On the Importance of Synchronization Primitives with Low Consensus Numbers

Pankaj Khanchandani
ETH Zurich
kpankaj@ethz.ch

Roger Wattenhofer
ETH Zurich
wattenhofer@ethz.ch

ABSTRACT

The *consensus number* of a synchronization primitive is the maximum number of processes for which the primitive can solve consensus. This has been the traditional measure of power of a synchronization primitive. Thus, the compare-and-swap primitive, which has infinite consensus number, is considered most powerful and has been the primitive of choice for implementing concurrent data structures. In this work, we show that the synchronization primitives with low consensus numbers can also be potentially powerful by using them along with the compare-and-swap primitive to design an $O(\sqrt{n})$ time *wait-free* and *linearizable* concurrent queue. The best known time bound for a wait-free and linearizable concurrent queue using only the compare-and-swap primitive is $O(n)$. Here, n is the total number of processes that can access the queue.

The queue object maintains a sequence of elements and supports the operations enqueue(x) and dequeue(). The wait-free property implies that every call to enqueue(x) and dequeue() finishes in a bounded number of steps irrespective of the schedule of other $n-1$ processes. The linearizable property implies that the enqueue(x) and dequeue() calls appear to be instantaneously applied within the duration of respective calls. We design a wait-free and a linearizable concurrent queue using shared memory registers that support the compare-and-swap primitive and two other primitives of consensus number one and two respectively. The enqueue(x) and dequeue() operations take $O(\sqrt{n})$ steps each. The total number of registers required are $O(nm)$ of $O(\max\{\log n, \log m\})$ bits each, where m is a bound on the total number of enqueue(x) operations.

CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**;

KEYWORDS

concurrent queue, wait-free queue, sublinear wait-free

ACM Reference Format:
Pankaj Khanchandani and Roger Wattenhofer. 2018. On the Importance of Synchronization Primitives with Low Consensus Numbers. In *ICDCN '18: 19th International Conference on Distributed Computing and Networking, January 4–7, 2018, Varanasi, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3154273.3154306

ICDCN '18, January 4–7, 2018, Varanasi, India

ACM ISBN 978-1-4503-6372-3/18/01…$15.00
https://doi.org/10.1145/3154273.3154306

1 INTRODUCTION

With the advent of asynchronous shared memory multiprocessor architectures, different parallel computers provided different mechanisms for communication between processes. The NYU Ultracomputer for instance used a fetch-and-add synchronization primitive [18]. Shortly after this first wave of architectures, Maurice Herlihy presented his seminal consensus hierarchy. Among other results, Herlihy showed that the compare-and-swap synchronization primitive is fundamentally more powerful than fetch-and-add, since it can implement consensus for infinitely many processes, whereas fetch-and-add is limited to just two [8]. He also showed a universal construction for implementing concurrent non-trivial data structures using consensus. Thus, compare-and-swap and other primitives with infinite consensus number are considered most powerful and occupy a top position in the Herlihy's hierarchy.

As a result, compare-and-swap primitive is now available in every multiprocessor in the world and data structures have been traditionally designed using compare-and-swap as base objects [13, 15, 16]. But, Ellen et al. [4] recently gave a simple $O(1)$ time algorithm to solve consensus for infinitely many processes by combining the functionality of fundamentally weak primitives onto the same register. The primitives they used were multiply and decrement, which have a consensus number of one each.

In other words, Ellen et al. [4] showed that weak primitives may replace strong primitives such as compare-and-swap, without extra cost. But can weak primitives even help to lower the cost when building concurrent data structures? We answer this question by designing a sublinear time wait-free and linearizable concurrent queue using the compare-and-swap primitive and two additional primitives of consensus number one and two respectively. As no sublinear solution for a wait-free and linearizable concurrent queue by using the compare-and-swap primitive only is known, our result shows the potential of applying low consensus number primitives in the design of efficient concurrent data structures.

Our queue supports the operations enqueue(x) and dequeue(). The enqueue(x) operation adds an element x to the queue. The dequeue() operation removes and returns the element y that was added earliest (by the operation enqueue(y)). The shared memory provides $O(nm)$ registers of size $O(\max\{\log m, \log n\})$ bits each, where n is the number of processes and m is a bound on the total number of enqueue(x) operations. The registers support read, write, compare-and-swap, half-increment and half-max operations. The half-increment operation increments the value in the first half of the register if it is less than or equal to the value in the second half of the register. If the increment happens, the operation returns the value in the first half prior to the increment, otherwise it returns -1. The half-max operation takes a single argument and writes it to the second half of the register only if it is larger than the value already there. This operation does not return anything. The operation half-increment has a consensus number of two and the operation half-max has a consensus number of one. We show that the `enqueue(`x`)` and `dequeue()` operation for every process is in $O(\sqrt{n})$. This sublinear bound is better than the known wait-free and linearizable queue implementations.

2 RELATED WORK

Ellen et al. [4] showed how to combine low consensus number primitives, such as decrement and multiply, to solve consensus for any given number of processes. In fact, modern hardware supports these low consensus number primitives along with some other primitives of low consensus number such as fetch-and-increment, xor and compare-and-swap-if-greater-than (max register) [10, 11]. It is not surprising that these low consensus number primitives are better for the tasks that they support atomically. For example, one can implement a wait-free and linearizable shared counter using fetch-and-increment primitive in a single step, where as it takes $\Omega(\log n)$ steps to implement it using compare-and-swap [2]. However, we are not aware of a non-trivial application of low consensus number primitives to design fast wait-free data structures such as queues.

Queue designs using both compare-and-swap and fetch-and-add primitive have been explored before. The one from Morrison et al. [17] is lock-free but ours is wait-free. The one from Yang et al. [19] is wait-free but the `dequeue()` operation takes $O(n^2)$ steps. They also do memory reclamation, which we do not. Petrank et al. [15] give a practical wait-free queue implementation based only on the compare-and-swap primitive. The `enqueue(`x`)` operation in their implementation takes $O(n)$ steps. It is not known if this time bound can be improved if we only use the compare-and-swap primitive.

Sublinear time implementations have been proposed earlier, but only with limited concurrency. For example, David's algorithm [3] gives an $O(1)$ implementation for both the `enqueue(`x`)` and `dequeue()` operations and supports multiple dequeuers but only a single enqueuer. David's algorithm uses fetch-and-add and swap primitives. Jayanti et al. [12] give an $O(\log n)$ implementation of the `enqueue(`x`)` and `dequeue()` operations that support multiple enqueuers but only one dequeuer. Their algorithm uses the load-link/store-conditional primitive, which is closely related to the compare-and-swap primitive and also has infinite consensus number.

