Reducing Compare-and-Swap to Consensus Number One Primitives

Pankaj Khanchandani
ETH Zurich, Switzerland
kpankaj@ethz.ch

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Abstract

The consensus number of an object is the maximum number of processes among which binary consensus can be solved using any number of instances of the object and read-write registers. Herlihy [6] showed in his seminal work that if an object has a consensus number of \( n \), then there is a universal construction for a wait-free and linearizable implementation of any non-trivial concurrent object or data structure that is shared among \( n \) processes. Thus, a synchronization object such as compare-and-swap with an infinite consensus number and the corresponding instruction can be viewed as “strong”. On the other hand, a synchronization object such as fetch-and-add with consensus number two and the corresponding fetch-and-add instruction can be viewed as “weak”.

Ellen et al. [2] observed recently that an object supporting two weak instructions can also achieve infinite consensus number like an object that supports one strong instruction. Using Herlihy’s universal construction, this implies that ignoring concerns about efficiency, one can design any concurrent data structure or algorithm using only weak instructions. However, is it possible that a combination of weak instructions is really powerful enough to efficiently replace a strong instruction, like compare-and-swap, without incurring a large overhead in time or space?

In this paper, we answer this question by giving an \( O(1) \) time wait-free and linearizable implementation of a compare-and-swap register shared among \( n \) processes using read-write registers and registers that support two synchronization primitives half-max and max-write, each having consensus number one. The size of the registers required is logarithmic in the length of the execution. Thus, any algorithm that solves some arbitrary synchronization problem using read-write and compare-and-swap registers can be transformed into an algorithm that has the same asymptotic time complexity and uses registers that are logarithmic in the length of the execution and only support consensus number one instructions.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases compare-and-swap, synchronization, consensus number

1 Introduction

Any multiprocessor chip needs to support some synchronization instructions, such as compare-and-swap or fetch-and-add, to coordinate among several concurrent processes that can take steps asynchronously at different rates. As it is not possible to support every other synchronization instruction on a multiprocessor, the choice of instructions to support is important. Herlihy [6] gave an elegant way to make such a choice based on consensus numbers. The consensus number of an object is defined as the maximum number of processes \( n \) among which binary consensus can be solved using any number of instances of the object.
and read-write registers. In binary consensus, each process is given an input of either 0 and 1. Each process must output the same value (agreement) within a finite number of its steps (termination) so that the output value is an input value of some process (validity).

Herlihy showed that objects of consensus number \( n \) can be used to construct a linearizable and wait-free implementation of any concurrent data structure or object, such as stacks or queues, shared among \( n \) processes. Linearizability implies that although each operation takes several steps to complete, it appears to take effect instantaneously at some point between its invocation and termination. The wait-free property implies that every process completes its operation within a finite number of its steps irrespective of the speed of other processes. As the compare-and-swap object or register has infinite consensus number, supporting compare-and-swap on a multiprocessor is a good and powerful choice. On the other hand, a fetch-and-increment object or register has a consensus number of two and is an inherently weak choice by itself.

Recently, Ellen et al. [2] observed that the above classification of synchronization instructions treats them as individual objects but in reality all the instructions supported by a multiprocessor can be applied on any register or memory location. They also give simple examples where two weak instructions can be combined on the same object to achieve infinite consensus number. This along with Herlihy’s universal construction implies that it is possible to construct any concurrent data structure or object by only using weak synchronization instructions. Although possible, such a construction would be inefficient both in time and space. It is reasonable to argue that the weak instructions are only powerful enough to solve consensus efficiently but not enough to efficiently replace a strong instruction in Herlihy’s hierarchy. In fact, in a followup work by Gelashvili et al. [4], the authors write the following:

“\textit{The practical question is whether we can really replace a compare-and-swap instruction in concurrent algorithms and data-structures with a combination of weaker instructions.”}

Note that when we refer to the consensus number of a synchronization instruction or a primitive, we refer to the consensus number of an object that supports two operations: the synchronization primitive and a read operation. It is essential that we also consider the read operation on the object, otherwise, arbitrarily powerful primitives that do not return any value would have consensus number one as there would be no way to read the object (for eg., a compare-and-swap primitive that does not return a value). Thus, consensus number one primitives are like read-write registers where the write operation is replaced with another weak “write-like” operation. The challenge is to come up with similar weak operations that can be combined to efficiently replace compare-and-swap.

In this paper, we show that it is possible to simulate a compare-and-swap register using a combination of weak instructions and the simulation is efficient both in space and time. Concretely, we introduce two consensus number one primitives half-max and max-write. We show that using read-write registers and registers that support half-max and max-write, we can construct a linearizable and wait-free implementation of a compare-and-swap register so that every compare-and-swap operation takes \( O(1) \) time. The size of the registers required is logarithmic in the length of the execution. The total number of registers required is \( O(n) \) where \( n \) is the number of processes. Thus, any \( O(T) \) algorithm using compare-and-swap and read-write registers can be transformed into an \( O(T) \) time algorithm that only uses consensus number one instructions on reasonably large registers. We also outline an extension for simulating \( m \) compare-and-swap registers where the total number of registers required is \( O(m + n) \).
2 Related Work

One of the most central questions in concurrent computing has been to quantify the power of synchronization instructions. Herlihy [6] originally defined the consensus number of an object as the maximum number of processes \( n \) that can solve consensus using a single instance of the object and any number of read-write registers. As a consequence of this definition, an object that has higher consensus number or is higher in the Herlihy’s hierarchy cannot be implemented using an object that has a lower consensus number or is lower in the Herlihy’s hierarchy. Jayanti [8] defined robustness of a hierarchy as the property that an object at a higher level in the hierarchy cannot be implemented using any number or combination of objects lower in the hierarchy. He gave an example of an object such that \( k \) instances of the object along with read-write registers can solve consensus for \( k + 1 \) processes. Thus, Herlihy’s hierarchy would not be robust if the consensus number definition is restricted to use only single objects.

