Two Elementary Instructions make Compare-and-Swap

Pankaj Khanchandani
ETH Zurich
Zurich, Switzerland
kpankaj@ethz.ch

Roger Wattenhofer
ETH Zurich
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 [1] showed in his seminal work that if an object has a consensus number of n, then its instances can be used to implement any non-trivial object or data structure that is shared among n processes, so that the implementation is *wait-free* and *linearizable*. Thus, an object such as *compare-and-set* with an infinite consensus number is "advanced" because its instances can be used to implement any non-trivial concurrent object shared among any number of processes. On the other hand, objects such as *fetch-and-add* or *fetch-and-multiply* have a consensus number of two and are "elementary".**

An important consequence of Herlihy's result was that any number of reasonable elementary objects are provably insufficient to implement an advanced object like compare-and-set. However, Ellen et al. [2] observed recently that real multiprocessors do not compute using objects but using instructions that are applied on memory locations. Using this observation, they show that it is possible to use a couple of elementary instructions on the same memory location to implement an advanced one, and consequently any non-trivial object or data structure.

However, the above result is only a possibility and uses a generic universal construction as a black-box, which is not how we implement objects in practice, as the generic construction is quite inefficient with respect to the number of steps taken by a process and the number of shared objects used in the worst case. Instead, the efficient implementations are built upon the widely supported compare-and-set instruction and one cannot conclude from the previous result whether the elementary instructions can also produce equally efficient implementations like compare-and-set does or they are fundamentally limited in this respect. In this paper, we answer this question by giving a wait-free and linearizable implementation of *compare-and-set* using just two elementary instructions, *half-max* and *max-write*. The implementation takes $O(1)$ steps per process and uses $O(1)$ shared objects per process. Thus, any known or unknown *compare-and-set* based implementation can also be done using only two elementary instructions without any loss in efficiency. An interesting aspect of these elementary instructions is that depending on the underlying system, their throughput in a highly concurrent setting is larger than that of the compare-and-set instructions by a factor proportional to n.

I. Introduction

Any multiprocessor chip needs to support some synchronization instructions, such as compare-and-set 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 [1] 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*).

Using objects of consensus number n, Herlihy gave a generic and universal construction 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.

So, a compare-and-set object, which updates the current value of the object to a given new value if and only if the current value is equal to a given expected value, has an infinite consensus number and is an *advanced* object to be supported by a multiprocessor. On the other hand, consensus number two objects such as fetch-and-add or fetch-and-multiply are *elementary* ones. Moreover, as it was later shown in [3] that it is provably impossible to implement an advanced object from any number of reasonable elementary ones, a multiprocessor should support an advanced object.

However, Ellen et al. [2] recently observed that the above classification treats a synchronization instruction as an individual object but in reality the instructions supported by a multiprocessor can be applied on every memory location, without any restriction that forbids the application of more than one instruction to a memory location. Based on the observation, they also give simple examples where a register supporting two elementary instructions can be used to solve binary consensus for any given number of processes in constant number of steps per process. Using Herlihy's universal construction, it then follows that it is possible to construct any concurrent data structure or object by only using elementary synchronization instructions.

The above possibility relies on the universal construction, which is inefficient both in number of steps taken by a process and the number of shared registers used, and is therefore not used in practical implementations. One may conjecture that the elementary instructions are only good enough for solving binary consensus but are fundamentally limited to produce efficient implementations as can be obtained by widely used advanced instructions like compare-and-set. In fact, in a followup work by Gelashvili et al. [4], the authors ask a similar question: "*The practical question is whether we can really replace a compare-and-set instruction in concurrent algorithms and data-structures with a combination of weaker instructions.*".

By weaker or elementary instructions, we refer to instructions of consensus number at most two, or ideally one. We define the *consensus number of an instruction as the consensus number of an object that supports the instruction and a read operation that returns the state of the object*. This models that an instruction can be applied on a memory location along with a read operation that returns the value of the location. It is crucial to add such a read operation to the object when defining consensus number of an instruction as otherwise, one can define instructions that seem to be advanced but still have consensus number one. For instance, consider the compare-and-set instruction without a return value and an object that supports such an instruction without any read operation. The consensus number of such an object is one because there is no way to read what the object computes. Moreover, we define the read operation to return the complete state of the object as reading a memory location returns its value or all the bits stored at that location. Thus, elementary instructions with consensus number one can be seen as a "write-type" instruction because objects or memory locations that only support read and write operations also have consensus number one. The challenge is to find the limits of elementary instructions with respect to efficiently simulating compare-and-set, which is primarily used for practical implementations of concurrent objects or data structures and has been proven to yield efficient implementations.