On the other hand, using the universal constructions of sequential objects [1, 8], we get a time complexity of $O(n)$. Fatourou et al. [7] give a universal construction that accesses the shared memory only constant number of times but executes $O(n)$ local steps. The registers used are of size $O(n^2)$ bits. Thus, we do not yet know a sublinear and wait-free queue implementation that supports multiple enqueuers and dequeuers and uses logarithmic sized registers. We hope that our work shows that synchronization primitives apart from compare-and-swap can be fundamentally important for building efficient concurrent data structures.

3 MODEL

Let us define some frequently used terms. A *sequential object* is defined by the four-tuple (S, O, R, T) where S is the set of states, O is the set of *operations* that can be applied on the object to change its state, R is the set of return values of the operations and $T : S \times O \rightarrow S \times R$ is a transition function that specifies the effect of each operation by specifying the new state and the return value. As a shorthand, we will sometimes just use the term object instead of the term sequential object.

A *register* is a sequential object whose state (value) set is $S \subseteq \mathbb{N}_0$ (or a sequence of bits). It supports the following operations.

(1) `read()`: This returns the current value of the register.
(2) `write(`s`)`: This updates the value of the register to s.
(3) `compare-and-swap(`s,s'`)`: This changes the value of the register to s' if and only if the value was s before the change. A value *true* is returned if the value of the register was changed to s', otherwise *false* is returned.
(4) `half-increment()`: This operation affects only the first half of the bits in the register. It increments the value in the first half of the register if it is less than or equal to the value in the second half. If the increment happens, the operation returns the value in the first half prior to the increment. Otherwise, it returns -1.
(5) `half-max(`x`)`: This operation affects only the second half of the bits in the register. It writes x to the second half of the register only if the value in the second half is less than x. The operation does not return anything.

Note that the operations supported by the registers are *atomic* i.e., if several processes execute them concurrently, they will be executed one after another. For the rest of the text, whenever we use the word *operation*, it will be atomic.

A *process* consists of local variables and can execute any sequence of instructions. An *instruction* is a computation on local variables or an operation on an object storing the result in a local variable. A *schedule* is a sequence of process IDs. A schedule S transforms into an *execution* $E(S)$ by replacing each ID in S with the next instruction in the sequence of instructions that the process ID executes. An *implementation* of an object defines a *function* (sequence of instructions) for every operation on the object. If a process wants to apply an operation on the object, it must execute the sequence of operations specified in the corresponding function (*call* the function). As a shorthand, instead of saying "calling a function of an object's implementation", we will just say "calling a function on an object". We will only consider executions in which a process finishes executing a function before calling another function on the object. Unlike the operations on an object, the functions are not atomic.

Consider an execution E. The *start* of a function call F is the first instruction in E that belongs to F and involves non-local computation. Similarly, the *end* of a function call F is the last instruction in E that belongs to F and involves non-local computation. Given an execution E and an object O, we define a partial order $P_O(E)$ of the function calls on O. The order $P_O(E)$ orders a function call A before a function call B if A ends before B starts in the execution E. An implementation of an object O is *linearizable* if for every execution E, the partial order $P_O(E)$ can be extended into a total order $T_O(E)$

so that the return value of the function calls that ended in E is same as obtained by applying the transition function on the order $T_O(E)$.

Usually, we get the order $T_O(E)$ by picking an instruction in the execution E, called the *linearization point*, for each function call and ordering the calls according to the order of corresponding linearization points in the execution E. We use the following property of linearizability from [9].

Lemma 1. *Consider a linearizable implementation I of a sequential object O. We can obtain another linearizable implementation I' of the object O as follows. We pick a subset S of sequential objects that the implementation I uses. We replace each operation on an object $s \in S$ by calling a corresponding function of a linearizable implementation of s.*

The state of a sequential object *queue* is a sequence of elements. The `enqueue(`x`)` operations adds an element x at the end of the sequence. The `dequeue()` operation removes and returns the element at the beginning of the sequence if there is one. Otherwise, it returns -1. We look for a linearizable and wait-free implementation of a queue that can be simultaneously used by all the n processes. The processes have IDs $1, 2, \ldots, n$. We assume a bound of m on the total number of `enqueue(`x`)` operations.

4 A BIRD’S EYE VIEW

A conceptual overview of the construction is as follows. We use an array of size m to store the elements of the queue. Each `enqueue(`x`)` call is assigned a slot in the array to store the element x. Ideally, we would want that the slot assignment and storing the element in the array happens atomically. As we cannot do this, we partition our original problem into the following two problems.

1. First, we want to implement a linearizable and wait-free set in which an `enqueue(`x`)` call inserts or announces the element to be enqueued and is also assigned a slot number in return. One can see this object as a combination of a set object and counter. We call this object as the *counting set.* The purpose of this object is to manage and order the pending enqueue operations. Concretely, the counting set stores at most one element per process and supports the following two operations.
   - (a) `insert(`x`)`: This operation inserts an element x into the set and returns the number of inserts completed (i.e., also counts apart from inserting).
   - (b) `remove(`i`)`: This operation takes an integer argument i that is at most the number of `insert(`x`)` operations already completed and returns the i^{th} inserted element if it was the latest element inserted by the corresponding process. Concretely, we have the following two cases for a legal argument i.
     - (i) The operation returns x if an `insert(`x`)` operation by a process p returned i and p did not issue an `insert(`y`)` operation after `insert(`x`)`.
     - (ii) The operation returns $\perp$ if an `insert(`x`)` operation by a process p returned i and p issued an `insert(`y`)` operation after `insert(`x`)`.

   Although the counting set object closely resembles a counter, we cannot use the existing counter implementations as they are either not wait-free or do not support the set semantics [5, 6, 14].
2. Second, we want to implement a linearizable and wait-free queue using the counting set object from the previous step.

We solve the first problem using recursion on the number of processes and get the recurrence $T(n) = T(n/2) + \sqrt{n}$ for every function call on the set. This yields a runtime of $O(\sqrt{n})$ for both the `insert(`x`)` and `remove(`i`)` calls.

Then, we solve the second problem using Lemma 1. In particular, we implement a queue assuming we have the counting set with its atomic operations. We will see that `enqueue(`x`)` can be implemented using the `insert(`x`)` operation plus $O(1)$ additional steps, and `dequeue()` can be implemented using the `remove(`i`)` operation plus $O(1)$ additional steps. In these implementations, we use the register operations `half-max(`x`)` and `half-increment()` to manage the head and the tail of the queue.

In the following section, we describe the algorithm to implement the counting set, and we analyze it. Section 6 describes a queue implementation using the counting set, and analyzes it. In Section 7, we show that the consensus number of the `half-increment()` operation is two and that the consensus number of the `half-max(`x`)` operation is one. In Section 8, we follow up with a discussion on the results and open questions.