A natural fix is to allow any number of instances of the object in the definition of consensus numbers, which is also the accepted definition and the one that we use [7]. Under this definition, Chandra et al. [1] show that Herlihy’s hierarchy is robust for two objects out of which one of is a consensus object and the other one is an arbitrary object. Ruppert [15] showed that Herlihy’s hierarchy is robust for read-modify-write and readable objects, which captures a large class of synchronization primitives. All these results assume that when a set of objects are used to implement another object, the synchronization operations supported by different objects are not merged onto a same object.

Ellen et al. [2] observed that if one relaxes the above assumption and does not treat a set of synchronization instructions as a set of individual objects but as a single object supporting the set of synchronization instructions, then Herlihy’s hierarchy is again not robust. They propose a space based hierarchy in which the power of set of synchronization instructions is quantified by the minimum amount of space required to solve obstruction free consensus among \( n \) processes. A small value of this quantity for a set of synchronization instructions means that the set of instructions is more powerful. This work has led to some more followup work to understand the power of a set of synchronization instructions from different perspectives when the instructions are assumed to be supported on the same register.

In [4], the authors give a lock-free implementation of a log data structure by only using x86 instructions of consensus number at most two. They report that the performance achieved was similar to that of a compare-and-swap based implementation. In our work, we do not restrict ourselves to instructions supported on modern architecture as our goal is to find if it is theoretically possible to efficiently compete with a strong instruction like compare-and-swap using low consensus number instructions only. In [13], we observed that a set of low consensus number instructions supported on the same register can help to improve the time bound of solving the fundamental synchronization task of designing a wait-free queue from \( O(n) \) to \( O(\sqrt{n}) \) for \( n \) processes.

In this paper, we look at the power of a set of low consensus number instructions supported on the same register with respect to their ability to efficiently simulate a strong instruction like compare-and-swap. We chose to simulate compare-and-swap not only because of its infinite consensus number but also because it is ubiquitous and has been shown to yield efficient implementations [14, 11, 10]. Our result then implies that a set of low consensus number instructions can be at least as powerful as compare-and-swap. In [5], the authors give a blocking implementation of comparison primitives by just using read-write registers and constant number of remote memory references. Their focus is to use read-write registers and
hence wait-freedom is impossible to achieve. Overall, there is no prior work that shows that a set of low consensus number instructions can be as powerful and efficient as compare-and-swap registers for an arbitrary synchronization task.

3 An Overview of the Method

Our method is based on the observation that if several compare-and-swap operations attempt to simultaneously change the value in the register, only one of them succeeds. So, instead of updating the final value of the register for each operation, we first determine the single operation that succeeds and update the final value accordingly. This is achieved by using two consensus number one primitives: max-write and half-max.

The max-write primitive takes two arguments. If the first argument is greater than or equal to the value in the first half of the register, then the first half of the register is replaced with the first argument and the second half is replaced with the second argument. Otherwise, the register is left unchanged. In any case, no value is returned. This primitive helps in keeping a version number along with a value.

The half-max primitive takes a single argument and replaces the first half of the register with that argument if the argument is larger. Otherwise, the register remains unchanged. Again, no value is returned in any case. This primitive is used along with the max-write primitive to determine the single successful compare-and-swap operation out of several concurrent ones. The task of determining the successful compare-and-swap operation can be viewed as a variation of tree-based combining (as in [3, 12] for example). The difference is that we do not use a tree as it would incur $\Theta(\log n)$ time overhead. Instead, our method does the combining in constant time as we will see later.

In the following section, we formalize the model and the problem. In Section 5, we give an implementation of the compare-and-swap operation using registers that support the half-max, max-write, read and write operations. In Section 6, we prove its correctness and show that the compare-and-swap operation runs in $O(1)$ time. In Section 7, we argue that the consensus numbers of the max-write and half-max primitives are both one. Finally, we conclude and discuss some extensions in Section 8.

4 Model

A sequential object is defined by the tuple $(S, O, R, T)$. Here, $S$ is the set of all possible states of the object, $O$ is the set of operations that can be performed on the object, $R$ is the set of possible return values of all the operations and $T : S \times O \rightarrow S \times R$ is the transition function that specifies the next state of the object and the return value given a state of the object and an operation applied on it.

A register is a sequential object and supports the operations read, write, half-max and max-write. The read() operation returns the current value (state) of the register. The write(v) operation updates the value of the register to v. The half-max(x) operation replaces the value in the first half of the register, say a, with max\{x, a\} and does not return any value. The max-write(x | y) operation replaces the first half of the register, say a, with x and second half of the register with y if and only if $x \geq a$. In any case, the operation does not return any value. The register operations are atomic, i.e., if different processes execute them simultaneously, then they execute sequentially in some order. In general, atomicity is implied whenever we use the word operation in the rest of the text.

An implementation of a sequential object is a collection of functions, one for each operation
defined by the object. A function specifies a sequence of instructions to be executed when the function is executed. An instruction is an operation on a register or a computation on local variables, i.e., variables exclusive to a process.

A process defines a sequence of instructions to be executed depending on the functions it executes. The processes have identifiers 1, 2, . . . , n. When a process executes a function, it is said to call that function. A schedule is a sequence of process identifiers. Given a schedule \( S \), an execution \( E(S) \) is the sequence of instructions obtained by replacing each process identifier in the schedule with the next instruction to be executed by the corresponding process.

Given an execution and a function called by a process, the start of the function call is the point in the execution when the first register operation of the function call appears. Similarly, the end of the function call is the point in the execution when the last register operation of the function call appears. A function call \( A \) is said to occur before another function call \( B \), if the call \( A \) ends before call \( B \) starts. Thus, the function calls of an implementation of an object \( O \) form a partial order \( P_O(E) \) with respect to an execution \( E \). An implementation on an object \( O \) is linearizable if there is a total order \( T_O(E) \) that extends the partial order \( P_O(E) \) for any given execution \( E \) so that the actual return value of every function call in the order \( T_O(E) \) is same as the return value determined by applying the specification of the object to the order \( T_O(E) \). The total order \( T_O(E) \) is usually defined by associating a linearization point with each function call, which is a specific point in the execution when the call takes effect. An implementation is wait-free if every function call returns within a finite number of steps of the calling process irrespective of the schedule of the other processes.

