The Synchronization Power of Atomic Bitwise Operations

Damien Imbs

Department of Mathematics, University of Bremen, Bremen, Germany
imbs@math.uni-bremen.de

Abstract

In a distributed system, processes must reach a certain level of synchronization to solve a common problem. The strongest form of synchronization can be reached through consensus: all the processes must agree on a common value that has been proposed by one of them. Consensus is universal in shared memory systems: any type of shared object can be implemented using it. Unfortunately, consensus is impossible to solve using only shared registers when processes can crash.

To circumvent this impossibility, one can use stronger objects, for example Test&Set or Compare&Swap. The synchronization power of these objects can be measured using the concept of Consensus Number: the maximum number of processes for which they can solve consensus in a crash-prone system.

Bitwise AND, OR and XOR operations are very widely used, but have received little attention in the distributed setting. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems. It is then natural to consider the level of synchronization that these operations can achieve.

This paper introduces shared AND/OR and AND/OR/XOR registers. A shared AND/OR register consists of an array of x bits and offers three atomic operations: AND and OR operations, which take an array of x bits as parameter and change the state of the register by applying the corresponding bitwise operation, and a read operation which returns the content of the array. A shared AND/OR/XOR register additionally offers a XOR operation.

We show that shared AND/OR registers of x bits have consensus number $\lfloor \frac{x+1}{2} \rfloor$, by presenting an algorithm that solves consensus using these registers, and by proving that consensus cannot be solved for n processes using AND/OR registers that have strictly less than $2n-1$ bits. We then show that shared AND/OR/XOR registers of x bits have consensus number x using a similar technique.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Asynchronous systems, Binary operations, Consensus, Consensus number, Read/write shared memory, Shared objects, Synchronization, Wait-freedom

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.26

1 Introduction

A fundamental issue in distributed systems is the synchronization of various processes. The highest level of synchronization can be attained when processes can reach *consensus,* that is, when they can all agree on a value proposed by one of them. Consensus is *universal* in shared memory: using consensus, one can build any shared object that has a sequential specification [8, 9, 14].

Unfortunately, in some communication models, consensus is impossible when even a single process can crash. This impossibility was first proven for message-passing systems [5], and was then extended to shared memory systems [13].

In order to implement consensus, one must then use objects stronger than shared registers, for example Compare&Swap. The synchronization power of such objects can be measured using their *consensus number:* the maximum number of processes for which they can solve consensus in a *wait-free* manner, that is, when any number of processes can crash [9]. Compare&Swap, for example, has consensus number $+\infty$: it can solve consensus for any number of processes. Test&Set, on the other hand, has a much lower synchronization power: its consensus number is only 2, meaning that it can solve consensus for 2 processes, but not for 3 processes.

The concept of consensus number was introduced in [9] but it was not clear whether this hierarchy was *robust*, that is, whether various objects of consensus number x could combine to give an object with a higher consensus number. In [12], an object is presented that, when only a single instance is used, can solve consensus for two processes but not for three processes. However, when x such objects are used, consensus can be solved for $x+1$ processes. The author of [12] refined the definition of the consensus number of an object to make this hierarchy robust: an object type X has consensus number x if, using any number of objects of type X and of shared registers, the maximum number of processes for which consensus can be solved is x. According to this definition, when combining various objects of different types that have consensus number at most x, one cannot obtain an object that has consensus number $y > x$.

Content of the paper

Bitwise operations are very widely used in the sequential setting but have received surprisingly little attention in the context of distributed computing. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems. It is then natural to consider the level of synchronization that these operations can achieve.

This paper introduces the concepts of *shared AND/OR* and *AND/OR/XOR* registers. An x-bits shared AND/OR register consists of an array of x bits and offers three operations: $and()$, $or()$, and $read()$. The $and()$ and $or()$ operations take as a parameter an array of x bits, and apply the corresponding bitwise operation to the object. The $read()$ operation returns the content of the whole array. An shared AND/OR/XOR register offers an additional $xor()$ operation that applies the XOR bitwise operation.

- The paper first presents an algorithm that solves wait-free consensus for n processes using $(2n-1)$-bits AND/OR registers. The algorithm consists of a series of competitions between one process that tries to impose its input as the value chosen for the consensus, and the other processes that try to prevent it from doing so. The algorithm is formally proved correct.
- A modification of the previous algorithm that solves wait-free consensus for n processes using n-bits AND/OR/XOR registers is then presented.
- Two impossibility results are given. It is shown that consensus cannot be solved for n processes using read/write registers and AND/OR registers that have strictly less than $(2n-1)$ bits. This impossibility implies that the first algorithm is optimal with respect to the number of processes that can solve consensus using AND/OR registers, and thus shows that the consensus number of x-bits AND/OR registers is $\lfloor \frac{x+1}{2} \rfloor$. It is then shown that consensus cannot be solved for n processes using read/write registers and

AND/OR/XOR registers that have strictly less than n bits, thus showing that the second algorithm is also optimal and that the consensus number of n-bits AND/OR registers is x.