5 THE COUNTING SET

We first implement a counting set that supports $k > 1$ processes assuming two counting set objects that support up to $k/2$ processes each. Let us call these counting set objects as C_l and C_r. Later, we use Lemma 1 to recursively replace the operations on C_l and C_r by their corresponding function calls. The base case of the recursion is just a sequential implementation for a single process, i.e., $k = 1$.

Algorithm 1 gives a conceptual overview of an `insert(`x`)` call that can be called by $k > 1$ processes simultaneously. We use the pro-

Algorithm 1: Outline of an `insert(`x`)` call.

```
insert(x)
    Let $I$ be the ID set of the processes that might call this function and $k = |I|$;
    Split $I$ into two halves $L$ and $R$ so that $|L| \leq k/2$ and $|R| \leq k/2$;
    Let $i$ be the ID of the process calling this function;
    if $i \in L$ then
        $t \leftarrow C_l$.insert($x$);
        Aggregate and log the insert($x$) operations completed on $C_l$ and $C_r$;
        Lookup the log for the return value using $t$ as the key;
    else /* $i \in R$ */
        $t \leftarrow C_r$.insert($x$);
        Aggregate and log the insert($x$) operations completed on $C_l$ and $C_r$;
        Lookup the log for the return value using $t$ as the key;
```

cess IDs to split them into two equal halves, called the *left* half and the *right* half. If a process from the left half calls `insert(`x`)`, then it is delegated to the operation `insert(`x`)` on C_l. Otherwise, it is delegated to the operation `insert(`x`)` on C_r. To compute the return value for the `insert(`x`)` call, we need to aggregate the `insert(`x`)` operations performed on the sets C_l and C_r. As this should be sub-linear, we cannot aggregate these operations sequentially. Instead, a bunch of them are aggregated together. As this must be wait-free as well, all the processes try to aggregate despite only one being successful. Some metadata about the successful aggregation is also logged in a history, so that the other processes can look it up to find their return value.

The `remove(`i`)` call is also implemented similarly using recursion. The base case for $k = 1$ process is just a sequential implementation. Algorithm 2 gives an overview of the `remove(`i`)` call for $k > 1$ processes. Here, we use the log to determine whether the `insert(`x`)` call that returned i executed the `insert(`x`)` operation on C_l or C_r. If it was performed on C_l, then we perform the `remove(`i`)` operation on C_l, otherwise on C_r.

Algorithm 2: Outline of a `remove(`i`)` call.

```
remove($i$)
    Lookup the log to find whether the insert($x$) call that
      returned $i$ called $C_l$.insert($x$) or $C_r$.insert($x$);
    Lookup the log to find the value $t$ returned by the
      corresponding insert($x$) operation on $C_l$ or $C_r$;
    Accordingly, perform remove($t$) operation on $C_l$ or $C_r$;
```

Admittedly, the above descriptions are high-level and they skip several details. For example, what is exactly stored in the log? How to maintain the log in sub-linear time ensuring wait-freedom? The following section describes the complete algorithm and answers these questions.

5.1 Algorithm

Figure 1 gives a pictorial overview of the data structures used. If there is only one process that might call the function, then we just need a single register P. The register P stores a pair of fields: e and t. The field e stores the element (or a pointer to it). The field t is the total number of elements that have been inserted by the process (including the current one). A single register is sufficient for the base case as there is at most one element per process.

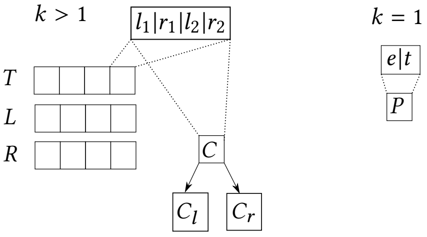

Figure 1: Overview of the data structures used in Algorithm 3.

In case there is more than one process that might call the function, we need a register C and arrays T, L, R apart from the set objects C_l and C_r. The register C stores four fields: l_1, r_1, l_2 and r_2. Whenever the register C is updated, these are updated as follows.

(1) The fields l_2 and r_2 are updated with the number of `insert(`x`)` operations completed on the sets C_l and C_r until this update.
(2) The fields l_1 and r_1 are updated with the values of l_2 and r_2 i.e., they store the number of `insert(`x`)` operations completed on the sets C_l and C_r until the previous update of register C.

The arrays T, L and R maintain the log of updates made to the register C. When the register C is updated, its content before the update is copied to appropriate indices of the arrays T, L and R. The array T is indexed by the total number of `insert(`x`)` operations performed on the set objects C_l and C_r ($l_2 + r_2$), array L by the number of `insert(`x`)` operations performed on C_l (l_2) and array R by the number of `insert(`x`)` operations performed on C_r (r_2).

Algorithm 3 gives the full top-level `insert(`x`)` function. Note that we also need an operation `total()` on the set objects C_l and C_r (Lines 15, 16, 22 and 23). This returns the total number of `insert(`x`)` operations that were performed on the object. The recursive nature of the construction implies that we also need an implementation for `total()` apart from `insert(`x`)` and `remove(`i`)`. Algorithm 4 gives the listing of `total()` function. It just returns the sum of the values l_2 and r_2. The auxiliary function `new_value` used in Lines 18 and 25 is also defined in Algorithm 4. It updates the fields l_2, r_2 and copies them to l_1 and r_1.

We postpone the definition of other auxiliary functions `log` and `lookup`, which are used in Lines 17, 24, 27 and 28. Nevertheless, we can already prove some interesting properties about Algorithm 3.

For $k = 1$, the j^{th} `insert(`x`)` call is well-defined as there is only a single process and the `insert(`x`)` function is called again only after the previous call is finished. We say that this j^{th} `insert(`x`)` call is *applied* if $P.t \geq j$. As the value $P.t$ is non-decreasing, an `insert(`x`)` call applied once remains so afterwards.

For $k > 1$, note that the value $C.l_2$ only comes from t_l (Lines 15 and 22). The value t_l is the return value of the operation `total()` on the set object C_l. It follows from the definition of this object that any distinct value of t_l or $C.l_2$ is associated with a unique set of `insert(`x`)` operations done on C_l. These are precisely the first $C.l_2$ `insert(`x`)` operations done on C_l. We say that an `insert(`x`)` operation on C_l is *applied* on the register C if it was the j^{th} `insert(`x`)` operation on C_l and $C.l_2 \geq j$. As the value l_2 written to the register C is non-decreasing, an `insert(`x`)` operation once applied remains so afterwards. The same is true for the operation `insert(`x`)` on the object C_r. We say that an `insert(`x`)` call is *applied* if the corresponding `insert(`x`)` operation it executes is applied on the register C.

The following lemma shows that every `insert(`x`)` call is applied before it ends. We assume for now that the functions `log` and `lookup` do not write to the register C (this is indeed true).