Our goal is to develop a wait-free and linearizable implementation of the compare-and-swap register. It supports read and compare-and-swap operations. The read() operation returns the current value of the register. The compare-and-swap\((a, b)\) operation returns true and updates the value of the register to \( b \) if the value in the register is \( a \). Otherwise, it returns false and does not change the value.

5 Algorithm

Figure 1 shows the (shared) registers that are used by the algorithm. There are arrays \( A \) and \( R \) of size \( n \) each. The \( i \)th entry of the array \( A \) consists of two fields: the field \( c \) keeps a count of the number of compare-and-swap operations executed by the process \( i \), the field \( val \) is used to store or announce the second argument of the compare-and-swap operation that the process \( i \) is executing. The \( i \)th entry of the array \( R \) consists of the fields \( c \) and \( ret \). The field \( ret \) is used for storing the return value of the \( c \)th compare-and-swap operation executed by the process \( i \). The register \( V \) stores the current value of the compare-and-swap object in the field \( val \) along with its version number in the field \( seq \). The fields \( seq \), \( pid \) and \( c \) of the register \( P \) respectively store the next version number, the process identifier of the process that executed the latest successful compare-and-swap operation and the count of compare-and-swap operations issued by that process. For all the registers, the individual fields are of equal sizes except for the register \( P \). The first half of this register stores the field \( seq \) where as the second half stores the other two fields, \( pid \) and \( c \).

Algorithm 1 gives an implementation of the compare-and-swap register. To execute the read function, a process simply reads and returns the current value of the object as stored in the register \( V \) (Lines 2 and 3). To execute the compare-and-swap function, a process starts by reading the current value of the object (Line 5). If the first argument of the function is not equal to the current value, then it returns false (Lines 6 and 7). If both the arguments are same as the current value, then it can simply return true as the new value is same as the
initial one (Lines 8 and 9).

Otherwise, the process competes with the other processes executing the compare-and-swap function concurrently. First, the process increments its local counter (Line 10). Then, the new value to be written by the process is announced in the respective entry of the array $A$ (Line 11) and the return value of the function is initialized to false by writing to the respective entry in the array $R$ (Line 12). The process starts competing with the other concurrent processes by trying to announce its identifier in $P$ using the max-write operation (Line 13). The competition is finished by writing a version number larger than used by the competing processes (Line 14).

**Algorithm 1:** The compare-and-swap and the read functions. The symbol $|$ is a field separator. The symbol $\_\_\_$ is a variable that is not used. The variable $id$ is the identifier of the process executing the function. At initialization, we have $c = 0$ and $V = (0 \mid x)$, where $x$ is the initial value of the compare-and-swap object.

```plaintext
1 read()
2 \hspace{1em} (_, | val) ← V.read();
3 \hspace{1em} return val;
4 compare-and-swap(a, b)
5 \hspace{1em} (seq | val) ← V.read();
6 \hspace{1em} if a ≠ val then
7 \hspace{2.5em} return false;
8 \hspace{1em} if a = b then
9 \hspace{2.5em} return true;
10 \hspace{1em} c ← c + 1;
11 \hspace{1em} A[id].write(c | b);
12 \hspace{1em} R[id].write(c | false);
13 \hspace{1em} P.max-write(seq + 1 | id | c);
14 \hspace{1em} P.half-max(seq + 2);
15 \hspace{1em} (seq | pid | cp) ← P.read();
16 \hspace{1em} (ca | val) ← A[pid].read();
17 \hspace{1em} if seq is even and cp = ca then
18 \hspace{2.5em} R[pid].max-write(ca | true);
19 \hspace{2.5em} V.max-write(seq | val);
20 \hspace{1em} (_, | ret) ← R[id].read();
21 \hspace{1em} return ret;
```

Once the winner of the competing processes is determined, the winner and the value announced by it is read (Lines 15 and 16), the winner is informed that it won after appropriate
checks (Lines 18, 17) and the current value is updated (Line 19). The value to be returned is then read from the designated entry of array $R$ (Line 20). A closer look at the algorithm reveals that the half-max and max-write operations are only combined on the register $P$. All other registers either only use max-write (and not half-max) or are only read-write registers.

In the following section, we analyze Algorithm 1 and show that it is a linearizable and $O(1)$ time wait-free implementation of the compare-and-swap object.

### 6 Analysis

Let us first define some notation. We refer to a field $f$ of a register $X$ by $X.f$. The term $X.f_k$ is the value of the field $X.f$ just after process $i$ executes Line $k$ during a call. We omit the call identifier from the notation as it will be always clear from the context. Similarly, $v^i_k$ is the value of a variable $v$, that is local to the process $i$, just after it executes Line $k$ during a call. The term $X.f_k$ is the value of a field $X.f$ at the end of an execution.

To prove that our implementation is linearizable, we first need to define the linearization points. The linearization point of the compare-and-swap function executed by a process $i$ is given by Definition 1. There are four main cases. If the process returns from Line 7 or Line 9, then the linearization point is the read operation in Line 5 as such an operation does not change the value of the object (Cases 1 and 2). Otherwise, we look for the execution of Line 19 that wrote the sequence number $V.seq_i + 2$ to the field $V.seq$ for the first time. This is the linearization point of the process $i$ if its compare-and-swap operation was successful as determined by the value of $P.pid$ (Case 3a). Otherwise, the failed compare-and-swap operations are linearized just after the successful one (Case 3b). The calls that have not taken effect are linearized after all other linearization points (Case 4).