In this paper, we show that it is possible to simulate a compare-and-set instruction using two elementary instructions and the simulation is efficient in the number of steps taken by each process and the number of shared registers or memory locations used per process. Concretely, we introduce two instructions *half-max* and *max-write* of consensus number one each. 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-set register so that every compare-and-set operation takes $O(1)$ steps. The size of the registers required is logarithmic in the length of the execution. The total number of shared registers required is $O(n)$ in a system with n processes or $O(1)$ shared registers per process. Thus, any $O(T)$ step algorithm using compare-and-set and read-write registers can be transformed into an $O(T)$ step algorithm that only uses elementary instructions of consensus number one supported on reasonably large registers. We give a couple of extensions of the above algorithm such as optimizing the number of shared objects used when several compare-and-set registers are needed instead of one. We also compare the throughput of the elementary instructions half-max and max-write against the throughput of compare-and-set in a highly concurrent setting and show that the throughput of the elementary instructions can be larger by a factor proportional to the number of processes.

II. Related Work

One of the most central question in concurrent computing has been to quantify the computing ability of synchronization instructions. Herlihy [1] 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 [5] 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 a single object.

A natural fix is to allow any number of instances of the object in the definition of consensus number, which is also the accepted definition and the one that we use [6]. Under this definition, Chandra et al. [7] show that Herlihy's hierarchy is robust for two objects out of which one is a consensus object and the other one is an arbitrary object. Ruppert [3] showed that Herlihy's hierarchy is robust for read-modify-write and readable objects, where a readable object returns some part or complete state of the object and a read-modify-write object returns the complete current state of the object and updates it according to a deterministic function in a single atomic step. These objects capture a large class of synchronization instructions but not all. 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 computational capability of set of synchronization instructions is quantified by the minimum amount of space required to solve obstruction free consensus among n processes. Obstruction freedom guarantees that a process returns the output eventually if it is allowed to take steps alone without being obstructed by other processes. A set of synchronization instructions is considered powerful if they require small space to solve obstruction free consensus. Their work has led to some more followup work to understand the capability 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-set based implementation. Their log data structure can be used to implement any object, including compare-and-set, but the progress guarantee is lock-free and does not exclude starvation of a process unlike our wait-free algorithm. The log data structure can be seen as a universal construction and the question remains if specific implementations using compare-and-set can also be done using elementary instructions without any sacrifice. Also, we do not restrict ourselves to instructions supported on modern architecture as our goal is to find if it is even theoretically possible to efficiently compete with an advanced instruction like compare-and-set using elementary instructions only, regardless of the task that we choose to do and not just a universal construction. In [8], we observed that a set of low consensus number instructions supported on the same register can help to improve the step complexity 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 capability of a set of elementary instructions supported on the same register with respect to their ability to efficiently simulate an advanced instruction like compare-and-set. We chose to simulate compare-and-set not only because of its infinite consensus number but also because it is ubiquitous and has been shown to yield efficient implementations [9], [10], [11]. Our result then implies that a set of elementary instructions can produce equally efficient implementations like compare-and-set, if not better. In [12], the authors give a blocking implementation of comparison instructions, which includes compare-and-set, 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 elementary instructions are at least as good as compare-and-swap registers with respect to the number of steps taken for completing any arbitrary synchronization task.

III. An Overview of the Method

Our method is based on the observation that if several compare-and-set 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 instructions: *max-write* and *half-max*.

The max-write instruction 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 instruction helps in keeping a version number along with a value.

The half-max instruction 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 instruction is used along with the max-write instruction to determine the single successful compare-and-set operation out of several concurrent ones. The task of determining the successful compare-and-set operation can be viewed as a variation of tree-based combining (as in [13], [14] for example). The difference is that we do not use a tree as it would incur $\Theta(\log n)$ overhead on step complexity. Instead, our method does the combining in constant number of steps as we will see later.

In the following section, we formalize the model and the problem. In Section V, we give an implementation of the compare-and-set operation using registers that support the half-max, max-write, read and write operations. In Section VI, we prove its correctness and show that the compare-and-set operation takes $O(1)$ steps for every process. In Section VII, we argue that the consensus numbers of the max-write and half-max instructions are both one. In Section VIII, we give a couple of extensions of the basic algorithm. In Section IX, we analyze the throughput of the half-max, max-write and compare-and-set instructions under high concurrency. Finally, we conclude and discuss the results in Section X.