Lemma 2. *Every `insert(`x`)` call is applied before it ends.*

Proof. For $k = 1$, the lemma holds as the value $P.t$ is incremented by one for each `insert(`x`)` call. For $k > 1$, say that a process *id* calls `insert(`x`)`. The register C is only modified in the Lines 19 and 26. W.l.o.g. assume that the process *id* invokes the

Algorithm 3: The insert(x) function.

```
1  insert($x$)
2      Let $I$ be the ID set of the processes that might call this
         function and $k = |I|$;
3      if $k = 1$ then
           /* Single process implementation */
           /* Symbol | is used as a field separator */
4          $e|t \leftarrow P$.read();
5          $P$.write($x|t + 1$);
6          return $t + 1$;
7      else
8          Split $I$ into two halves $L$ and $R$ so that $|L| \leq k/2$ and
             $|R| \leq k/2$;
9          Let $id$ be the ID of the process calling this function;
           /* store the value from the insert($x$)
              operation in $r$ */
10         if $id \in L$ then
11             $r \leftarrow C_l$.insert($x$);
12         else /* $id \in R$ */
13             $r \leftarrow C_r$.insert($x$);
           /* $\widehat{R}$ denotes a local copy of a register $R$
              */
14         $\widehat{C} \leftarrow C$.read();
15         $t_l \leftarrow C_l$.total();
16         $t_r \leftarrow C_r$.total();
17         log($\widehat{C}$);
18         $\widehat{C}' \leftarrow$ new_value($\widehat{C}, t_l, t_r$);
19         $success \leftarrow C$.compare-and-swap($\widehat{C}$,$\widehat{C}'$);
20         if $success$ = false then
21             $\widehat{C} \leftarrow C$.read();
22             $t_l \leftarrow C_l$.total();
23             $t_r \leftarrow C_r$.total();
24             log($\widehat{C}$);
25             $\widehat{C}' \leftarrow$ new_value($\widehat{C}, t_l, t_r$);
26             $C$.compare-and-swap($\widehat{C}$,$\widehat{C}'$);
27         log($C$.read());
28         return lookup($r$, $id \in L$);
```

insert(x) operation on C_l. Let it be the j^{th} insert operation on C_l. It follows from the definition of the set object C_l that the value $t_l \geq j$ for both the Lines 15 and 22. Thus, if the compare-and-swap operation in Lines 19 or 26 succeed, then the insert(x) call is applied.

Now, consider the case when the compare-and-swap operation in both the Lines 19 and 26 fail. As it failed in Line 19, a successful compare-and-swap operation was executed on the register C between the Lines 14 and 19 by some other process id'. Thus, the Line 15 of the process id' may or may not have executed after the process id executed C_l.insert(x). If it was after, then $t_l \geq j$ and we

Algorithm 4: The total() and new_value($\widehat{C}$, t_l, t_r) function.

```
1  total()
2      Let $I$ be the ID set of the processes that might call this
         function and $k = |I|$;
3      if $k = 1$ then
4          $e|t \leftarrow P$.read();
5          return $t$;
6      else
7          $\widehat{C} \leftarrow C$.read();
8          return $\widehat{C}.l_2 + \widehat{C}.r_2$;
9  new_value($\widehat{C}$, $t_l$, $t_r$)
       /* Symbol | is used as a field separator */
10     return $\widehat{C}.l_2|\widehat{C}.r_2|t_l|t_r$;
```

are done. Otherwise, we know that the compare-and-swap operation in Line 26 also fails. Thus, a compare-and-swap operation from some other process id'' executed successfully between the Lines 21 and 26. Thus, the line $t_l \leftarrow C_l$.total() from process id'' executes after the successful compare-and-swap operation from the process id' and thus, after the process id executed C_l.insert(x). Therefore, it holds that $t_l \geq j$ for the process id'' and the insert(x) call is applied. □

In fact, if we ignore the return value of insert(x) call for the time being, we can also show that the calls to insert(x) and total() are linearizable.

Lemma 3. *If we ignore the return value from* insert(x)*, then the functions* total() *and* insert(x) *are linearizable.*

Proof. For $k = 1$, we have a single process and the calls are trivially linearizable. For $k > 1$, let the linearization point of total() be Line 7. Using Lemma 2, we know that every insert(x) call is applied before it ends. Thus, there exists a compare-and-swap operation that applies the insert(x) call. Let the linearization point of the insert(x) call that is applied be the compare-and-swap operation that applied that call. The linearization point of insert(x) call that is not applied is after all the previous linearization points. Let us call the order of linearization points of insert(x) and total() calls as LO and the same order of the corresponding calls as CO.

As all the linearization points are assigned within the duration of their respective calls, the order defined by CO extends the partial order of insert(x) and total() calls. If multiple calls are assigned the same linearization point (for example, multiple insert(x) calls applied by the same compare-and-swap operation), then we assign them some total order relative to each other not changing their order relative to the other calls in CO. So, we have a total order CO on insert(x) and total() calls that extends their partial order.

Now, we need to check if the return value of these calls is consistent with the order CO. As the return value of the insert(x) calls is not the subject of this lemma, we ignore them. Now, consider a total() call c in the order CO and the corresponding linearization point p in the order LO. Let j be the number of insert() calls before c in CO. Then, there are j linearization points before p in LO

and j `insert(x)` calls were applied until p. Thus, the `total()` call c returns at least j. In fact, it returns exactly j as otherwise more than j `insert(x)` calls are applied until p and there are more than j calls until c in CO, a contradiction. □

But, our `insert(x)` call should also return values. The set object specification demands that these be distinct. We know from the previous lemmas that a successful `compare-and-swap` operation on the register C applies a bunch of `insert(x)` calls. If the register C was updated with the fields $l_1|r_1|l_2|r_2$, then a total of $(l_2 + r_2 - l_1 - r_1)$ `insert(x)` calls were applied. Of these, there were $l_2 - l_1$ `insert(x)` calls that executed C_l`.insert(x)` and $r_2 - r_1$ `insert(x)` calls that executed C_r`.insert(x)`. The range of return values for these calls should be $t_1 + 1, t_1 + 2, \ldots, t_2$ where $t_1 = (l_1 + r_1)$ and $t_2 = (l_2 + r_2)$. We choose to assign the first $l_2 - l_1$ values of the range to the `insert(x)` calls that executed C_l`.insert(x)` and the remaining $r_2 - r_1$ values to the `insert(x)` calls that executed C_r`.insert(x)`.

The problem in assigning these values is that not all the processes whose `insert(x)` calls are applied might get a chance to read the register C before it is updated with a new value. Therefore, we had the `log` function in Lines 17, 24 and 27 that stored the contents of the register C before updating it. Algorithm 5 shows a way to log these values so that they can be looked up later using return value of C_l`.insert(x)` or C_r`.insert(x)`. The values stored in the array L are used by the `insert(x)` calls that executed C_l`.insert(x)`, the ones stored in array R are used by the `insert(x)` calls that executed C_r`.insert(x)`. The array T stores the information to lookup the set object (C_l or C_r) on which the `insert(x)` operation was executed given the return value of an `insert(x)` call. This array is used by the `remove(i)` function.