Related work

In [9], after introducing the concept of consensus number, its value is studied for various objects. Among other results, it is shown that Test&Set, Fetch&Add, Swap, the stack and the FIFO queue all have consensus number 2, Compare&Swap, memory-to-memory move, memory-to-memory-swap all have consensus number $+\infty$ and that m-register assignment (writing to m registers atomically) has consensus number $2m-2$. The bitwise operations considered here differ from m-register assignment in that they can only modify bits of a single bounded register, whereas m-register assignment can modify any arbitrary set of m registers. Additionally, in the case of the XOR bitwise operation, the new values of the bits modified by the operation depend on their previous values; m-register assignment simply overwrites previous values.

Among the objects that have consensus number 2, the objects of the *Common2* class [2] have an additional property: they can all be implemented using consensus objects that can only be accessed by two processes. In [1] it is shown that the stack belongs to Common2. Whether the FIFO queue belongs to Common2 is still an open problem.

Test&Set belongs to the Common2 class. In [11], its specification is slightly modified to obtain a new object that has consensus number $+\infty$. The idea of the modification is to share the value returned by the operation.

In [7], a model of multicore architectures such as Compute Unified Device Architecture (CUDA) is presented, and its consensus number is studied.

The consensus problem has been generalized to the *set agreement* problem [4]. In the k-set agreement problem, processes must decide at most k different values that have been proposed by some process. The k-set agreement problem has been shown to be impossible to solve in shared memory when any number of processes can crash, even when $k = n-1$ [3, 10, 15]. In [6], tasks (one-shot objects) are classified according to their ability to solve k-set agreement, for k from 1 (consensus) to $n-1$.

Roadmap

The paper is composed of 6 sections. Section 2 presents the model, introduces the concept of AND/OR and AND/OR/XOR registers, and presents the consensus number hierarchy. Section 3 presents an algorithm that solves consensus using AND/OR registers. Section 4 presents a modification of the previous algorithm that solves consensus using AND/OR/XOR registers. Section 5 shows that these algorithms are optimal with respect to the number of processes that can solve consensus, and determines the consensus number of AND/OR and AND/OR/XOR registers. Finally, Section 6 concludes the paper.

2 Model and definitions

We consider a set Π of n processes $p_1, \ldots, p_n$. Processes are *asynchronous;* there is no assumption on their respective speeds. Moreover, any process can crash: it can stop its execution at any point in time.

In a given execution, a process that crashes is said to be *faulty*. Otherwise, it is *correct* and executes an infinite number of steps.

2.1 Communication model: shared memory, AND/OR registers and AND/OR/XOR registers

Processes communicate by reading and writing atomic registers. In addition, they can also access *shared AND/OR registers* or *shared AND/OR/XOR registers.*

An AND/OR register REG consists of an array $STATE$ of x bits and offers three operations: $and()$, $or()$, and $read()$. All these operations are atomic: they appear as being executed at a single time instant.

- *The $REG.and(array)$ operation.* The $and()$ operation takes as a parameter an array $array$ of x bits. It does not return a value.
  For each entry $y \in [1..x]$ of its array $STATE$, it executes the binary *and* operation with the corresponding entry $array[y]$ of the operation parameter. It then stores the result back in the y^{th} entry of its own array. More formally, it executes the following:

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \wedge array[y]$$

  Due to the nature of the binary AND operation, it can be reformulated as follows:

  $$\forall y \in [1..x] : \textbf{ if } array[y] = 0 \textbf{ then } STATE[y] \leftarrow 0 \textbf{ end if}$$

- *The $REG.or(array)$ operation.* The $or()$ operation is similar to the $and()$ operation, the difference being that instead of applying the binary *and* operation, it applies the binary *or* operation. It executes the following.

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \vee array[y]$$

  Again due to the nature of the binary OR operation, it can be reformulated as follows:

  $$\forall y \in [1..x] : \textbf{ if } array[y] = 1 \textbf{ then } STATE[y] \leftarrow 1 \textbf{ end if}$$

- *The $REG.read()$ operation.* The $read()$ operation atomically returns the content of the x-bits array of the register.

An AND/OR/XOR register REG offers an additional operation $xor()$.

- *The $REG.xor(array)$ operation.* The atomic $xor()$ operation is similar to the $and()$ and $or()$ operations. It applies the binary *xor* operation. It executes the following.

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \oplus array[y]$$

  Differently from the $and()$ and $or()$ operations, it cannot be reformulated as a set of writes: for any entry of $array$ and $STATE$, the result depends on both values.

2.2 The consensus number hierarchy

The consensus number of an object is a measure of its power of synchronization in failure-prone shared memory systems [9]. It is based on the *consensus* problem.

Consensus is a one-shot problem: it offers a single operation propose() that each process can invoke only once. The propose(v) operation takes a value v as input and returns an output: a process *decides* a value if it chooses this value as its output.

The consensus problem is defined by the following properties.

- *Validity.* The decided value is a proposed value.
- *Agreement.* No two different processes decide different values.
- *Wait-free termination.* Every correct process decides.

