B}|}(b_x - |\mathcal{X}_h(B_x)|)$ distinct sequences of values after p_0-only, $\{V_1, \ldots, V_h\}$-absent executions from $C_0\alpha_h$. Since $\gamma_{V'}$ is a p_0-only, $\{V_1, \ldots, V_h\}$-absent execution for all $V' \in \mathcal{V}'$, we have

$$\prod_{x=1}^{|\mathcal{B}|}\big(b_x - |\mathcal{X}_h(B_x)|\big) \geq |\mathcal{V}'| = \prod_{y=1}^{k} c_y \; - h.$$

In Appendix A, we show how this implies that $|\mathcal{B}| \geq \frac{1}{2} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y + \frac{h}{b'} - \log_{b'}(h+1)\Big)$. ◀

A specific case that motivated our work in [22] is when $c_1 = \ldots = c_k = b_1 = \ldots = b_{|\mathcal{B}|} = b$. By applying Theorem 7 with $b' = c_1 = \ldots = c_k = b$, we obtain

$$|\mathcal{B}| \geq \frac{1}{2} \cdot (\sum_{y=1}^{k} \log_b b + \frac{h}{b} - \log_b(h+1)).$$

Since $\sum_{y=1}^{k} \log_b b = k$, we have $|\mathcal{B}| \geq \frac{1}{2} \cdot (k + \frac{h}{b} - \log_b(h+1))$. When $n \leq b^k - kb + k$, taking $h = n - 1$ gives us Corollary 8 (a). When $n > b^k - kb + k$, taking $h = b^k - kb + k - 1$ gives us $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} + \frac{k-1}{b} - \log_b(b^k - bk + k)\big)$. Since $\log_b(b^k - bk + k) \leq \log_b b^k = k$, this gives us Corollary 8 (b).

▶ **Corollary 8.** *If the domain sizes of every component $O[1], \ldots, O[k]$ and the domain sizes of the base objects $B_1, \ldots, B_{|\mathcal{B}|}$ are all equal to b, then*
(a) $|\mathcal{B}| \geq \frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b n)$ *when $n \leq b^k - bk + k$, and*
(b) $|\mathcal{B}| \geq \frac{1}{2} \cdot \big(b^{k-1} - \frac{(b-1)k+1}{b}\big)$ *when $n > b^k - bk + k$.*

6 Conclusion

When the domain sizes of the components and base objects used by the implementation are all equal to b and $n \leq b^k - bk + k$, our obstruction-free, single-updater lower bound of $\frac{1}{2} \cdot (k + \frac{n-1}{b} - \log_b \frac{n-1}{2})$ asymptotically matches our obstruction-free, multi-updater upper bound of $k + \lceil \frac{n}{b-1} \rceil$. For all values of n, we conjecture that $k + \lceil \frac{n}{b} \rceil$ base objects with domain size b are required by any obstruction-free, multi-updater, n-process implementation of a scannable object with k fully reusable components that have domain size b. When $n > b^k - bk + k$, we gave an obstruction-free, single-updater lower bound of $\frac{1}{2} \cdot (b^{k-1} - \frac{(b-1)k+1}{b})$ base objects. If b is a constant, then this asymptotically matches the wait-free, single-updater implementation from b^k binary registers in Section 1. This means that, in order to prove a space lower bound better than b^k for larger values of n, we need to consider more complex executions that contain concurrent *Apply* operations. We may also be able to improve our lower bound by considering stronger progress requirements like lock-freedom or wait-freedom.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993. doi:10.1145/153724.153741.

2 James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993. doi:10.1007/BF02242703.

3 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 62(1):3:1–3:22, 2015. doi:10.1145/2732263.

4 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi:10.1016/0196-6774(90)90021-6.

5 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

6 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014. doi:10.2200/S00551ED1V01Y201311DCT012.

7 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, March 1995. doi:10.1007/BF02242714.

8 Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, July 1994. doi:10.1145/179812.179902.

9 Hagit Attiya and Ophir Rachman. Atomic snapshots in o(n log n) operations. *SIAM J. Comput.*, 27(2):319–340, 1998. doi:10.1137/S0097539795279463.