IV. Model

A *sequential object* is defined by a set of *operations* that can be performed on the object. Each operation takes zero or more arguments, updates the *state* or *value* of the object according to a specified set of rules and optionally returns a value. The value of the object is a sequence of bits, or just an integer.

A *register* is a sequential object and supports the operations *read*, *write*, *half-max* and *max-write*. The read() operation returns the current value 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($x \mid y$) 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* executes a sequence of instructions. The processes have identifiers $1, 2, \ldots, n$. 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 the 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 of 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-set* register. It supports the *read* operation and the *compare-and-set* operation. The read operation returns the current value of the register. The compare-and-set operation takes two arguments a and b. It updates the value of the register to b if the value in the register is a and leaves it unchanged otherwise. If the value is updated, a true value is returned and false otherwise.

V. Algorithm

Figure 1 shows the (shared) registers that are used by the algorithm. There are the 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-set operations executed by the process i, the field val is used to store or *announce* the second argument of the compare-and-set 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-set operation executed by the process i. The register V stores the current *value* of the compare-and-set 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-set operation and the count of compare-and-set 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-set 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-set 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).

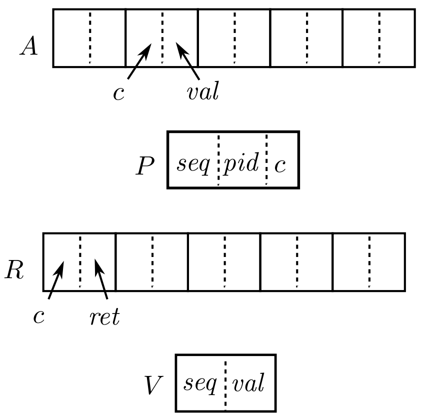

Fig. 1. An overview of data structures used by Algorithm 1.

Otherwise, the process competes with the other processes executing the compare-and-set 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).

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 an $O(1)$ step wait-free implementation of the compare-and-set object.

VI. Analysis

Let us first define some notation. We refer to a field f of a register X by $X.f$. The term $X.f_k^i$ is the value of the field $X.f$ just after the 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_k^i 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 $X.f_e$ is the value of a field $X.f$ at the end of an execution.

Algorithm 1: The compare-and-set 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 $V = (0 \,|\, x)$, where x is the initial value of the compare-and-set object.

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

To prove that our implementation is linearizable, we first need to define the linearization points. The linearization point of the compare-and-set 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 $V.seq^i_5 + 2$ to the field $V.seq$ for the first time. This is the linearization point of the process i if its compare-and-set operation was successful, as determined by the value of $P.pid$ (Case 3a). Otherwise, the failed compare-and-set operations are linearized just after the successful one (Case 3b). The calls that have not taken effect are linearized after all the other linearization points (Case 4).

Definition 1. *The linearization point of a compare-and-set call by a process i is defined as follows.*

1) *If $V.val^i_5 \neq a^i_4$, then the linearization point is the point when i executes Line 5.*
2) *If $V.val^i_5 = a^i_4 = b^i_4$, then the linearization point is the point when i executes Line 5.*
3) *If $V.val^i_5 = a^i_4 \neq b^i_4$ and $V.seq_e \geq V.seq^i_5 + 2$, then let p be the point when Line 19 is executed by a process j so that $V.seq^j_{19} = V.seq^i_5 + 2$ for the first time.*
   (a) *If $pid^j_{15} = i$, then the linearization point is p.*
   (b) *If $pid^j_{15} \neq i$, then the linearization point is just after p.*
4) *If $V.val^i_5 = a^i_4 \neq b^i_4$ and $V.seq_e < V.seq^i_5 + 2$, 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 $V.seq_e \geq V.seq^i_5 + 2$, then there is an execution of Line 19 by a process j with the value $V.seq^j_{19} = V.seq^i_5 + 2$. So, we first show in the following lemmas that this is indeed true.