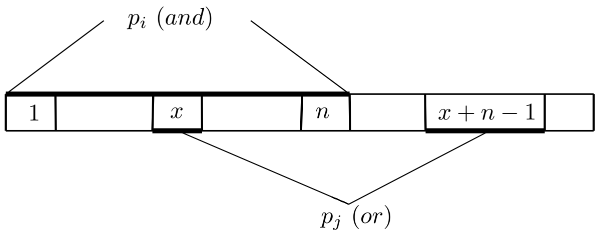

Figure 1 $AND\_OR[i]$ $(2n-1$ bits): bits modified by the owner p_i and another process p_j.

An object type T is then said to have consensus number c if, in an asynchronous shared memory system in which any number of processes can crash, consensus can be solved for c processes using atomic read/write registers and any number of objects of type T, but not for $c+1$ processes. If consensus can be solved for any number of processes, T is said to have consensus number ∞.

3 Solving consensus using AND/OR registers

In this section, we present an algorithm that solves consensus for n processes using $(2n-1)$-bits AND/OR registers. The algorithm, presented in Figure 2, constitutes a proof that x-bits AND/OR registers have consensus number at least $\lfloor \frac{x+1}{2} \rfloor$.

3.1 Mechanism of the algorithm

Every process, except p_n, competes against all the other processes. There are then $n-1$ "competitions", each associated to a single process, the "owner" of the competition. If a process wins in its competition, its value may be decided. If it loses, its value cannot be decided. Among the processes that win their competition, the one with the greatest id wins the consensus, that is, all the correct processes decide its input value. If there is no such process (all the processes $p_1, \ldots, p_{n-1}$ lost their competition), then the value of p_n is decided.

Process p_i, for $i < n$, participates in its own competition before participating in the competitions of all the other processes. Process p_n participates directly in all the competitions. If a process participates alone in its own competition, it wins. If a process participates alone in the competition of another process, the owner loses. This guarantees that, if p_n does not participate, at least one process wins its own competition and its value can be decided. If no process wins its own competition, then p_n has participated and its value can be decided.

Each competition uses an AND/OR register $AND\_OR[i]$ of $2n-1$ bits. The register is initialized with $AND\_OR[i][1..n] = [1, \ldots, 1]$ and $AND\_OR[i][n+1..2n-1] = [0, \ldots, 0]$. The owner p_i uses an *and* operation, while the other processes use an *or* operation. Process p_i overwrites (by having the corresponding bits of the parameter of its $and()$ operation set to 0) the bits $AND\_OR[i][1..n]$. Process p_j overwrites (by having the corresponding bits of the parameter of its $or()$ operation set to 1) the bits $AND\_OR[i][x]$ and $AND\_OR[i][x+n-1]$, where $x = j+1$ if $j < i$, and $x = j$ otherwise (the difference between $j < i$ and $j > i$ comes from the fact that p_i does not have dedicated $AND\_OR[i][x]$ and $AND\_OR[i][x+n-1]$ bits). The modifications of $AND\_OR[i][1]$ by p_i and of $AND\_OR[i][x+n-1]$ by p_j allow determining if the corresponding process issued its operation. The modification of $AND\_OR[i][x]$ by both p_i and p_j allows determining which process issued its operation first. Figure 1 presents the layout of the bits of an AND/OR register modified during a competition.

```
Initially:
  $\forall x \in [1..n-1]$ :
      $\forall y \in [1..n] : AND\_OR[x][y] = 1$;
      $\forall z \in [n+1..2n-1] : AND\_OR[x][z] = 0$.

Operation propose$_i(v)$:    % Code for $p_i$ %
(01)  $IN[i] \leftarrow v$;
(02)  if $(i < n)$ then                   % $and\_array$: array of $2n-1$ bits %
(03)     $and\_array \leftarrow [0, \ldots, 0]$;
(04)     for $x$ from $n+1$ to $2n-1$ do
(05)        $and\_array[x] \leftarrow 1$
(06)     end for;
(07)     $AND\_OR[i].and(and\_array)$
(08)  end if;
(09)  for $j$ from $1$ to $n-1$ do
(10)     if $(j \neq i)$ then              % $or\_array$: array of $2n-1$ bits %
(11)        $or\_array \leftarrow [0, \ldots, 0]$;
(12)        if $(i < j)$ then
(13)           $or\_array[i+1] \leftarrow 1$;
(14)           $or\_array[n+i] \leftarrow 1$
(15)        else   % $i > j$ %
(16)           $or\_array[i] \leftarrow 1$;
(17)           $or\_array[n+i-1] \leftarrow 1$
(18)        end if;
(19)        $AND\_OR[j].or(or\_array)$
(20)     end if
(21)  end for;
(22)  $output \leftarrow \bot$;
(23)  for $j$ from $1$ to $n-1$ do
(24)     $current \leftarrow AND\_OR[j].read()$;
(25)     if $(current[1] = 0)$ then
(26)        if $\big(\nexists x \in [2..n] : (current[x] = 0) \wedge (current[n+x-1] = 1)\big)$
(27)           then $output \leftarrow IN[j]$
(28)        end if
(29)     end if
(30)  end for;
(31)  if $(output = \bot)$ then  $output \leftarrow IN[n]$ end if;
(32)  return$(output)$.
```

Figure 2 An algorithm that solves consensus for n processes using $(2n-1)$-bits AND/OR registers (code for p_i).