#### Definition 1. The compare-and-swap call by a process $i$ is linearized as follows.

1. If $V.val_i \neq a_i^4$, then the linearization point is the point when $i$ executes Line 5.
2. If $V.val_i = a_i^4 = b_i^4$, then the linearization point is the point when $i$ executes Line 5.
3. If $V.val_i = a_i^4 \neq b_i^4$ and $V.seq_e \geq V.seq_i + 2$, then let $p$ be the point when Line 19 is executed by a process $j$ so that $V.seq_{19} = V.seq_i + 2$ for the first time.
   (a) If $pid_{19} = i$, then the linearization point is $p$.
   (b) If $pid_{19} \neq i$, then the linearization point is just after $p$.
4. If $V.val_i = a_i^4 \neq b_i^4$ and $V.seq_e < V.seq_i + 2$, then the linearization point is at the end, after all the other linearization points in some order.

Note that we assume in Case 3 that if $V.seq_e \geq V.seq_i + 2$, then there is an execution of Line 19 by a process $j$ with the value $V.seq_{19} = V.seq_i + 2$. So, we first show in the following lemmas that this is indeed true.

#### Lemma 2. The value of $V.seq$ is always even.

**Proof.** We have $V.seq = 0$ at initialization. The modification only happens in Line 19 with an even value. ▷

#### Lemma 3. Whenever $V.seq$ changes, it increases by 2.

**Proof.** Say that the value of the field was changed to $V.seq_i$ when a process $i$ executed Line 19. Then, the value $seq_{17}$ is even and so is the value $V.seq_{19}$. Thus, the value $V.seq_{19}$ was written to $P.seq$ by a process $j$ and that $V.seq_j = V.seq_{19} - 2$. As the field $V.seq$ is only modified by a max-write operation so it only increases. Thus, we have $V.seq \geq V.seq_{19} - 2$ just before $i$ modifies it. As $V.seq$ is even by Lemma 2 and $i$ modifies it, we have $V.seq = V.seq_{19} - 2$ before the modification. So, the value increases by 2. ▷
Lemma 4. The linearization point as given by Definition 1 is well-defined.

Proof. The linearization point as given by Definition 1 clearly exists for all the cases except for Case 3. For Case 3, we only need to show that if $V.seq_e \geq V.seq_1^i + 2$, then there exists an execution of Line 19 by a process $j$ so that $V.seq_1^j = V.seq_1^i + 2$. As $V.seq_1^i$ is even by Lemma 2 and the value of $V.seq$ only increases in steps of 2 by Lemma 3, it follows from $V.seq_e \geq V.seq_1^i + 2$ that $V.seq_1^j + 2$ was written to $V.seq$ at some point.

To show that the implementation is linearizable, we need to prove two main statements. First, the linearization point is within the start and end of the corresponding function call. Second, the value returned by a finished call is same as defined by the sequence of linearization points up to the linearization point of the call. In the following two lemmas, we show the first of these statements.

Lemma 5. If the condition $V.val_5^i = a_4^i \neq b_4^i$ is true for a compare-and-swap call by a process $i$, then the value of $V.seq$ is at least $V.seq_1^i + 2$ at the end of the call.

Proof. We define a set of processes $S = \{ j : V.seq_1^j = V.seq_1^i \}$. Consider the process $k \in S$ that is the first one to execute Line 17. As the first field of $P.seq$ is always modified by a max operation and process $k$ writes $V.seq_1^k + 2$, then we have $seq_{17}^k = seq_{15}^k \geq V.seq_1^k + 2$. If $seq_{15}^k > V.seq_1^i + 2$, then $V.seq_{15}^i \geq V.seq_1^k + 2$ and we are done.

So, we only need to check the case when $seq_{15}^k = V.seq_1^i + 2$. As $V.seq_1^i$ is even by Lemma 2, so is $seq_{17}^k = seq_{15}^k$. Moreover, the process $pid_{15}^k \in S$ as some process(es) (including $k$) executed Line 13. As $A[pid_{15}^k].c$ always increases whenever modified (Line 11), we have $ca_{16}^k \geq cp_{15}^k$. But, if $ca_{16}^k > cp_{15}^k$, then the process $pid_{15}^k$ finished even before the process $k$, a contradiction. So, it holds that $ca_{16}^k = cp_{15}^k$ and the process $k$ executes Line 19.

Now, the execution of Line 19 by the process $k$ either changes the value of $V.seq$ or does not. If it does, then $V.seq_{15}^k = V.seq_1^i + 2$ and we are done. Otherwise, someone already changed the value of $V.seq$ to at least $V.seq_1^i + 2$ because of Lemma 3.

Lemma 6. The linearization point as given by Definition 1 is within the corresponding call duration.

Proof. The statement is true for Cases 1 and 2 as the instruction corresponding to the linearization point is executed by the process $i$ itself.

For Case 3, we analyze the case of finished and unfinished call separately. Say that the call is unfinished. As $V.seq_e \geq V.seq_1^i + 2$ and $V.seq_1^i$ is the value of $V.seq$ at the start of the call, the linearization point as given by Definition 1 is after the call starts. Now, assume that the call is finished. We know from Lemma 5 that the value of $V.seq$ is at least $V.seq_1^i + 2$ when the call ends. So, the point when Line 19 writes $V.seq_1^i + 2$ to $V.seq$ is within the call duration.

We know from Lemma 5 that if the call finishes, then we have $V.seq_e \geq V.seq_1^i + 2$. So, if $V.seq_e < V.seq_1^i + 2$, then the call is unfinished and it is fine to linearize it at the end as done for Case 4.

Now, we need to show that the value returned by the calls is same as the value determined by the order of linearization points. We show this in the following lemmas.

Lemma 7. Assume that $x = seq_{15}^i = seq_{15}^j$ for two distinct processes $i$ and $j$ and that $x$ is even. Then, it implies that $pid_{15}^i = pid_{15}^j$ and $cp_{15}^i = cp_{15}^j$. 