Lemma 1. *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 2. *Whenever $V.seq$ changes, it increases by 2.*

Proof: If $V.seq$ was changed to x by a process i, then x must be even (Lines 19 and 17). So, a process j wrote x to $P.seq$ and either $x = V.seq^j_5 + 1$ or $x = V.seq^j_5 + 2$ (Lines 13 and 14). As $V.seq^j_5$ is even by Lemma 1 and x is even too, it must be that $x = V.seq^j_5 + 2$. As $V.seq$ always increases, we have $V.seq \geq x - 2$ before i modifies it. Also $V.seq$ is always even by Lemma 1, so we have $V.seq = x - 2$ before i modifies it and the value increases by 2. □

Lemma 3. *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 $V.seq_e \geq V.seq^i_5 + 2$, then there exists an execution of Line 19 by a process j so that $V.seq^j_{19} = V.seq^i_5 + 2$. As $V.seq^i_5$ is even by Lemma 1 and the value of $V.seq$ only increases in steps of 2 by Lemma 2, it follows from $V.seq_e \geq V.seq^i_5 + 2$ that $V.seq^i_5 + 2$ 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 the end of the corresponding function call. Second, the value returned by a finished call is same as defined by the sequence of the linearization points up to the linearization point of the call. In the following two lemmas, we show the first of these statements.

Lemma 4. *If the condition $V.val^i_5 = a^i_4 \neq b^i_4$ is true for a compare-and-set call by a process i, then the value of $V.seq$ is at least $V.seq^i_5 + 2$ at the end of the call.*

Proof: We define a set of processes $S = \{j : V.seq^j_5 = V.seq^i_5\}$. 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 $V.seq^i_5 + 2$ to that field, we have $seq^k_{17} = seq^k_{15} \geq V.seq^i_5 + 2$. If $seq^k_{15} > V.seq^i_5 + 2$, then $V.seq^k_{15} \geq V.seq^i_5 + 2$ and we are done.

So, we only need to check the case when $seq^k_{15} = V.seq^i_5 + 2$. As $V.seq^i_5$ is even by Lemma 1, so is $seq^k_{17} = seq^k_{15}$. Moreover, the process $pid^k_{15} \in S$ as some process(es) (including k)

executed Line 13. As $A[pid^k_{15}].c$ always increases whenever modified (Line 11), we have $ca^k_{16} \geq cp^k_{15}$. But, if $ca^k_{16} > cp^k_{15}$, then the process pid^k_{15} finished even before the process k, a contradiction. So, it holds that $ca^k_{16} = cp^k_{15}$ 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 $V.seq^k_{19} = V.seq^i_5 + 2$ and we are done. Otherwise, someone already changed the value of $V.seq$ to at least $V.seq^i_5 + 2$ because of Lemma 2. □

Lemma 5. *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 $V.seq_e \geq V.seq^i_5 + 2$ and $V.seq^i_5$ 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 4 that the value of $V.seq$ is at least $V.seq^i_5 + 2$ when the call ends. So, the point when Line 19 writes $V.seq^i_5 + 2$ to $V.seq$ is within the call duration.

We know from Lemma 4 that if the call finishes, then we have $V.seq_e \geq V.seq^i_5 + 2$. So, if $V.seq_e < V.seq^i_5 + 2$, 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 6. *Assume that $x = seq^i_{15} = seq^j_{15}$ for two distinct processes i and j and that x is even. Then, it implies that $pid^i_{15} = pid^j_{15}$ and $cp^i_{15} = cp^j_{15}$.*

Proof: Without loss of generality, assume that the process i executes Line 15 before the process j does so. As $x = seq^i_{15} = seq^j_{15}$ by assumption, the only way in which the field $P.pid$ can change until the process j executes Line 15, is by a max-write operation on P with x as the first field. This is not possible as x is even and the max-write on P is only executed with an odd value of the first field (Line 13). So, it holds that $pid^i_{15} = pid^j_{15}$. Similarly, we have $cp^i_{15} = cp^j_{15}$. □

Lemma 7. *As long as the value of $V.seq$ remains the same, the value of $V.val$ does not change.*

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^i_{16}. To have a different value of $V.val$ with x as the value of $V.seq$, another process j must execute Line 19 with $seq^j_{15} = x$ but $val^i_{16} \neq val^j_{16}$. As $seq^j_{15} = x = seq^i_{15}$, it follows from Lemma 6 that $pid^j_{15} = pid^i_{15}$ and $cp^i_{15} = cp^j_{15}$. As the condition in Line 17 is true for both the processes i and j, it then follows that $ca^i_{16} = ca^j_{16}$. As the field $A[pid^j_{15}].val$ is updated only once for a given value of $A[pid^j_{15}].c$ (Line 11), it holds that $val^i_{16} = val^j_{16}$ and the claim follows. □

Lemma 8. *Say that $seq^i_{15} = x$ is even and $pid^i_{15} = j$ during a call by a process i, then $V.seq^j_5 = x - 2$ for some call by the process j.*