3.2 Shared objects

The algorithm uses the following shared objects.

- *An array* $IN[1..n]$ *of read/write registers.* The array IN contains one entry per process. When a process starts its execution, it writes its input in the corresponding entry of the array IN (line 01). The array is used to determine the input value of the process whose value is decided (lines 27 and 31).
- *An array* $AND\_OR[1..n-1]$ *of* $(2n-1)$*-bits shared AND/OR registers.* Each process, except p_n, has an associated AND/OR register. These registers are used as arbiters. Process p_i (for $i \neq n$) uses $AND\_OR[i]$ to compete against all other processes. In this competition, p_i uses an *and* operation (line 07), while the other processes use an *or* operation (line 19). If p_i is the first to invoke an operation on $AND\_OR[i]$, it wins and its value *may* be chosen. Otherwise, another process has invoked an operation on $AND\_OR[i]$ before p_i and the value of p_i will not be chosen. After competing in all AND/OR registers, process p_i reads them all to determine which value to decide (line 24).

3.3 Process behavior

When it begins its execution, process p_i writes its input value in its entry of the array IN (line 01). If it has an associated AND/OR register $AND\_OR[i]$ (that is, if $i \neq n$), it then prepares the array that will be used as a parameter for its *and* operation on $AND\_OR[i]$ (lines 02–06). The goal of the *and* operation is (1) to let other processes determine if p_i has issued its *and* operation when they read $AND\_OR[i]$ (line 25) and (2) to let them determine if an *or* operation has been issued on $AND\_OR[i]$ before this *and* operation (line 26). The entry $and\_array[1]$ is used to signify that p_i has issued the *and* operation, while the entries $and\_array[2..n]$ are used to determine whether p_i was the first process to issue an operation on $AND\_OR[i]$.

After issuing its *and* operation on $AND\_OR[i]$ (line 07), p_i prepares the array that will be used for the *or* operations on the AND/OR registers $AND\_OR[j]$ with $j \neq i$ (lines 11–18). In each of these registers, the *or* operation may modify 2 bits: the first bit ($AND\_OR[j][i+1]$ if $i < j$, $AND\_OR[j][i]$ otherwise) is used to determine whether p_i issued its *or* operation before the *and* operation by p_j, if the latter has been issued. The second bit ($AND\_OR[j][n+i]$ if $i < j$, $AND\_OR[j][n+i-1]$ otherwise) is used to signify that p_i issued its *or* operation on $AND\_OR[j]$.

Process p_i then determines the value that it will return. For all the AND/OR registers $AND\_OR[j]$, p_i determines whether p_j won its competition, that is, whether p_j was the first process to issue an operation on $AND\_OR[j]$ (lines 25–26). If that is the case, p_i updates its estimate of the value it has to return by setting *output* to $IN[j]$ that contains p_j's input (line 27). If no process has won its competition, p_i returns p_n's input value (line 31).

3.4 Proof of the algorithm

▶ **Lemma 1.** *The decided value is a proposed value.*

Proof. The variable *output* is returned at line 32. If it is not written at line 27, during the loop at lines 23–30, it is written at line 31. When it is returned, *output* thus always contains the value of an entry of the array IN. There are then two cases.

1. The last write of *output* is at line 27.
   The write of *output* at line 27 can only happen if the first bit of the corresponding AND/OR register $AND\_OR[j]$ has been set to 0 (read of $AND\_OR[j]$ at line 24 and condition at line 25) and if the condition at line 26 is respected. This can only happen if p_j, the process to which $AND\_OR[j]$ is associated, has invoked the *and* operation on $AND\_OR[j]$ at line 07, which it does after writing its input value in $IN[j]$ at line 01. The read of $IN[j]$ at line 27 thus returns p_j's input value.
2. The last write of *output* is at line 31.
   Let us note that, except p_n (which doesn't have an associated AND/OR register), every process competes in its own associated AND/OR register (line 07) before competing in any other AND/OR register (line 19). Let us then consider the set of participating processes minus p_n, and the first operation on an AND/OR register by any of these processes. Because operations on AND/OR registers are atomic, the first operation is well defined. Let p_i be the process that issues it.
   Suppose, by way of contradiction, that p_n issued its first operation on $AND\_OR[i]$ after p_i, or not at all. By definition of p_i, when it issues its *and* operation on the AND/OR register $AND\_OR[i]$, no other process has issued an *or* operation on it yet. Note that no other *and* operation will be issued on $AND\_OR[i]$. After this operation, the condition at line 25 is respected. The condition at line 26 is also respected: before

any process issues an *or* operation, for any $x \in [n+1..2n-1]$, $AND\_OR[i][x] = 0$ (initialization of $AND\_OR[i]$). Any subsequent *or* operation by any process p_j will set an entry $AND\_OR[i][n+x-1]$ with $x \in [2..n]$ to 1, but it will also set $AND\_OR[i][x]$ to 1 (lines 13–14 if $j < i$ or lines 16–17 if $j > i$, and *or* operation at line 19).
Any process that reads $AND\_OR[i]$ (line 24) after the first operation by p_i will then observe the conditions at lines 25 and 26 as respected, and will execute line 27. It will then not execute the write of *output* at line 31, a contradiction. In case (2), process p_n must then have issued its first operation on an AND/OR register before any other process. Its write of *IN* precedes its first operation: the read of $IN[n]$ at line 31 thus returns p_n's input value, which concludes the proof of the lemma. ◀