As the field seq can change until the process j executes Line 15, by a max-write operation on P with the value x as the first field. This is not possible as x is even and the max-write on P is only executed with odd value as the first field (Line 13). So, it holds that pid\(_{15}\) = pid\(_{15}\). Similarly, we have cp\(_{15}\) = cp\(_{15}\).  

\[\text{Lemma 8. As long as the value of V.seq remains same, the value of V.val does not change.}\]

\[\text{Proof. Say that a process i is the first one to write a value x to V.seq. The value written to the field V.val by the process i is } val_{i6}.\text{ To have a different value of V.val with x as the value of V.seq, another process j must execute Line 19 with seq\(_{15}\) = x but val\(_{16}\) = val\(_{16}\). As seq\(_{15}\) = x = seq\(_{15}\), it follows from Lemma 7 that pid\(_{15}\) = pid\(_{15}\) and cp\(_{15}\) = cp\(_{15}\). As the condition in Line 17 is true for both the processes i and j, it then follows that ca\(_{16}\) = ca\(_{16}\). As the field A[pid\(_{15}\)].val is updated only once for a given value of A[pid\(_{15}\)].c (Line 11), it holds that val\(_{16}\) = val\(_{16}\) and the claim follows.}\]

\[\text{Lemma 9. Say that seq\(_{15}\) = x is even and pid\(_{15}\) = j during a call by a process i, then V.seq\(_{i}\) = x - 2 for some call by process j.}\]

\[\text{Proof. As seq\(_{15}\) = x, some process h modified P by executing Line 13 or Line 14 with x as the first argument. As x is even and V.seq\(_{h}\) is even by Lemma 2, the process h modified P by executing Line 14. So, it holds that V.seq\(_{h}\) = x - 2. Also, process h executed Line 13 with x - 1 as the first field. As pid\(_{15}\) = j, the process j also executed Line 13 with x - 1 as the first field after the process h did so. So, it holds that V.seq\(_{i}\) = x - 2.}\]

\[\text{Lemma 10. For every even value x ∈ [2, V.seq\(_{i}\)], there is an execution of Line 19 by a process i so that seq\(_{15}\) = x and the first such execution is the linearization point of some call.}\]

\[\text{Proof. Consider an even value x ∈ [2, V.seq\(_{i}\)]. Then, we know from Lemma 3 that x is written to V.seq by an execution of Line 19. Let p be the point of first execution of Line 19 by a process j so that seq\(_{15}\) = x. So, it holds for the process pid\(_{15}\) = h that V.seq\(_{h}\) = x - 2 using Lemma 9. As point p is the first time when x is written to the field V.seq, it holds that V.seq\(_{19}\) = x. Thus, p is the linearization point of the process h by Definition 1.}\]

\[\text{Lemma 11. The value V.val is only modified at a Case 3a linearization point.}\]

\[\text{Proof. Let q be a Case 3a linearization point. Say that the value of V.seq is updated to x at q. Let p be the first point in the execution when the value of V.seq is x - 2. Using Lemma 10, we conclude that p is either a linearization point (for x - 2 ≥ 2) or the initialization point (for x - 2 = 0). Using Lemma 8, the value of V.val is not modified between p and q.}\]

We want to use the above lemma in an induction argument on the linearization points to show that the values returned by the corresponding calls are correct. First, we introduce some notation for \(k \geq 1\). The term \(L.val_{k}\) is the value of the abstract compare-and-exchange object after the \(k^{th}\) linearization point. The terms \(V.seq_{k}\) and \(V.val_{k}\), respectively, are the values of \(V.seq\) and \(V.val\) after the \(k^{th}\) linearization point. These terms refer to the respective values just after initialization for \(k = 0\). For \(k \geq 1\), the term \(L.val_{k}\) is the expected return value of the call corresponding to the \(k^{th}\) linearization point. The following two lemmas prove the correctness using induction on the linearization points and checking the different linearization point cases separately.
Lemma 12. After \( k \geq 0 \) linearization points, we have \( Lval_k = Vval_k \) except for Case 4 linearization points. For \( k \geq 1 \), the \( Lret_k \) values are false for Case 1, true for Case 2, true for Case 3a and false for Case 3b.

Proof. We prove the claim by induction on \( k \). For the base case of \( k = 0 \), the claim is true as \( V.val \) is initialized with the initial of the compare-and-swap object. Let \( LP_k \) be the \( k \)th linearization point for \( k \geq 1 \) and say that it corresponds to a call by a process \( i \). We have the following cases.

Case 1: Let \( LP_{k'} \) be the linearization point previous to \( LP_k \). By induction hypothesis, it holds that \( Lval_{k'} = Vval_{k'} \). By Lemma 11, the value of \( V.val \) does not change until \( LP_k \). As we have a read operation at \( LP_k \), it holds that \( V.val_{k'} = Vval_{k'} \). By Definition 1, we know that \( V.val_k \neq a'_i \). So, it holds that \( Lval_{k'} = Vval_{k'} = V.val_k \neq a'_i \). Thus, it follows from the specification of the compare-and-swap object that \( Lval_k = L.val_{k'} = V.val_k \). Moreover, we have \( Lret_k = false \) as \( L.val_{k'} = V.val_k \neq a'_i \).

Case 2: Again, we let \( LP_{k'} \) to be the linearization point previous to \( LP_k \). As argued in the previous case, it holds that \( V.val_{k'} = V.val_k \). By Definition 1, we know that \( V.val_k = a'_i = b'_i \). So, it holds that \( L.val_{k'} = V.val_{k'} = V.val_k = a'_i \). Thus, it follows from the object’s specification that \( L.val_k = b'_i = V.val_k \). Further, we have \( Lret_k = true \) as \( V.val_k = a'_i \).