Algorithm 5: The `log(`$\widehat{C}$`)` and `lookup(`r, inL`)` function.

```
1  log($\widehat{C}$)
2      $l_1|r_1|l_2|r_2 \leftarrow \widehat{C}$;
3      $T[l_2 + r_2]$.write($\widehat{C}$);
4      if $l_2 \neq l_1$ then
5          $L[l_2]$.write($\widehat{C}$);
6      if $r_2 \neq r_1$ then
7          $R[r_2]$.write($\widehat{C}$);
8  lookup($r$, $inL$)
9      $s \leftarrow r$;
10     if $inL$ = true then
11         while $L[s]$.read() = $\perp$ do
12             $s \leftarrow s + 1$;
13         $l_1|r_1|l_2|r_2 \leftarrow L[s]$.read();
14         return $(l_1 + r_1) + (r - l_1)$;
15     else
16         while $R[s]$.read() = $\perp$ do
17             $s \leftarrow s + 1$;
18         $l_1|r_1|l_2|r_2 \leftarrow R[s]$.read();
19         return $(l_1 + r_1) + (l_2 - l_1) + (r - r_1)$;
```

The following lemma shows that using these functions we get a linearizable implementation of the `insert(x)` function.

Lemma 4. *Algorithm 3 is a linearizable implementation of the* `insert(x)` *function.*

Proof. For $k = 1$, we have a sequential implementation as per the specification. For $k > 1$, let the linearization point of every `insert(x)` call that is applied be the `compare-and-swap` operation that applies it. Using Lemma 2, this `compare-and-swap` operation occurs within the duration of the `insert(x)` call. The linearization points of the `insert(x)` that are not applied are at the end of the execution in some arbitrary order. The resulting order of LO of linearization points defines an order CO of corresponding `insert(x)` calls, which extends their partial order.

Consider a `compare-and-swap` operation that applies a bunch of `insert(x)` calls. Say that it writes the value $l_1|r_1|l_2|r_2$ to the register C. It follows from the specification of the object C_l that the corresponding `insert(x)` operations on C_l (the ones that were executed by the applied `insert(x)` calls) returned values $l_1 + 1$, $l_1 + 2, \ldots, l_2$. Similarly, the corresponding `insert(x)` operations on C_r returned values $r_1 + 1, r_1 + 2, \ldots, r_2$. To extend the order CO into a total order, we order the bunch of `insert(x)` calls that were applied together by first ordering the `insert(x)` calls that executed C_l`.insert(x)` before the ones that executed C_r`.insert(x)` and then in the order of return values of the `insert(x)` operations executed by these calls. The updated order CO still extends the partial order of `insert(x)` calls.

Now, consider the j^{th} `insert(x)` call in the order CO. Say that the `compare-and-swap` operation that applies the `insert(x)` call writes $l_1|r_1|l_2|r_2$ to the register C and the `insert(x)` call executed C_l`.insert(x)`. It follows from the definition of the order CO that C_l`.insert(x)` returned $(j - r_1)$. Using Lemma 2, the `insert(x)` call executes the `compare-and-swap` operation of Line 26 at the linearization point or afterwards. Thus, the `insert(x)` call gets to execute the `log` function in Line 27 before it ends and after it is applied. If the value of the register C does not change until that function is executed, then the value $l_1|r_1|l_2|r_2$ is written to the register $L[l_2]$. If it does, then between the linearization point and the `log` function in Line 27 there was a successful `compare-and-swap` operation from another process. This process would again write the value $l_1|r_1|l_2|r_2$ to the register $L[l_2]$ before updating the register C.

Thus, the `lookup` function in Line 28 will find the non-empty value $l_1|r_1|l_2|r_2$ at index l_2 of L when searching forwards from index l_1. Thus, the value $(l_1 + r_1) + ((j - r_1) - l_1) = j$ is returned (Line 14 of Algorithm 5). By an argument along the same lines, it also holds that the return value is also j when the `insert(x)` call executes C_r`.insert(x)` operation. □

Note that not every slot in the array L or R is occupied. So, it could take sometime before the lookup function finds a non-empty slot. As there can be at most k `insert(x)` calls that aren't applied yet at any given moment (one per process), a single `compare-and-swap` operation applies at most k of them. Thus, the `lookup` function could take $O(k)$ steps in the worst case before it finds the next non-empty slot. But, we can improve this by patching this $O(k)$ empty slots with equally spaced $O(\sqrt{k})$ non-empty ones. Algorithm 6 shows the modified log function.

Algorithm 6: The optimized log($\widehat{C}$) function.

```
log($\widehat{C}$)
    $l_1|r_1|l_2|r_2 \leftarrow \widehat{C}$;
    update($T$, $l_1 + r_1$, $l_2 + r_2$, $\widehat{C}$);
    if $l_2 \neq l_1$ then
        update($L$, $l_1$, $l_2$, $\widehat{C}$);
    if $r_2 \neq r_1$ then
        update($R$, $r_1$, $r_2$, $\widehat{C}$);
update($A$, $i_1$, $i_2$, $\widehat{C}$)
    $i \leftarrow i_1 + \lfloor\sqrt{k}\rfloor$;
    while $i \leq i_2$ do
        $A[i]$.write($\widehat{C}$);
        $i \leftarrow i + \lfloor\sqrt{k}\rfloor$;
    $A[i_2]$.write($\widehat{C}$);
```

The following lemma shows that the insert(x) calls are linearizable even if we use the optimized log($\widehat{C}$) function.

Lemma 5. *Algorithm 3 using Algorithm 6 for* log($\widehat{C}$) *is a linearizable implementation of the* insert(x) *function and takes* $O(\sqrt{k})$ *steps.*

Proof. The correctness proof is same as in Lemma 4 except that the log function not only writes the value $l_1|r_1|l_2|r_2$ on $L[l_2]$ but also at $L[l_1 + i \cdot \lfloor\sqrt{k}\rfloor]$, $1 \leq i \leq l_2/\lfloor\sqrt{k}\rfloor$. The lookup still finds the value $l_1|r_1|l_2|r_2$ when searching forwards from index l_1.

At any given moment, there can be at most k insert(x) calls that are not finished (one per process). Using Lemma 2, there are at most k insert(x) calls that are not applied at any given moment. Thus, a single compare-and-swap operation can apply at most k insert(x) calls and $l_2 - l_1 \leq k$. Therefore, if we copy the value $l_1|r_1|l_2|r_2$ at equal distance of $O(\sqrt{k})$ between the indices l_1 and l_2 (log function), then the lookup functions finds the value in at most $O(\sqrt{k})$ steps. The log function also takes $O(\sqrt{k})$ steps and everything else in the insert(x) call takes $O(1)$ steps. Thus, the insert(x) call takes $O(\sqrt{k})$ steps. □

To complete our implementation, we also need the remove(i) function. The idea is to lookup the array T to find whether to execute C_l.remove(i) or C_r.remove(i). Algorithm 7 gives the implementation of the remove(i) function. The call log($\widehat{C}$) in Line 12 just ensures that the latest update to the register C was logged.