▶ **Lemma 2.** *No two different processes decide different values.*

Proof. The proof relies on the fact that, for each AND/OR register, the result of the competition (whether the process to which it is associated has won or lost) is fixed after the first operation applied on it. All the processes apply an operation on all AND/OR registers before deciding, and thus have the same "view" of the competition.

Let us first note that, before entering the loop at lines 23–30, and thus checking the conditions at lines 25 and 26, any process applies either an *and* or an *or* operation on all AND/OR registers (*and* operation at line 07 or *or* operations at line 19 in the loop at lines 09–21). Consider the AND/OR register $AND\_OR[i]$, for any $i \in [1..n-1]$. Because operations on AND/OR registers are atomic, the first operation on an AND/OR register is well defined. There can then be two cases.

1. The first operation on $AND\_OR[i]$ is an *and* operation.
   Let us first consider the value of $AND\_OR[i][1]$. The *and* operation by process p_i (the only *and* operation issued on $AND\_OR[i]$) sets $AND\_OR[i][1]$ to 0 (lines 04–06 and *and* operation at line 07). The condition at line 25 will then be observed as respected by all the processes which read $AND\_OR[i]$ (line 23 which, for any process, happens after its *and* or *or* operation on $AND\_OR[i]$).
   Let us now consider the condition at line 26, that depends, for any $x \in [2..n]$, on the values of $AND\_OR[i][x]$ and $AND\_OR[i][n+x-1]$. Before any process issues an *or* operation on $AND\_OR[i]$, the value of $AND\_OR[i][n+x-1]$, for any $x \in [2..n]$, is equal to 0 (initialization of $AND\_OR[i]$) and thus the condition at line 26 is respected. Because any *or* operation on $AND\_OR[i]$ happens after the first and only *and* operation on it, after any such *or* operation by a process p_j, both the values of $AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ (if $j < i$, lines 13 and 14) or of $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ (if $j > i$, lines 16 and 17) will be equal to 1. The condition at line 26 will then always be observed as respected.
   Any process that reads $AND\_OR[i]$ at line 24 will then observe the conditions at lines 25 and 26 as respected, and will execute line 27, overwriting the value of *output* with p_i's input value.
2. The first operation on $AND\_OR[i]$ is an *or* operation.
   Before p_i executes its *and* operation on $AND\_OR[i]$, the value of $AND\_OR[i][1]$ is equal to 1. Any process that reads $AND\_OR[i]$ (line 24) before the *and* operation by p_i will then not execute line 27.
   Let us now consider the case of a process that reads $AND\_OR[i]$ after the *and* operation by p_i. Let p_j be the first process that issued an *or* operation on $AND\_OR[i]$. By definition, this operation happened before p_i's *and* operation. The *or* operation of p_j sets the values of $AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ (if $j < i$, lines 13–14)

or $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ (if $j > i$, lines 16–17) to 1 . For any $x \in [2..n]$, p_i's operation then sets $AND\_OR[i][x]$ to 0 (lines 04–06 and *and* operation, line 07). Apart from p_i, p_j is the only process that can modify its corresponding entries of $AND\_OR[i]$ ($AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ if $j < i$, lines 13–14, or $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ if $j > i$, lines 16–17).
Any process that reads $AND\_OR[i]$ after the *and* operation by p_i will then observe $AND\_OR[i][x]$ equal to 0 and $AND\_OR[i][n+x-1]$ equal to 1, for $x = j+1$ if $j < i$ or $x = j$ if $j > i$. It will then not execute line 27.
We then have the following: for any AND/OR register $AND\_OR[i]$, if the first operation on it is an *and* operation, every correct process will execute $output \leftarrow IN[i]$ at line 27. If the first operation on it is an *or* operation, no process will execute $output \leftarrow IN[i]$ at line 27. Every correct process will then have the same sequence of assignments in the loop at lines 23–30. For every correct process, the last assignment to *output* will then correspond to the same entry of IN. By Lemma 1, this entry, say $IN[j]$, always corresponds to the input value of p_j, which concludes the proof of the lemma. ◀

▶ **Lemma 3.** *Every correct process decides.*

Proof. The algorithm does not contain any blocking operation, and the only three loops (lines 04–06, lines 09–21 and lines 23–30) are *for* loops, which terminate after a predetermined number of iterations. Any correct process will then terminate its execution of the algorithm by deciding a value at line 32, which concludes the proof of the lemma. ◀

▶ **Theorem 4.** *The algorithm presented in Figure 2 solves consensus for n processes in the shared memory model extended with shared $(2n-1)$-bits AND/OR registers.*

Proof. The correctness of the algorithm follows from Lemma 1 (validity), Lemma 2 (agreement) and Lemma 3 (wait-free termination). ◀

The following corollary is a direct consequence of Theorem 4.