Proof: As $seq^i_{15} = x$, some process h modified P by executing Line 13 or Line 14 with x as the first argument. As x is even and $V.seq^h_5$ is even by Lemma 1, the process h modified P by executing Line 14. So, it holds that $V.seq^h_5 = x - 2$. Also, process h executed Line 13 with $x - 1$ as the first field. As $pid^i_{15} = j$, the process j also executed Line 13 with $x - 1$ as the first field after the process h did so. So, it holds that $V.seq^j_5 = x - 2$. □

Lemma 9. *For every even value $x \in [2, V.seq_e]$, there is an execution of Line 19 by a process i so that $seq^i_{15} = x$ and the first such execution is the linearization point of some call.*

Proof: Consider an even value $x \in [2, V.seq_e]$. Then, we know from Lemma 2 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 $seq^j_{15} = x$. So, it holds for the process $pid^j_{15} = h$ that $V.seq^h_5 = x - 2$ using Lemma 8. As point p is the first time when x is written to the field $V.seq$, it holds that $V.seq^j_{19} = x$. Thus, p is the linearization point of the process h by Definition 1. □

Lemma 10. *The value $V.val$ is only modified at a Case 3a linearization point.*

Proof: Let q be a Case 3a linearization point. Say that the value of $V.seq$ is updated to $x \geq 2$ at q. Let p be the first point in the execution when the value of $V.seq$ is $x - 2$. Using Lemma 9, we conclude that p is either a linearization point (for $x - 2 \geq 2$) or the initialization point (for $x - 2 = 0$). Using Lemma 7, 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-set 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 the initialization when $k = 0$. For $k \geq 1$, the term $L.ret_k$ 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 11. *After $k \geq 0$ linearization points, we have $L.val_k = V.val_k$ except for Case 4 linearization points. For $k \geq 1$, the $L.ret_k$ 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 value of the compare-and-set 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 $L.val_{k'} = V.val_{k'}$. By Lemma 10, the value of $V.val$ does not change until LP_k. As we have a read operation at LP_k, it holds that $V.val_{k'} = V.val_k$. By Definition 1, we know that $V.val_k \neq a_4^i$. So, it holds that $L.val_{k'} = V.val_{k'} = V.val_k \neq a_4^i$. Thus, it follows from the specification of the compare-and-set object that $L.val_k = L.val_{k'} = V.val_k$. Moreover, we have $L.ret_k = \mathit{false}$ as $L.val_{k'} = V.val_k \neq a_4^i$.

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 $V.val_k = a_4^i = b_4^i$. So, it holds that $L.val_{k'} = V.val_{k'} = V.val_k = a_4^i$. Thus, it follows from the object's specification that $L.val_k = b_4^i = V.val_k$. Further, we have $L.ret_k = \mathit{true}$ as $L.val_{k'} = a_4^i$.

Case 3a: Consider the point $LP_{k'}$ when the value $V.seq_k - 2$ was written to $V.seq$ for the first time. As $V.seq_k$ is even by Lemma 1, it follows from Lemma 9 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 $V.seq_5^i = V.seq_{k'}$. Thus, it holds that $V.val_5^i = V.val_{k'}$ by Lemma 7. Therefore, $V.val_5^i = L.val_{k'}$ as $L.val_{k'} = V.val_{k'}$ by induction hypothesis. Using definition of Case 3a, it also holds that $a_4^i = V.val_5^i$. Thus, we have $a_4^i = L.val_{k'}$ and $L.val_k = b_4^i$.

Now, assume that the instruction at LP_k was executed by a process j. Using definition of Case 3a, we have $i = pid_{15}^j$. As LP_k is the first time when the value of $V.seq$ is $V.seq_k = V.seq_5^i + 2$, we conclude that the process i is not finished until LP_k by using Lemma 4. As $seq_{15}^j = V.seq_k = V.seq_5^i + 2$, it is true that some process i' has $V.seq_5^{i'} = V.seq_5^i$ and that the process executed Line 13 until LP_k. As $i = pid_{15}^j$, 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 4 that there is a unique call for any process h given a fixed value of $V.seq_5^h$. Thus, the process i already executed Line 11 with b_4^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 $val_{16}^j = b_4^i$ and that $V.val_k = b_4^i$ as well. Because $a_4^i = L.val_{k'}$ as shown before, we also have $L.ret_k = \mathit{true}$.