Note that we did not define the return value in case the argument i is even greater than the number of insert(x) operations performed on the counting set. This is not a problem as we never make such a call. In the following lemma, we show that the remove(i) function is linearizable. We will assume that the remove(i) function is called only after the insert(x) function that returned i is applied. We will prove this assumption to be true later when we use the counting set.

Algorithm 7: The remove(i) function.

```
1  remove($i$)
2      Let $k$ be the number of processes that might call this
         function;
3      if $k = 1$ then
4          $e|t \leftarrow P$.read();
5          if $i < t$ then
6              return $\perp$;
7          else
8              $P$.write($\perp|t$);
9              return $e$;
10     else
11         $\widehat{C} \leftarrow C$.read();
12         log($\widehat{C}$);
13         $h \leftarrow i$;
14         while $T[h]$.read() $= \perp$ do
15             $h \leftarrow h + 1$;
16         $l_1|r_1|l_2|r_2 \leftarrow T[h]$.read();
17         if ($l_1 + r_1 < i \leq (l_1 + r_1) + (l_2 - l_1)$) then
               /* The insert($x$) call that returned $i$
                  executed $C_l$.insert($x$) */
               /* Determine the value returned by the
                  operation $C_l$.insert($x$) */
18             $i' \leftarrow i - (l_1 + r_1) + l_1$;
19             return $C_l$.remove($i'$);
20         else
               /* The insert($x$) call that returned $i$
                  executed $C_r$.insert($x$) */
               /* Determine the value returned by the
                  operation $C_r$.insert($x$) */
21             $i' \leftarrow i - (l_1 + r_1) - (l_2 - l_1) + r_1$;
22             return $C_r$.remove($i'$);
```

Lemma 6. *The* remove(i) *function from Algorithm 7 is linearizable and takes* $O(\sqrt{k})$ *steps.*

Proof. For $k = 1$, we have a sequential implementation that is as per the specification. For $k > 1$, let the linearization point of every remove(i) call be Line 11. The order of these linearization points, along with those defined previously for insert(x) and total() calls, defines a total order CO on the calls that extends their partial order.

As per the usage constraints, the i^{th} insert(x) call in CO is applied before the linearization point of a call remove(i). Say, that the i^{th} insert(x) call executed the operation C_r.insert(x) and the register C was updated to $l_1|r_1|l_2|r_2$ when it was applied. From the definition of the linearization points of insert(x) calls, the operation C_r.insert(x) returned $i - (l_1 + r_1) - (l_2 - l_1) + r_1$. Thus, if the remove($i$) call finds the value $l_1|r_1|l_2|r_2$ in Line 16, it will return the correct element in Line 21.

There could be a successful `compare-and-swap` operation on the register C between the linearization point of the i^{th} `insert(`x`)` call and the linearization point of the `remove(`i`)` call. In that case, the value $l_1|r_1|l_2|r_2$ is written to the register $T[l_2 + r_2]$ and also to the registers $T[(l_1 + r_1) + j \cdot \lfloor\sqrt{k}\rfloor]$, $1 \le j \le (l_2 + r_2)/\lfloor\sqrt{k}\rfloor$. If there was no such `compare-and-swap` operation, then the `remove(`i`)` call does the same by calling `log(`$\widehat{C}$`)` in Line 12. Again, as at most k `insert(`x`)` calls are applied by a single `compare-and-swap` operation, we have $(l_2 + r_2) - (l_1 + r_1) \le k$. Thus, the loop of Line 14 will find the value $l_1|r_1|l_2|r_2$ in $O(\sqrt{k})$ steps. The argument is similar for the case when the i^{th} `insert(`x`)` call executes C_l`.insert(`x`)`. □

5.2 Analysis

Note that we still do not yet have an implementation using register operations. We need to use the previous implementations recursively to get down from n processes to 1 process. The following lemma shows it how.

Lemma 7. *There is a linearizable and wait-free implementation of the counting set using register operations where the* `insert(`x`)` *and* `remove(`i`)` *calls take $O(\sqrt{n})$ steps each and the* `total()` *call takes $O(1)$ steps.*

Proof. Let I be an implementation that consists of Algorithm 3, Algorithm 4, Algorithm 6, Algorithm 7 and the `lookup` function from Algorithm 5. Assume that we have a wait-free and linearizable implementation $I_{k/2}$ of the counting set using register operations for $k/2 \ge 1$ processes.

We know from Lemma 3, Lemma 5 and Lemma 6 that the implementation I is linearizable. So, we can use Lemma 1 to replace the operations on the objects C_l and C_r in the implementation I with calls to corresponding functions of implementation $I_{k/2}$ and we get a linearizable implementation using register operations for k processes.

The number of steps taken by the `insert(`x`)` and `remove(`i`)` calls in the resulting implementation can be expressed by the recurrence $T(k) = T(k/2) + O(\sqrt{k})$. Solving this for $k = n$ and $T(1) = O(1)$ gives $T(n) = O(\sqrt{n})$. The `total()` operation is not recursive and is in $O(1)$. Thus, the resulting implementation is also wait-free. □

If m is the total number of `insert(`x`)` operations performed on the counting set, then we require the arrays T, L and R to have $O(m)$ slots. The space requirements can be expressed by the recurrence $T(n) = O(m) + 2T(n/2)$ where $T(1) = O(m)$. This gives $T(n) = O(nm)$. Also, we have that the size of the registers C, T, L and R should be large enough to hold the value m. If we assume that the element x or 'pointer' to it in the call `insert(`x`)` can be stored in $O(\log n)$ bits, we can state the following corollary.

Corollary 8. *The counting set in Lemma 7 can be implemented using $O(nm)$ registers of size $O(\max\{\log m, \log n\})$ bits each where m is a bound on the number of* `insert(`x`)` *calls.*

6 THE QUEUE

The queue is implemented as an array. The `enqueue(`x`)` call tries to write an element at the head and the `dequeue()` call removes the element at the tail if the queue is non-empty. To manage the tail and the head indices, we store them in a single register TH and use the operations `half-increment()` and `half-max(`i`)` on the register. Algorithm 8 specifies the effect of these operations if the value of the first field in the TH register is t and the value of the second field is h.