10 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with $n - k + 1$ atomic read/write registers. *Distrib. Comput.*, 31(2):99–117, April 2018. doi:10.1007/s00446-017-0301-7.

11 J.E. Burns and N.A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.

12 Tian Ze Chen and Yuanhao Wei. Step-optimal implementations of large single-writer registers. *Theoretical Computer Science*, 826-827:40–50, 2020. Special issue on OPODIS 2016. doi:10.1016/j.tcs.2020.04.008.

13 Faith Ellen, Panagiota Fatourou, and Eric Ruppert. Time lower bounds for implementations of multi-writer snapshots. *J. ACM*, 54(6):30–es, December 2007. doi:10.1145/1314690.1314694.

14 Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based classification for multiprocessor synchronization. *Distributed Computing*, 33(2):125–144, April 2020. doi:10.1007/s00446-019-00361-3.

15 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Comput.*, 20(3):165–177, 2007. doi:10.1007/s00446-007-0042-0.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

17 Jaap-Henk Hoepman and John Tromp. Binary snapshots. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, WDAG '93, pages 18–25, Berlin, Heidelberg, 1993. Springer-Verlag.

18 Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1994. doi:10.1007/BFb0020429.

19 A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical systems theory*, 28(5):469–486, September 1995. doi:10.1007/BF01185868.

20 Prasad Jayanti. F-arrays: Implementation and applications. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 270–279, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/571825.571875.

21 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, April 2000. doi:10.1137/S0097539797317299.

22 Sean Ovens. The space complexity of scannable binary objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 509–519, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467916.

23 K. Vidyasankar. Converting lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6):287–290, August 1988. doi:10.1016/0020-0190(88)90175-5.

24 Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 31–46, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437801.3441602.

25 Leqi Zhu and Faith Ellen. Atomic snapshots from small registers. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.17.

A Finishing the proof of Theorem 7

Proof. In the proof sketch of Theorem 7 in Section 5, we showed that

$$\prod_{x=1}^{|\mathcal{B}|}\left(b_x - |\mathcal{X}_h(B_x)|\right) \geq |\mathcal{V}'| = \prod_{y=1}^{k} c_y \; - h. \tag{2}$$

Taking the base b' logarithm of both sides of this inequality gives us the following.

$$\sum_{x=1}^{|\mathcal{B}|}\log_{b'}\left(b_x - |\mathcal{X}_h(B_x)|\right) \geq \log_{b'}\left(\prod_{y=1}^{k} c_y - h\right) \tag{3}$$

Let $c = \prod_{y=1}^{k} c_y$. We will now show that $\log_{b'}(c-h) \geq \log_{b'} c - \log_{b'}(h+1)$. Notice that $\log_{b'}(c-h) - (\log_{b'} c - \log_{b'}(h+1)) = \log_{b'} \frac{(h+1)\cdot(c-h)}{c}$. In order to show that $\log_{b'} \frac{(h+1)\cdot(c-h)}{c} \geq 0$, it suffices to show that $(h+1)\cdot(c-h) - c \geq 0$. Notice that $(h+1)\cdot(c-h) - c = hc - h^2 - h = h(c-h-1)$. Since $h \leq \prod_{y=1}^{k} c_y - \sum_{y=1}^{k} c_y + k - 1$, we have $c \geq h+1$. We also have $h \geq 0$. Thus, $h(c-h-1) \geq 0$, which implies that $\log_{b'} \frac{(h+1)\cdot(c-h)}{c} \geq 0$. Therefore, $\log_{b'}(c-h) \geq \log_{b'} c - \log_{b'}(h+1)$. By definition of c, this implies that $\log_{b'}\left(\prod_{y=1}^{k} c_y - h\right) \geq \log_{b'}\left(\prod_{y=1}^{k} c_y\right) - \log_{b'}(h+1)$. Substituting this into (3), we obtain the following.

$$\begin{aligned}\sum_{x=1}^{|\mathcal{B}|}\log_{b'}\left(b_x - |\mathcal{X}_h(B_x)|\right) &\geq \log_{b'}\left(\prod_{y=1}^{k} c_y\right) - \log_{b'}(h+1) \\ &= \sum_{y=1}^{k}\log_{b'} c_y - \log_{b'}(h+1).\end{aligned}$$