Case 3b: Let $LP_{k'}$ be the first point when the value $V.seq_k$is written to $V.seq$ ($LP_{k'}$ 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 $V.seq_5^i = V.seq_{k'} - 2$. As process j wrote $V.seq_{k'}$ to $V.seq$, we have $V.seq_5^j = V.seq_{k'} - 2$ as well. So, we have $V.val_5^i = V.val_5^j$ using Lemma 7. Using definition of Case 3a and Case 3b, respectively, we have $a_4^j = V.val_5^j \neq b_4^j$ and $a_4^i = V.val_5^i$. So, we have $a_4^i \neq b_4^j$. We have $b_4^j = L.val_{k'}$ as argued in the previous case, so $L.val_k = L.val_{k'}$. 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 $L.ret_k = \mathit{false}$ as $a_4^i \neq b_4^j = L.val_{k'}$. □

Lemma 12. *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 $L.ret_k$.*

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 11, we have $L.ret_k = \mathit{false}$ 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 11, we have $L.ret_k = \mathit{true}$ 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 $V.seq_k - 1$. So, the process i also initialized $R[i]$ to $(cp_{15}^j \,|\, \mathit{false})$ in Line 12. Moreover, the process j wrote the value $(cp_{15}^j | \mathit{true})$ to $R[i]$ afterwards using a max-write operation. Thus, the value of $R[i].ret$ 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 $L.ret_k$ given by Lemma 11.

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 $(x \,|\, \mathit{false})$ during the call (Line 12). Consider a process j that tries to write *true* to $R[i].ret$ 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 $V.seq_k$. First, consider that $seq_{15}^j > V.seq_k$. As $pid_{15}^j = i$ and seq_{15}^j is even, we have $V.seq_5^i = seq_{15}^j - 2$ using Lemma 8. So, we have $V.seq_5^i > V.seq_k - 2$. This cannot happen until i finishes as $V.seq_5^i = V.seq_k - 2$ for the current call by i using definition of Case 3b. Second, consider that $seq_{15}^j = V.seq_k$. Using definition of Case 3b, there is a process h so that $pid_{15}^h \neq i$ and $seq_{15}^h = V.seq_k$. As $seq_{15}^j = V.seq_k$ by assumption, we have $pid_{15}^j \neq i$ using Lemma 6. This contradicts our assumption that $pid_{15}^j = i$. Third, consider that $seq_{15}^j < V.seq_k$. As $pid_{15}^j = i$ and seq_{15}^j is even, we have $V.seq_5^i = seq_{15}^j - 2$ using Lemma 8. So, we have $V.seq_5^i < V.seq_k - 2$. This corresponds to a previous call by the process i as $V.seq_5^i = V.seq_k - 2$ for the current call by i. So, it holds that $ca_{16}^j < x$ and execution of Line 18 has no effect. Thus, the process i returns *false* for Case 3b which matches the $L.ret_k$ value given by Lemma 11.

If the k^{th} linearization point is a Case 4 point, then we know from Lemma 4 that the call is unfinished and we need not consider it. □

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

Theorem 1. *Algorithm 1 is a wait-free and linearizable implementation of the compare-and-set register where both the compare-and-set and read functions take $O(1)$ steps.*

Proof: We conclude that the compare-and-set function as given by Algorithm 1 is linearizable by using Lemma 5 and

Lemma 12. 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 10. So, it holds that $V.val_k = L.val_{k'}$ using Lemma 11. Moreover, both the compare-and-set and read functions end after executing $O(1)$ steps and the implementation is wait-free. □

VII. Consensus Numbers

In this section, we prove that each of the max-write and the half-max instructions 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 in all the possible executions starting from the configuration, the processes terminate and decide 0. Similarly, a configuration is called 1-*valent* if in all the possible executions starting from the configuration, the processes terminate and decide 1. A configuration is called a univalent configuration 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. Without loss of generality, 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: Without loss of generality, 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 $a \mid x$ and $b \mid y$ for A and B respectively. Without loss of generality, 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 indistinguishable 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 instruction. Thus, its consensus number is one.

For the half-max instruction, we do a similar case analysis. The first three cases are the same as in the case of max-write instruction. 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 without loss of generality that $b \geq a$. Say that $(v|w)$ is the value of register R in C. Then, the value of R in both the configurations C_0s_b and C_1 is $(\max\{v, b\}|w)$ whether or not B modifies R in s_b. So, B will decide the same value if run until termination from these configurations, a contradiction. Thus, the critical configuration cannot be reached and the processes A and B cannot solve consensus using the half-max instruction. Thus, its consensus number is one as well.

VIII. Extensions

In this section, we discuss couple of extensions of the previous algorithm. First, we note that the compare-and-set instruction returns either true or false depending on whether the value is updated or not. Instead, one may want to return the value of the register prior to the application of the instruction.