Case 3a: Consider the point \( LP_{k'} \) when the value \( V.seq_k - 2 \) was written to \( V.seq \) for the first time. As \( V.seq \) is even by Lemma 2, it follows from Lemma 10 that \( LP_k \) is a linearization point or the initialization point. Using definition of Case 3a, \( LP_k \) is the first point when the value \( V.seq_k \) was written to the field \( V.seq \). So, we have \( V.seq_k = V.val_{k'} \). Thus, it holds that \( V.val_{k'} = V.val_k \) by Lemma 8. Therefore, \( V.val_{k'} = V.val_k \) as \( V.val_{k'} = V.val_k \) by induction hypothesis. Using definition of Case 3a, it also holds that \( a'_i = V.val_{k'} \). Thus, we have \( a'_i = L.val_{k'} \) and \( L.val_k = b'_i \).

Now, assume that the instruction at \( LP_k \) was executed by a process \( j \). Using definition of Case 3a, we have \( i = pid_{15} \). As \( LP_k \) is the first time when the value of \( V.seq \) is \( V.seq_k = V.seq_k + 2 \), we conclude that the process \( i \) is not finished until \( LP_k \) by using Lemma 5. As \( seq_{15} = V.seq_k = V.seq_k + 2 \), it is true that some process \( i' \) has \( V.seq_{k'} = V.seq_k \) and that the process executed Line 13 until \( LP_k \). As \( i = pid_{15} \), the process \( i' = i \). Moreover, the process \( i \) did this during the call corresponding to the linearization point \( LP_k \) as it follows from Lemma 5 that there is a unique call for any process \( h \) given a fixed value of \( V.seq_{k'}^h \). Thus, the process \( i \) already executed Line 11 with \( b'_i \) as the value of the second field. This field has not changed as the call by process \( i \) is not finished until \( LP_k \). So, we have \( val_{15}^{h'} = b'_i \) and that \( V.val_k = b'_i \) as well. Because \( a'_i = L.val_k \) as shown before, we also have \( Lret_k = true \).

Case 3b: Let \( LP_{k'} \) and \( LP_{k''} \) be the first points when the value \( V.seq_k \) and \( V.seq_k - 2 \) is written to \( V.seq \) respectively (\( LP_{k''} \) is just before the point \( LP_k \) as defined by Case 3b). Let \( i \) and \( j \) be the processes that execute the calls corresponding to the points \( LP_k \) and \( LP_{k'} \), respectively. By definition of Case 3b, we have \( V.seq_{k'} = V.seq_{k} - 2 \). As process \( j \) wrote \( V.seq_{k'} \) to \( V.seq \), we have \( V.seq_{k'} = V.seq_{k} - 2 \) as well. So, we have \( V.val_{k'} = V.val_k \) using Lemma 8. Using definition of Case 3a and Case 3b, respectively, we have \( a'_i = V.val_{k'} \neq b'_i \) and \( a'_i = V.val_{k'} \). So, we have \( a'_i \neq b'_i \). We have \( b'_i = L.val_{k'} \) as argued in the previous case, so it holds that \( L.val_k = L.val_{k'} \). By induction hypothesis, we have \( L.val_{k'} = V.val_{k'} \). Moreover, there are no operations after \( LP_{k'} \) and until \( LP_k \) by definition of Case 3b. So, we have \( V.val_{k'} = V.val_k \) and thus \( L.val_k = V.val_k \). Also, we have \( Lret_k = false \) as \( a'_i \neq b'_i = L.val_{k'} \). ▶
Lemma 13. If the \(k\)th linearization point for \(k \geq 1\) corresponds to a finished call by a process \(i\), then the value returned by the call is \(L.\text{ret}_k\).

Proof. Say the \(k\)th linearization point is a Case 1 point. Using its definition, the value returned by the corresponding call is \(false\) as the condition in Line 6 holds true. Using Lemma 12, we have \(L.\text{ret}_k = false\) as well for Case 1. Next, assume that the \(k\)th linearization point is a Case 2 point. Then, the value returned by the corresponding call is \(true\) as the condition in Line 8 is true by definition. Using Lemma 12, we have \(L.\text{ret}_k = true\) as well for Case 2.

Now, consider that the \(k\)th linearization point is a Case 3a point. Say that the process \(j\) executes the operation at the linearization point. As \(pid_{15}^j = i\) by definition of Case 3a, the process \(i\) already executed Line 13 with the first field as \(V.\text{seq}_k\) = 1. So, the process \(i\) also initialized \(R[i]\) to \((cp_{15}^i | false)\) in Line 12. Moreover, the process \(j\) wrote the value \((cp_{15}^j | true)\) to \(R[i]\) afterwards using a max-write operation. Thus, the value of \(R[i], \text{ret}\) after \(LP_k\) is \(true\). This field is not changed by \(i\) until it returns. And, other processes only write \(true\) to the field. So, the call returns \(true\) which is same as the value of \(L.\text{ret}_k\) given by Lemma 12.

Next, consider that the \(k\)th linearization point is a Case 3b point. Let \(p\) be the point when the process \(i\) initializes \(R[i]\) to a value \((x | false)\) during the call (Line 12). Consider a process \(j\) that tries to write \(true\) to \(R[i].\text{ret}\) after \(p\) (by executing Line 18). So, it holds that \(pid_{15}^j = i\) and that \(seq_{15}^j\) is even. Now, we consider three cases depending on the relation between \(seq_{15}^j\) and \(V.\text{seq}_k\). First, consider that \(seq_{15}^j > V.\text{seq}_k\). As \(pid_{15}^i = i\) and \(seq_{15}^j\) is even, we have \(V.\text{seq}_k = seq_{15}^j - 2\) using Lemma 9. So, we have \(V.\text{seq}_k > V.\text{seq}_k - 2\). This cannot happen until \(i\) finishes as \(V.\text{seq}_k = V.\text{seq}_k - 2\) for the current call by \(i\) using definition of Case 3b. Second, consider that \(seq_{15}^j = V.\text{seq}_k\). Using definition of Case 3b, there is a process \(h\) so that \(pid_{15}^h \neq i\) and \(seq_{15}^h = V.\text{seq}_k\). As \(seq_{15}^j = V.\text{seq}_k\) by assumption, we have \(pid_{15}^j \neq i\) using Lemma 7. This contradicts our assumption that \(pid_{15}^i = i\). Third, consider that \(seq_{15}^j < V.\text{seq}_k\). As \(pid_{15}^i = i\) and \(seq_{15}^i\) is even, we have \(V.\text{seq}_k = seq_{15}^i - 2\) using Lemma 9. So, we have \(V.\text{seq}_k < V.\text{seq}_k - 2\). This corresponds to a previous call by the process \(i\) as \(V.\text{seq}_k = V.\text{seq}_k - 2\) for the current call by \(i\). So, it holds that \(ca_{15}^i < x\) and execution of Line 18 has no effect. Thus, the process \(i\) returns \(false\) for Case 3b which matches the \(L.\text{ret}_k\) value given by Lemma 12.