Algorithm 8: Effect of operations `half-max(`i`)` and `half-increment()` on a register TH having a value $t|h$.

```
half-increment()
    /* r stores the return value */
    $r \leftarrow -1$;
    if $t \le h$ then
        $r \leftarrow t$;
        $t \leftarrow t + 1$;
    return $r$;
half-max($i$)
    $h \leftarrow \max\{h, i\}$;
```

Algorithm 9 gives the implementation of the queue using the TH register, a counting set object S and an array A to store the queue elements. The register TH is initialized to $1|0$. The set S is empty initially. The `enqueue(`x`)` call starts by inserting the element into the counting set and gets a slot number in return (Line 12). It then stores the element in the queue array at the assigned slot (Line 13), removes the element from the counting set to indicate that it is done (Line 14) and updates the head (Line 15). The `dequeue()` call gets the next available slot number if the queue is non-empty (Line 2). The call then removes the element from the counting set using the slot number (Line 6). If the remove operation on the counting set returns an element, the `dequeue()` call returns it (Line 8). Otherwise, the corresponding `enqueue(`x`)` call already stored the element in the queue array and the `dequeue()` call can read and return the element from the queue array (Line 10).

Algorithm 9: The `dequeue()` and `enqueue(`x`)` function.

```
1  dequeue()
2      $i \leftarrow TH$.half-increment();
3      if $i = -1$ then
4          return $i$;
5      else
6          $x \leftarrow S$.remove($i$);
7          if $x \ne \perp$ then
8              return $x$;
9          else
10             return $A[i]$.read();
11 enqueue($x$)
12     $i \leftarrow S$.insert($x$);
13     $A[i]$.write($x$);
14     $S$.remove($i$);
15     $TH$.half-max($i$);
```

Before we show the correctness, we first prove the following assumption about the usage of the `remove(`i`)` function.

Lemma 9. *It holds for Algorithm 9 that* `remove(`i`)` *is only called after the* `insert(`x`)` *call that returns i is applied.*

Proof. The `remove(`i`)` operation is called either in Line 6 or Line 14. Clearly, the claim holds for the call in Line 14 as the corresponding `insert(`x`)` call is finished after Line 12 and hence applied using Lemma 2. For the call in Line 6, $i \neq -1$ at that point. Thus, we have that $TH.h \geq i$. This implies that there was an `enqueue(`x`)` call that executed `half-max(`j`)` where $j \geq i$. Thus, a call S`.insert(`x`)` returned a value $j \geq i$ and was applied using Lemma 2. Thus, the `insert(`x`)` call that returned i was also applied. □

Lemma 10. *Algorithm 9 is a linearizable implementation of a queue.*

Proof. Let every completed `enqueue(`x`)` call be linearized at the point of execution of Line 12. If the `enqueue(`x`)` call is incomplete, let i be the value returned in Line 12. In case $TH.h \geq i$ at the end of the execution, the incomplete `enqueue(`x`)` call is again linearized at Line 12. Otherwise, the incomplete `enqueue(`x`)` call is linearized after all other calls. Let every `dequeue()` call be linearized at the point of execution of Line 2. The resulting order LO of linearization points defines a total order CO on the corresponding calls that extends their partial order.

We now prove that the return values of the calls is same as per the order CO. As an `enqueue(`x`)` call does not return anything, there is nothing to check. Consider a `dequeue()` call and its linearization point p in the order LO. Let $t|h$ be the value of TH register before the linearization point p (before Line 2 is executed). There are two cases depending on whether the call returns -1 or a value from Line 6 or Line 10.

Say that the call returns -1. Then, an `enqueue(`x`)` call wrote the value h to the second field of the register TH in Line 15. As S`.insert(`x`)` in Line 12 returns a different value each time, there are h `enqueue(`x`)` linearization points in LO before p. As the `dequeue()` call returns -1, we have $t > h$. Moreover, there are exactly h `dequeue()` linearization points until p so that the corresponding calls returned an element from Line 6 or Line 10. If the element returned was from Line 6, then it was non-empty ($\neq \perp$). If it was returned from Line 10, then it received an empty response ($= \perp$) in Line 6 because some other call already removed that element from S. That call can only be an `enqueue(`x`)` call because only one `dequeue()` call executes S`.remove(`i`)` in Line 6 for a fixed i. And, an `enqueue(`x`)` call removes an element from S in Line 14 only after writing it to the array A in Line 13. Thus, an element returned from Line 10 of a `dequeue()` call was also non-empty ($\neq \perp$). Thus, there are h `dequeue()` linearization points before p in the order LO so that the corresponding `dequeue()` calls returned a non-empty value. Therefore, the return value is also -1 as per the order CO.

Now, consider the other case when the `dequeue()` call returns an element y from Line 6 (S`.remove(`t`)`) or Line 10 ($A[t]$`.read()`). As argued above, we have $y \neq \perp$. Moreover, there are $t-1$ linearization points of `dequeue()` calls before p that returned a non-empty element. Thus, we have to show that `enqueue(`y`)` call is the t^{th} one in the order CO. If the call returns S`.remove(`t`)`, then this value was inserted by an operation S`.insert(`x`)` that returned t because of the counting set specification. Thus, this was executed by an `enqueue(`x`)` call that is the t^{th} one in CO and $x = y$. If the `dequeue()` call returns $A[t]$`.read()`, then S`.remove(`t`)` was executed by some other call c. This cannot be another `dequeue()` call as S`.remove(`i`)` is called only by the `dequeue()` call that receives i in Line 2. Thus, c is an `enqueue(`x`)` call. Therefore, c executed $A[t]$`.write(`x`)` and received t in Line 12. Thus, `enqueue(`x`)` call c is the t^{th} one in CO and $x = y$. □

As before, we can use Lemma 1 to replace the operations on the set object with calls to its linearizable implementation. Note that the `enqueue(`x`)` and `dequeue()` functions execute $O(1)$ steps besides the counting set operations. Thus, we have the following corollary.

Corollary 11. *Algorithm 9 gives a wait-free and linearizable implementation of queue using register operations where* `enqueue(`x`)` *and* `dequeue()` *take* $O(\sqrt{n})$ *time steps. The implementation requires* $O(nm)$ *registers of size* $O(\max\{\log n, \log m\})$ *bits each where m is a bound on the total number of* `enqueue(`x`)` *calls.*

7 THE CONSENSUS NUMBERS

Here, we show that the consensus number of the `half-max(`i`)` operation is one and the consensus number of the `half-increment()` operation is two. We first show that we can solve consensus among one process using the `half-max(`i`)` operation and two processes using the `half-increment()` operation. The case for the `half-max(`i`)` operation is trivial as every operation can solve consensus among one process (itself) by just deciding on its input value. For the `half-increment()` operation, we initialize a register R to the value $0|2$. Each process then announces its input value and executes the `half-increment()` operation on the register R. If the operation returns 0, the process decides on its input value. If the operation returns 1, the process decides on the input value of the other process. Both the processes decide on the same input value as the semantics of the `half-increment()` operation ensure that only one process gets the return value of 0. To show that the `half-max(`i`)` operation cannot solve consensus among two processes and that the `half-increment()` operation cannot solve consensus among three processes, we use the standard indistinguishability arguments.