This is also known as the *compare-and-swap* operation. The presented algorithm can be easily modified to accommodate this requirement. The algorithm just needs to return the value val read from the register V at the start of the operation. Thus, we have the following corollary.

Corollary 1. *There is a wait-free and linearizable implementation of the compare-and-swap register using elementary instructions so that both the compare-and-swap and read functions take $O(1)$ steps.*

Second, the algorithm that we presented simulates a single compare-and-set register using $O(n)$ registers that support the half-max, max-write, read and write instructions. If m compare-and-set 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-set 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-set 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 multiple compare-and-set registers are implemented. Otherwise, we can use a single counter replacing both c and seq. So, we also have the following corollary.

Corollary 2. *There is a wait-free and linearizable implementation of m compare-and-set registers using $O(n+m)$ registers supporting elementary instructions so that both the compare-and-set and read functions take $O(1)$ steps.*

IX. Throughput

In this section, we analyze the throughput of compare-and-set versus half-max and max-write in a highly concurrent setting. We assume that there are n processes in the system that communicate to the memory via an interconnect. We compare the throughput of the following two scenarios.

1) Each process executes a sequence of m compare-and-set operations to the same memory location.
2) Each process executes a sequence of operations so that a read operation alternates with either max-write or half-max to the same memory location. The number of non-read operations is m for each process.

In the second scenario, we assume that a process executes a read operation after each of the elementary operations as they do not return a value in our case. We assume that a batch of instructions — at most one instruction from each process — can enter the interconnect in each step for updating the memory. The interconnect takes t_b steps to process a batch of instructions. We call an instruction as *compact* if the corresponding function satisfies the following: given any sequence S of function calls, there is a smaller sequence S' of function calls from S and having the same effect as the sequence S. Thus, we have the following result.

Lemma 13. *Half-max and max-write are compact instructions whereas compare-and-set is not.*

Proof: Consider a sequence S of half-max operations. We consider a smaller sequence S' so that an operation with the largest argument, say half-max(l), is in S'. Let the current value of the register be $(a|b)$. Thus, applying the sequence S or S' results in the same value $(\max\{a,l\},b)$ of the register.

Consider a sequence S of max-write operations. Let max-write$(l|w)$ be the last operation in S with the largest value of the first argument. We construct a smaller sequence S' where max-write$(l|w)$ is the last operation with the value l as the first argument. Let the current value of the register be $(a|b)$. If $l \geq a$, then applying the sequence S or S' results in the same final value $(l|w)$ as l is the largest value of the first argument and max-write$(l|w)$ is the last operation in S or S' with l as the first argument. If $l < a$, then neither the operations in S or S' change the value of the register as the first argument of all the operations is strictly smaller than a.

Consider a set S of compare-and-set operations so that all the $2|S|$ arguments of the operations are unique values. Assume for contradiction that compare-and-set is compact. Then, there is a smaller sequence S' that has the same effect as the sequence S. So, there is a compare-and-set operation that is in the sequence S but not in the sequence S'. Let (a_f, b_f) be the argument of that operation. If the current value of the register is a_f, then applying the sequence S results in the final value b_f, since all the argument values are unique and consequently, all the operations before and after the one with the argument (a_f, b_f) are unsuccessful. However, applying the sequence S' does not change the value of the register as none of those operations have first argument as a_f. So, the final value remains $a_f \neq b_f$, a contradiction. □

Say that the underlying interconnect is such that the time t_b to process a batch of instructions is composed of stages, where the batch size decreases in subsequent stages and the stages are *pipelined*. We define the *concurrent throughput* of compare-and-set as the average number of instructions that are processed and applied to the memory in a single step of the first scenario. Similarly, the *concurrent throughput* of half-max and max-write is the average number of instructions that are processed and applied to the memory in a single step of the second scenario. The following theorem quantifies the concurrent throughput of the instructions.

Theorem 2. *The concurrent throughput of compare-and-set is $\frac{n}{t_b}$ and that of half-max and max-write for a pipelined interconnect is $\frac{n}{6+t_b/m}$.*

Proof: For compare-and-set, each process executes a sequence of m operations so a total of mn operations are executed. However, the next batch of instructions cannot enter the interconnect unless the current batch of instructions is processed and the resulting value is returned since compare-and-set is not a compact instruction by Lemma 13. Thus, each batch takes t_b steps and mt_b steps are taken in total. So, the throughput is $\frac{mn}{mt_b} = \frac{n}{t_b}$.