▶ **Corollary 5.** *Shared x-bits AND/OR registers have consensus number at least $\lfloor \frac{x+1}{2} \rfloor$.*

4 Solving consensus using AND/OR/XOR registers

This section presents an algorithm that solves consensus for n processes using n-bits AND/OR/XOR registers. The algorithm, presented in Figure 3, constitutes the proof that x-bits AND/OR/XOR registers have consensus number at least x.

Differences with the previous algorithm

The main difference resides in the way that the objects are used to decide the output of each competition. The previous algorithm used two bits in each AND/OR register to determine whether the process associated to the object has issued its *and* operation before another given process has issued its *or* operation. The use by the process associated to the AND/OR/XOR register of a *xor* operation allows using a single bit to determine whether this process issued its operation before another given process. This is due to the fact that *xor* operations don't work as multiple assignments: the state of each individual bit after a *xor* operation depends on its previous state, even when it is modified.

Each competition uses an AND/OR/XOR register $AND\_OR\_XOR[i]$ of n bits. The owner p_i uses an *xor* operation, while the other processes use an *or* operation. Process p_i

```
Initially:
  $\forall x \in [1..n-1]$ :
      $\forall y \in [1..n]$ : $AND\_OR\_XOR[x][y] = 0$.

Operation propose$_i(v)$:   % Code for $p_i$ %
IN[i] ← v;
if ($i < n$) then                      % $xor\_array$: array of $n$ bits %
   $xor\_array \leftarrow [1, \ldots, 1]$;
   $AND\_OR\_XOR[i].xor(xor\_array)$
end if;
for $j$ from 1 to $n-1$ do
   if ($j \neq i$) then                % $or\_array$: array of $n$ bits %
      $or\_array \leftarrow [0, \ldots, 0]$;
      if ($i < j$) then
         $or\_array[i+1] \leftarrow 1$
      else   % $i > j$ %
         $or\_array[i] \leftarrow 1$
      end if;
      $AND\_OR\_XOR[j].or(or\_array)$
   end if
end for;
$output \leftarrow \perp$;
for $j$ from 1 to $n-1$ do
   $current \leftarrow AND\_OR\_XOR[j].read()$;
   if ($current[1] = 1$) then
      if ($\nexists x \in [2..n] : current[x] = 0$)
         then $output \leftarrow IN[j]$
      end if
   end if
end for;
if ($output = \perp$) then  $output \leftarrow IN[n]$ end if;
return($output$).
```

Figure 3 An algorithm that solves consensus for n processes using n-bits AND/OR/XOR registers (code for p_i).

modifies (by having the corresponding bits of the parameter of its $xor()$ operation set to 1) the bits $AND\_OR\_XOR[i][1..n]$. Process p_j overwrites (by having the corresponding bit of the parameter of its $or()$ operation set to 1) a single bit $AND\_OR\_XOR[i][x]$, where $x = j + 1$ if $j < i$, and $x = j$ otherwise. The modification of $AND\_OR\_XOR[i][1]$ by p_i allows determining if p_i issued its operation. The modification of $AND\_OR\_XOR[i][x]$ by both p_i and p_j allows determining (a) if p_j issued its operation (in combination with $AND\_OR\_XOR[i][1]$) and (b) which process issued its operation first. The layout of the bits of an AND/OR/XOR register modified during a competition is presented in Figure 4.

▶ **Theorem 6.** *The algorithm presented in Figure 3 solves consensus for n processes in the shared memory model extended with shared n-bits AND/OR/XOR registers.*

Proof. Apart from the differences outlined previously, the algorithm is similar to the algorithm presented in Figure 2, and thus the proof is similar to the proof of Theorem 4. ◀

The following corollary is a direct consequence of Theorem 6.

▶ **Corollary 7.** *Shared x-bits AND/OR/XOR registers have consensus number at least x.*

5 Optimality of the algorithms

This section first presents a proof that x-bits shared AND/OR registers cannot solve consensus for more than $\lfloor \frac{x+1}{2} \rfloor$ processes, when any number of processes can crash. This shows that

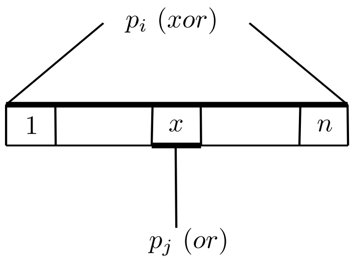

Figure 4 $AND\_OR\_XOR[i]$ (n bits): bits modified by the owner p_i and another process p_j.

the algorithm presented in Section 3 is optimal with respect to the number of processes that can solve consensus and thus that the consensus number of x-bits AND/OR registers is exactly $\lfloor \frac{x+1}{2} \rfloor$. It then presents a proof that x-bits shared AND/OR/XOR registers cannot solve consensus for more than x processes, when any number of processes can crash, and thus that the algorithm presented in Section 4 is optimal and that the consensus number of x-bits AND/OR/XOR registers is exactly x.