If the \(k\)th linearization point is a Case 4 point, then we know from Lemma 5 that the call is unfinished and we need not consider it. ▶

We can now state the following main theorem about Algorithm 1.

Theorem 14. Algorithm 1 is a wait-free and linearizable implementation of the compare-and-swap register where both the compare-and-swap and read functions take \(O(1)\) time.

Proof. We conclude that the compare-and-swap function as given by Algorithm 1 is linearizable by using Lemma 6 and Lemma 13. The read operation is linearized at the point of execution of Line 2. Clearly, this is within the duration of the call. To check the return value, let \(LP_k\) be the linearization point of the read operation and \(LP_k'\) be the linearization point previous to \(LP_k\). Then, we have \(V.val_k = V.val_{k'}\) using Lemma 11. So, it holds that \(V.val_k = L.val_k'\) using Lemma 12. Moreover, both the compare-and-swap and read functions end after executing \(O(1)\) steps and the implementation is wait-free. ▶

7 Consensus Numbers

In this section, we prove that each of the max-write and the half-max primitives has consensus number one. Note that these are two separate claims. One, that it is impossible to solve
consensus for two processes using read-write registers and registers that support the max-write and read operation. Second, that it is impossible to solve consensus for two processes using read-write registers and registers that support the half-max and read operation. Trivially, both operations can solve binary consensus for a single process (itself) by just deciding on the input value. To show that these operations cannot solve consensus for more than one process, we use an indistinguishability argument.

First, we define some terms. A configuration of the system is the value of the local variables of each process and the value of the shared registers. The initial configuration is the input 0 or 1 for each process and the initial values of the shared registers. A configuration is called a bivalent configuration if there are two possible executions starting from the configuration so that in one of them all the processes terminate and decide 0 and in the other all the processes terminate and decide 1. A configuration is called 0-valent if it is either 0-valent or 1-valent. A bivalent configuration is called critical if the next step by any process changes it to a univalent configuration. Consider an initial configuration in which there is a process $X$ with the input 0 and a process $Y$ with the input 1. This configuration is bivalent as $X$ outputs 0 if it is made to run until it terminates and $Y$ outputs 1 if it is made to run until it terminates. As the terminating configuration is univalent, a critical configuration is reached assuming that the processes solve wait-free binary consensus.

Assume that the max-write operation can solve consensus between two processes $A$ and $B$. Then, a critical configuration $C$ is reached. W.l.o.g., say that the next step $s_a$ by the process $A$ leads to a 0-valent configuration $C_0$ and that the next step $s_b$ by the process $B$ leads to a 1-valent configuration $C_1$. In a simple notation, $C_0 = Cs_a$ and $C_1 = Cs_b$. We have the following cases.

1. $s_a$ and $s_b$ are operations on different registers: The configuration $C_0s_b$ is indistinguishable from the configuration $C_1s_a$. Thus, the process $B$ decides the same value if it runs until termination from the configurations $C_0s_b$ and $C_1s_a$, a contradiction.

2. $s_a$ and $s_b$ are operations on the same register and at least one of them is a read operation: W.l.o.g., assume that $s_a$ is a read operation. Then, the configuration $C_0s_b$ is indistinguishable to $C_1$ with respect to $B$ as the read operation by $A$ only changes its local state. Thus, the process $B$ decides the same value if it runs until termination from the configurations $C_0s_b$ and $C_1$, a contradiction.

3. $s_a$ and $s_b$ are write operations on the same register: Then, the configuration $C_0s_b$ is indistinguishable from the configuration $C_1$ as $s_b$ overwrites the value written by $s_a$. Thus, the process $B$ will decide the same value if it runs until termination from the configurations $C_0s_b$ and $C_1$, a contradiction.

4. $s_a$ and $s_b$ are max-write operations on the same register $R$: Say that the arguments of these operations are $a|\bar{x}$ and $b|\bar{y}$ for $A$ and $B$ respectively. W.l.o.g., assume that $b \geq a$. Then, there are following two cases.

   a. Operation $s_b$ does not modify the register $R$. Thus, operation $s_a$ will also leave it unchanged as $b \geq a$. Also, the contents of $R$ in $C_1s_a$ is same as in $C_0$ because $s_b$ did not modify $R$ by assumption. So, the configuration $C_1s_a$ is indistinguishable from the configuration $C_0$ with respect to $A$ and it will decide same value if run until termination from the two configurations, a contradiction.

   b. Operation $s_b$ modifies the register $R$. In this case, the configurations $C_0s_b$ is indistin-
guishable from $C_1$ as $b \geq a$ and the operation $s_b$ will overwrite both the fields of the register $R$. Thus, the process $B$ will decide the same value from these configurations, a contradiction.

So, the critical configuration cannot be reached and the processes $A$ and $B$ cannot solve consensus using the max-write primitive. Thus, its consensus number is one.