Consider the case of half-max and max-write operations. Each process executes $2m$ operations including the read operations. We divide each batch of instructions to be processed into three batches so that each batch contains only a single type of instruction, i.e., read, half-max or max-write. As these are compact instructions by Lemma 13, the next batch of instructions can enter the pipeline without waiting for the current batch of instructions to finish processing. Thus, the total number of steps to process all the mn non-read operations is $3 \cdot 2m + t_b$. So. the throughput is $\frac{mn}{6m+t_b} = \frac{n}{6+t_b/m}$. $\square$

If $m >> n$, then the throughput values are approximately $\frac{n}{6}$ for the elementary instructions versus $\frac{n}{t_b}$ for compare-and-set, which is lower by a factor of $\frac{t_b}{6}$. The quantity t_b is usually proportional to the number of processes for current systems [15].

X. Conclusion

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-set 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-set operations executed per process and the other one counts the total number of successful compare-and-set operations only. We still think that the result helps us to understand the capability of a set of elementary instructions with respect to their ability to efficiently simulate compare-and-set.

Using our result, one can transform any $O(T)$ step algorithm that uses compare-and-set and read-write registers into an $O(T)$ step algorithm that uses reasonably large registers and support the instructions half-max, max-write, read and write. As the transformation is wait-free, it even works for algorithms that are not wait-free. But, is it also true that any $O(T)$ step algorithm using registers that support half-max, max-write, read and write instructions can be transformed into an $O(T)$ step algorithm using compare-and-set and read-write registers? There is an $\Omega(\log n)$ lower bound [16] on information aggregation among n processes which applies to compare-and-set 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 $o(\log n)$ steps using max-write, half-max, read and write registers but $\Omega(\log n)$ steps using compare-and-set and read-write registers. Simple max registers were introduced in [17], [18] and were used to build a wait-free implementation of any monotone circuit, which is a generalization of a non-decreasing function on a single input to a function that transforms multiple inputs into a single output in stages and in a non-decreasing fashion.

There are other practical factors too other than step complexity and number of shared registers used. For example, the half-max and the max-write operations do not return a value. Thus, they can spend lesser time in traversing the interconnect and have lesser latency. In this paper however, we show that a set of two elementary instructions can be at least as good as compare-and-set with respect to step complexity, the number of shared registers used and also the throughput in a highly concurrent setting. This highlights the capability of a set of elementary instructions and shows another aspect in choosing the best synchronization instructions to support in general.

References

[1] M. Herlihy, "Wait-free Synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[2] F. Ellen, R. Gelashvili, N. Shavit, and L. 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] E. Ruppert, "Determining Consensus Numbers," in *16th Annual ACM Symposium on Principles of Distributed Computing (PODC), Santa Barbara, California*, Aug 1997.

[4] R. Gelashvili, I. Keidar, A. Spiegelman, and R. Wattenhofer, "Brief Announcement: Towards Reduced Instruction Sets for Synchronization," in *31st International Symposium on Distributed Computing (DISC), Vienna, Austria*, Oct 2017.

[5] P. Jayanti, "On the robustness of Herlihy's hierarchy," in *12th Annual ACM Symposium on Principles of Distributed Computing (PODC), Ithaca, New York*, Aug 1993.

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[7] T. Chandra, V. Hadzilacos, P. Jayanti, and S. 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.

[8] P. Khanchandani and R. 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.

[9] P. Jayanti and S. Petrovic, "Efficient and Practical Constructions of LL/SC Variables," in *22nd Annual Symposium on Principles of Distributed Computing (PODC), Boston, Massachusetts*, 2003 Jul.

[10] M. 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.

[11] P. Jayanti and S. 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.

[12] W. Golab, V. Hadzilacos, D. Hendler, and P. 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.

[13] F. Ellen and P. Woelfel, "An Optimal Implementation of Fetch-and-Increment," in *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel*, Oct 2013.

[14] P. Khanchandani and R. 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.

[15] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *24th ACM Symposium on Operating System Principles (SOSP), Farminton, Pennsylvania*, Nov 2013.

[16] P. 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.

[17] J. Aspnes, H. Attiya, and K. Censor, "Max Registers, Counters, and Monotone Circuits," in *Proceedings of the 28th ACM symposium on Principles of Distributed Computing (PODC), Calgary, AB, Canada*, Aug 2009.

[18] J. Aspnes, H. Attiya, and K. Censor-Hillel, "Polylogarithmic Concurrent Data Structures from Monotone Circuits," *Journal of the ACM*, 2012.