5.1 Preliminaries

The following concepts are used in the proofs of Lemmas 9 and 12.

Steps and executions

During a step, a process applies an atomic operation on a base shared object. An execution consists of a set of initial local states (one for each process) and a sequence of steps. A correct process takes steps in the execution until it decides (returns a value). A faulty process stops taking steps before deciding.

Prefixes and extensions

A prefix of a given execution consists of the set of initial local states of the execution and a prefix of the sequence of steps that constitutes it. An extension of a prefix starts with the prefix and is completed by a (possibly empty) sequence of steps by processes of the system. The first step that a process can take after a prefix is defined by the algorithm it is executing.

Indistinguishable executions

For a given process, an execution A is indistinguishable from an execution B if its local state is the same in both executions. Note that this is the case if the values it obtained from its operations are the same in A and in B. If it is correct and finishes its execution of the algorithm, it will then have to decide the same value in A and in B.

Valence

The prefix P of an execution is univalent if, in any execution of which P is a prefix, the correct processes decide the same value v. This means that in P, the value that processes will decide is already fixed. The same concept can apply to an extension of a prefix. A prefix that is not univalent is multivalent.

Modification of a bit

As was explained in Section 2.1, *and*() and *or*() operations can be seen as a series of assignments. The *and*() operation can only modify the bits of the object on which it is applied that correspond to the entries of its parameter that are equal to 0. The *or*() operation can only modify the bits that correspond to the entries of its parameter that are equal to 1. The *xor*() operation cannot be seen as a series of assignments, but it only modifies the bits that correspond to the entries of its parameter that are equal to 1.

In a given execution, we then say that a process can modify a given bit of a shared AND/OR or AND/OR/XOR register if its next step is (a) an *and*() operation and the corresponding bit of its parameter is set to 0, or (b) an *or*() or *xor*() operation and the corresponding bit of its parameter is set to 1.

The following lemma is similar to results that have been shown in various other papers (see e.g. [9]). It is used in the proofs of Lemmas 9 and 12. Due to page limitations, its proof is not presented here.

▶ **Lemma 8.** *Any multivalent prefix of an execution of any wait-free consensus protocol in the shared memory model extended with shared objects has a multivalent extension such that:*
1. *the next operation of any process forces a decision,*
2. *all these operations are on the same object X,*
3. *all these operations modify the state of X and*
4. *X is not a read/write register.*

5.2 Consensus number of shared AND/OR registers

▶ **Lemma 9.** *Shared AND/OR registers of x bits have consensus number at most $\lfloor \frac{x+1}{2} \rfloor$.*

Proof. Let us consider the empty prefix of an execution in which the processes propose at least two different values. By Lemma 8, there is an extension E such that E is multivalent, the next step of any process imposes a decision and these steps all modify a single AND/OR register X (and thus, they are *and* or *or* operations). We will prove that X cannot have less than $2n-1$ bits, and thus that in a system of $n > \frac{x+1}{2}$ processes in which they communicate only through read/write registers and AND/OR registers of x bits, consensus is impossible.

- *For each process p_i, there must be at least one bit of X such that only p_i can modify it.*

Consider the extension E such that, if p_i's step is the next step, then the value v_1 must be decided and, for some process p_j, if p_j's step is the next step, then the value $v_2 \neq v_1$ must be decided. Because E is multivalent, p_j must exist. Consider now the extension of E in which p_i executes one step first and then crashes, then p_j executes one step, then all other processes (if any) execute one step. Suppose now, by way of contradiction, that there is no bit of X such that only p_i can modify it. Let us recall that *and*() and *or*() operations can be viewed as assignments (Section 2.1). The *and*() and *or*() operations that follow p_i's step then overwrite its modifications to X. After these operations, the remaining processes cannot distinguish the previous extension of E, in which v_1 must be decided, from the one in which p_i did not execute its step (all other steps being executed in the same order), in which v_2 must be decided. A contradiction which proves that, for each process p_i, there must be at least one bit of X such that only p_i can modify it.

- *For each pair of processes p_i and p_j such that the next step of p_i imposes the decision of v_1 and the next step of p_j imposes the decision of $v_2 \neq v_1$, there must be at least one bit of X such that only p_i and p_j can modify it.*

Consider again the extension of E in which p_i executes one step first, then p_j, then all other processes. Suppose, again by way of contradiction, that there is no bit of X such that only p_i and p_j can modify it. All the modifications by p_i and p_j of a common bit have then been overwritten by the other processes (if p_i and p_j are the only processes in the system, $x \leq 2$ and thus, because of the previous item, they cannot modify the same bit). After all these steps, the processes cannot distinguish the previous extension of E, in which v_1 must be decided, from the one in which p_i took its step after p_j, in which v_2 must be decided. A contradiction which proves that, for each pair of processes p_i and p_j such that the next step of p_i imposes the decision of v_1 and the next step of p_j imposes the decision of $v_2 \neq v_1$, there must be at least one bit of X such that only p_i and p_j can modify it.