Let us first define some terms. The *configuration* of the system is the value of the local variables of each process and the value of shared registers. The *initial* configuration consists of the initial value of the shared registers and the input value of 0 or 1 for each process. A given configuration is called a *bivalent* configuration if there are two possible executions from the configuration so that in one of the executions all the processes terminate and decide 1 and in the other all the processes terminate and decide 0. A configuration is called 0*-valent* if in all the possible executions from the configuration, all the processes terminate and decide 0. Similarly, we define a 1*-valent* configuration. A configuration is called *univalent* if it is either 0-valent or 1-valent. A configuration is called *critical* if it is bivalent and a step by any process changes it to a univalent configuration. The initial configuration is bivalent as a process with input 1 will decide 1 if it is the only process that takes steps, and a process with input 0 will decide 0 if it is the only process that takes steps. As the terminating state is univalent, every correct consensus algorithm must reach a critical configuration. Now, we can show the respective consensus numbers by contradiction.

Consider the `half-max(i)` operation. Assume that it can solve consensus for processes A and B. Then, a critical configuration c is reached. Say that the next step s_A by the process A gives a 0-valent configuration c_0. Then, the next step s_B by the process B gives a 1-valent configuration c_1. We have the following cases.

(1) The steps s_A and s_B are operations on different registers: In this case, the configurations obtained by taking the step s_B on c_0 and by taking the step s_A on c_1 are indistinguishable to both A and B. Thus, A running alone on these configurations decides the same value, a contradiction.

(2) One of the steps s_A or s_B is a read operation: Say s_A is the read operation. Then, the configuration obtained by taking the step s_B on c_0 is indistinguishable from c_1 to the process B. The reason is that the read operation s_A only changes A's local state. Thus, B running alone on these configurations decides the same value, a contradiction.

(3) The steps s_A and s_B are `half-max(i)` operations on the same register: In this case, the configurations obtained by taking the step s_B on c_0 and by taking the step s_A on c_1 are indistinguishable to both A and B as the `half-max(i)` operation is commutative and does not return anything. Thus, A running alone on them decides the same value, a contradiction.

Therefore, the critical configuration c cannot be reached and the operation `half-max(i)` cannot solve consensus for two processes.

Let us now consider the `half-increment()` operation. Assume that it can solve consensus for three processes A, B and C. Then, a critical configuration c is reached. Say, that the next step s_A by the process A gives a 0-valent configuration c_0 and that the next step s_B by the process B gives a 1-valent configuration c_1. As per the reasoning in case of `half-max(i)` operations, the steps s_A and s_B cannot be operations to different registers. Also, neither of them can be a read operation. So, the only case left is that both s_A and s_B are `half-increment()` operations to the same register R. In this case, consider the configuration obtained by executing the step s_B on c_0 and the configuration obtained by executing the step s_A on c_1. Due to the specification of the `half-increment()` operation, the value of the register R is same in both these configurations. Thus, the process C running alone on these configurations decides the same value, a contradiction.

8 DISCUSSION

If we look back at our queue design, then we used registers supporting the compare-and-swap operation, and another register (the *TH* register) that supports the half-increment and half-max operations. The problem is that these low consensus number operations are not supported by the modern hardware. But, closely related operations fetch-and-increment and compare-and-swap-if-greater-than (max) are supported [10, 11]. It is an interesting question if there is a theoretical basis for choosing a certain set of low consensus number primitives to implement in hardware. In terms of space usage, we are conceptually using an infinite array (arrays T, L and R of the counting set can be merged with the queue array A). One can try to improve this as a future work by upgrading the logging mechanism to a dynamic one from a static one. However, our goal in this work was to show as a proof-of-concept that low consensus number primitives can also be fundamentally powerful for designing concurrent data structures. Moreover, the problem of designing a sublinear wait-free queue with unbounded concurrency seems hard even with infinite arrays. David's algorithm [3], for example, gives a bounded concurrency queue with infinite arrays.

Also, we could consider the register *TH* as a separate object and ask the question whether it is possible to implement such a object in sublinear time using only compare-and-swap operations? Or, by also using a breadth of other read-modify-write operations such as decrement, multiply, xor, etc., that are commonly supported in hardware. In general, we wonder whether we should focus on designing our concurrent data structures using a specific primitive with infinite consensus number (in particular `compare-and-swap`) or all the primitives that the modern hardware may provide, essentially ignoring their consensus numbers!

REFERENCES

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. 1995. Wait-free Made Fast. In *27th Annual ACM Symposium on Theory of Computing (STOC), Las Vegas, Nevada, USA.*

[2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. 2014. Tight Bounds for Asynchronous Renaming. *Journal of the ACM (JACM)* (2014).

[3] Matei David. 2005. A Single-Enqueuer Wait-Free Queue Implementation. In *18th International Conference on Distributed Computing (DISC), Cracow, Poland.*

[4] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. 1990. A Complexity-Based Hierarchy for Multiprocessor Synchronization: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC), Chicago, IL, USA.*

[5] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. 2007. SNZI: Scalable NonZero Indicators. In *26th Annual ACM Symposium on Principles of Distributed Computing (PODC), Portland, Oregon.*

[6] Faith Ellen and Philipp Woelfel. 2013. An Optimal Implementation of Fetch-and-Increment. In *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel.*

[7] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-efficient Wait-free Universal Construction. In *23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), San Jose, California, USA.*

[8] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1991).

[9] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1990).

[10] Hybrid Memory Cube Consortium 2015. *Hybrid Memory Cube Specification 2.1, Comparison Atomics, Page 68.* Hybrid Memory Cube Consortium.

[11] Intel 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, Section 8.1.2.2.* Intel.

[12] Prasad Jayanti and Srdjan Petrovic. 2005. Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks. In *25th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Hyderabad, India.*

[13] Prasad Jayanti and Srdjan Petrovic. Jul. Efficient and Practical Constructions of LL/SC Variables. In *22nd Annual Symposium on Principles of Distributed Computing (PODC), Boston, Massachusetts.*

[14] Pankaj Khanchandani and Roger Wattenhofer. 2017. Brief Announcement: Fast Shared Counting using O(n) Compare-and-Swap Registers. In *ACM Symposium on Principles of Distributed Computing (PODC), Washington, DC, USA.*

[15] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), San Antonio, TX, USA.*

[16] Maged M. Michael. 2004. Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS. In *18th International Symposium on Distributed Computing (DISC), Amsterdam, Netherlands.*

[17] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Shenzhen, China.*

[18] Jacob T. Schwartz. 1980. Ultracomputers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1980).

[19] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Barcelona, Spain.*