For the half-max primitive, we do a similar case analysis. The first three cases are the same as in the case of max-write primitive. For the last case, assume that $s_a$ and $s_b$ are half-max operations on the same register $R$. Say that the argument of these operations are $a$ and $b$ for processes $A$ and $B$ respectively. Assume w.l.o.g. that $b \geq a$. We have the following two cases as before.

1. Say that $s_b$ does not modify $R$. In this case, even $s_a$ does not modify $R$ as $b \geq a$. Thus, the contents of $R$ is same in the configurations $C_0$ and $C_1 s_a$ and these configurations are indistinguishable to $A$. So, it will decide the same value if run until termination from these configurations, a contradiction.

2. Say that $s_b$ modifies the register $R$. In this case, the first half of the register $R$ in the configurations $C_0 s_b$ is same as the first half of $R$ in $C_1$. This is because $s_b$ overwrites the first half of $R$ in both the configurations $C_0$ and $C$. The second half is not modified by either $s_a$ or $s_b$ so the contents of $R$ is same in $C_0 s_b$ and $C_1$. Therefore, these configurations are indistinguishable with respect to $B$ and it will decide the same value if run until termination from these configurations, a contradiction.

So, the critical configuration cannot be reached and the processes $A$ and $B$ cannot solve consensus using the half-max primitive. Thus, its consensus number is one as well.

8 Conclusion

The algorithm that we presented simulates a single compare-and-swap register using $O(n)$ registers that support the half-max, max-write, read and write primitives. If $m$ compare-and-swap registers are to be simulated, then a straightforward approach requires $O(mn)$ registers. However, we can improve this if we observe that there is at most one pending operation per process even if $m$ compare-and-swap registers have to be simulated. The arrays $A$ and $R$ store the information about the latest pending call per process so there is no need to allocate them for every compare-and-swap register. Only the registers $P$ and $V$ need to be allocated separately. As the counter value $c$ used in the first half of each entry of array $A$ or $R$ is always increasing, we will be conceptually running $m$ instances of the presented algorithm using $O(m + n)$ registers. Actually, if one observes closely, the three fields used in the register $P$ are useful only when more than one compare-and-swap registers need to be implemented. Otherwise, we can use a single counter replacing both $c$ and $seq$.

One issue with the presented algorithm is that it uses unbounded sequence numbers. Thus, the algorithm only works if the size of the registers is at least logarithmic in the total number of compare-and-swap operations executed. Actually, the growth in sequence numbers can be much slower as out of the two unbounded counter types, one of them counts the total number of compare-and-swap operations executed per process and the other one counts the total number of successful compare-and-swap operations only. Also, as a first step towards understanding the power of a set of weak instructions with respect to their ability to efficiently simulate compare-and-swap, we did not focus on bounding the sequence numbers.

Using our result, one can transform any $O(T)$ time algorithm that uses compare-and-swap and read-write registers into an $O(T)$ time algorithm that uses reasonably large registers and support the instructions half-max, max-write, read and write. As the transformation
Reducing Compare-and-Swap to Consensus Number One Primitives

is wait-free, it even works for algorithms that are not wait-free. But, is it also true that any $O(T)$ time algorithm using registers that support half-max, max-write, read and write instructions can be transformed into an $O(T)$ time algorithm using compare-and-swap and read-write registers? There is an $\Omega(\log n)$ lower bound [9] on information aggregation among $n$ processes which applies to compare-and-swap and read-write registers but not to registers that support half-max, max-write, read and write instructions. Thus, it may be possible that there are tasks that take $o(\log n)$ time using max-write, half-max, read and write registers but $\Omega(\log n)$ time using compare-and-swap and read-write registers.

There are other practical factors too that can affect efficiency. For example, the half-max and the max-write operations are associative and do not return a value. Thus, they can be easier to combine in the processor memory interconnect when there is contention for the same memory location. In this paper however, we only show that a set of weak instructions can be theoretically at least as good as compare-and-swap with respect to time complexity. Although this highlights the power of a set of weak instructions, it also opens up the question that what is the best set of synchronization instructions in general.

References

1 Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies (extended abstract). In 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), Los Angeles, California, Aug 1994.
2 Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. 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, Jul 2016.
3 Faith Ellen and Philipp Woelfel. An Optimal Implementation of Fetch-and-Increment. In 27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel, Oct 2013.
4 Rati Gelashvili, Idit Keidar, Alexander Spiegelman, and Roger Wattenhofer. Brief Announcement: Towards Reduced Instruction Sets for Synchronization. In 31st International Symposium on Distributed Computing (DISC), Vienna, Austria, Oct 2017.
5 Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations. In 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Portland, Oregon, Aug 2007.
6 Maurice Herlihy. Wait-free Synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991.
7 Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.
8 Prasad Jayanti. On the robustness of Herlihy’s hierarchy. In 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), Ithaca, New York, Aug 1993.
9 Prasad Jayanti. A Time Complexity Lower Bound for Randomized Implementations of Some Shared Objects. In 17th Annual ACM Symposium on Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, Jun 1998.
10 Prasad Jayanti and Srđjan Petrovic. 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, Dec 2005.
11 Prasad Jayanti and Srđjan Petrovic. Efficient and Practical Constructions of LL/SC Variables. In 22nd Annual Symposium on Principles of Distributed Computing (PODC), Boston, Massachusetts, 2003 Jul.
12 Pankaj Khanchandani and Roger Wattenhofer. Brief Announcement: Fast Shared Counting using O(n) Compare-and-Swap Registers. In ACM Symposium on Principles of Distributed Computing (PODC), Washington, DC, USA, Jul 2017.

13 Pankaj Khanchandani and Roger Wattenhofer. On the Importance of Synchronization Primitives with Low Consensus Numbers. In 19th International Conference on Distributed Computing and Networking (ICDCN), Varanasi, India, Jan 2018.

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

15 Eric Ruppert. Determining Consensus Numbers. In 16th Annual ACM Symposium on Principles of Distributed Computing (PODC), Santa Barbara, California, Aug 1997.