Dividing by $|\mathcal{B}|$ on both sides of this inequality, we obtain

$$\sum_{x=1}^{|\mathcal{B}|}\frac{1}{|\mathcal{B}|}\cdot\log_{b'}\left(b_x - |\mathcal{X}_h(B_x)|\right) \geq \frac{1}{|\mathcal{B}|}\cdot\left(\sum_{y=1}^{k}\log_{b'} c_y \; - \log_{b'}(h+1)\right). \tag{4}$$

Since log is concave, Jensen's inequality implies that

$$\begin{aligned}\sum_{x=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{B}|} \cdot \log_{b'}\big(b_x - |\mathcal{X}_h(B_x)|\big) &\le \log_{b'}\Big(\sum_{x=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{B}|} \cdot \big(b_x - |\mathcal{X}_h(B_x)|\big)\Big) \\ &= \log_{b'}\Big(b' - \frac{1}{|\mathcal{B}|} \cdot \sum_{x=1}^{|\mathcal{B}|} |\mathcal{X}_h(B_x)|\Big) \\ &= \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big).\end{aligned}$$

Substituting this into (4), we have

$$\log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \ge \frac{1}{|\mathcal{B}|} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y \ - \log_{b'}(h+1)\Big).$$

Multiplying by $|\mathcal{B}|$ on both sides of the inequality, we obtain

$$|\mathcal{B}| \cdot \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \ge \sum_{y=1}^{k} \log_{b'} c_y \ - \log_{b'}(h+1). \tag{5}$$

Let $x = -\big(\frac{h}{|\mathcal{B}|}\big)$. So $\log_{b'}\big(b' - \frac{h}{|\mathcal{B}|}\big) = \log_{b'}(b'+x)$. Since $|\mathcal{B}| \ge \frac{h}{b'-1}$, we have $1 - b' \le x < 0$. The Maclaurin series expansion of $\log_{b'}(b'+x)$ is the following.

$$1 + \frac{x}{b' \cdot \ln(b')} - \frac{x^2}{2(b')^2 \cdot \ln(b')} + \frac{x^3}{3(b')^3 \cdot \ln(b')} - \frac{x^4}{4(b')^4 \cdot \ln(b')} \cdots$$

The series converges provided $|x| < |b'|$. Since $x < 0$, every term of $-\frac{x^2}{2(b')^2 \cdot \ln(b')} + \frac{x^3}{3(b')^3 \cdot \ln(b')} - \frac{x^4}{4(b')^4 \cdot \ln(b)} \cdots$ is negative. Hence, we have

$$1 + \frac{x}{b' \cdot \ln(b')} \ge \log_{b'}(b'+x). \tag{6}$$

Notice that $\frac{x}{b' \cdot \ln(b')} - \frac{x}{b'} = \frac{x \cdot (1 - \ln(b'))}{b' \cdot \ln(b')} \le 1$, since $|x| \le b' - 1$ and $b' \ge 2$. Hence, we have $1 + \frac{x}{b'} \ge \frac{x}{b' \cdot \ln(b')}$. Combined with (6) and the definition of x, this gives us

$$1 + \Big(1 - \frac{h}{|\mathcal{B}| \cdot b'}\Big) \ge 1 - \frac{h}{|\mathcal{B}| \cdot b' \cdot \ln(b')} \ge \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big).$$

Combined with (5), this gives us

$$|\mathcal{B}| \cdot \Big(2 - \frac{h}{|\mathcal{B}| \cdot b'}\Big) \ge |\mathcal{B}| \cdot \log_{b'}\Big(b' - \frac{h}{|\mathcal{B}|}\Big) \ge \sum_{y=1}^{k} \log_{b'} c_y \ - \log_{b'}(h+1).$$

Thus, we have

$$2 \cdot |\mathcal{B}| - \frac{h}{b'} \ge \sum_{y=1}^{k} \log_{b'} c_y \ - \log_{b'}(h+1).$$

Add $\frac{h}{b'}$ to both sides and then divide by 2 to obtain

$$|\mathcal{B}| \ge \frac{1}{2} \cdot \Big(\sum_{y=1}^{k} \log_{b'} c_y \ + \frac{h}{b'} - \log_{b'}(h+1)\Big).$$

This concludes the proof of the theorem. ◀