Let us now partition the processes in sets $P_1, \ldots, P_k$ such that, if the next operation after E is by a process in P_ℓ, the decided value has to be v_ℓ, with $\ell \neq m \Rightarrow v_\ell \neq v_m$. Because E is multivalent, there are at least two such sets and thus $k \geq 2$. The number of pairs of processes belonging to different sets is then equal to the number of edges of a complete k-partite graph in which the processes of P_ℓ correspond to the vertices of the ℓ^{th} part of the graph. For $k \geq 2$, complete k-partite graphs are connected. The minimum number of edges of a connected graph of n vertices being $n-1$, the number of pairs of processes belonging to different sets is at least $n-1$. The number of bits needed for all pairs of processes that impose different values is then at least $n-1$.

The AND/OR register X must then have at least (1) one bit per process, and (2) one bit per pair of processes that can impose different values, giving a minimum of $n+n-1 = 2n-1$. Consensus is thus impossible in a system in which processes communicate only through read/write registers and AND/OR registers of strictly less than $2n-1$ bits, and thus shared AND/OR registers of x bits have consensus number at most $\lfloor \frac{x+1}{2} \rfloor$. ◀

▶ **Theorem 10.** *Shared AND/OR registers of x bits have consensus number exactly $\lfloor \frac{x+1}{2} \rfloor$.*

Proof. The proof follows from Corollary 5 (lower bound) and Lemma 9 (upper bound). ◀

5.3 Consensus number of shared AND/OR/XOR registers

▶ **Lemma 11.** *Let S be a set and A a family of subsets of S. If $|A| > |S|$, there exists a non-empty $A' \subseteq A$ such that every element of S appears in an even number of elements of A'.*

Proof. Let $n = |S|$. Define the signature sig_B of a family B of subsets of S as an n-entries vector, in which $sig_B[i] = 0$ if the i^{th} element of S appears in an even number of elements of B, and 1 otherwise (any arbitrary total order can be used on the elements of S). There are then 2^n possible signatures for subfamilies of A.

Because $|A| > n$, there are at least 2^{n+1} different possible subfamilies of A, of which at least $2^{n+1} - 1 > 2^n$ are non-empty. By the pigeonhole principle, there must then be at least two different non-empty subfamilies B and C of A such that $sig_B = sig_C$.

Let $B' = B - (B \cap C)$ and $C' = C - (B \cap C)$. B' and C' are then disjoint. Because $sig_B = sig_C$, we also have $sig_{B'} = sig_{C'}$. Because $B \neq C$, at least one of B' and C' is non-empty.

Let $A' = B' \cup C'$. A' is then non-empty. Because B' and C' are disjoint and have the same signature, every element of S appears in an even number of elements of A', which concludes the proof of the lemma. ◀

▶ **Lemma 12.** *Shared AND/OR/XOR registers of x bits have consensus number at most x.*

Proof. Like in Lemma 9, let us consider the empty prefix of an execution in which the processes propose at least two different values. By Lemma 8, there is an extension E such that E is multivalent, the next step of any process forces a decision and these steps all modify a single AND/OR/XOR register X (and thus, they are *and*, *or* or *xor* operations). We will prove that X cannot have less than n bits, and thus that in a system of $n > x$ processes in which they communicate only through read/write registers and shared AND/OR/XOR registers of x bits, consensus is impossible.

Note that the case in which the next step of any process is an $and()$ or $or()$ operation is already covered by Lemma 9. We will thus consider that in E, the next step of at least one process is a $xor()$ operation.

Let P_{xor} be the set of processes such that their next step in E is a $xor()$ operation. Let p_i and p_j be processes in P_{xor} such that the next step of p_i in E imposes the value v_i and the next step of p_j imposes the value v_j. The execution in which p_i executes one step first (and thus v_i must be decided), then p_j executes a step, cannot be distinguished by any process from the execution in which p_j executes its step first (and thus v_j must be decided) and then p_i. All the next steps in E of processes in P_{xor} must then impose the same value v_{xor}.

- *There must be at least $|P_{xor}|$ bits of X that can only be modified by the processes in P_{xor}.*

Consider the extension of E in which first, the processes in some $P'_{xor} \subseteq P_{xor}$ execute one step (in any order) and crash. Then, a process p_i such that its next step in E imposes a value $v_i \neq v_{xor}$ executes one step, then all the remaining processes (if any) also execute one step. Suppose, by way of contradiction, that strictly less that $|P_{xor}|$ bits are modified only by the processes in P_{xor}. We will show that in this case, there exists a non-empty P'_{xor} such that this execution (in which v_{xor} must be decided) is indistinguishable by the other processes from the execution in which all the processes in P_{xor} crash before executing a step, and p_i imposes the value $v_i \neq v_{xor}$ (note that if $|P_{xor}| = 1$, no bit is modified by the single process in P_{xor} and this is trivially verified).

Let S be the set of bits modified only by the processes in P_{xor}. By Lemma 11, there exists a non-empty set $P'_{xor} \subseteq P_{xor}$ such that every bit in S is modified by an even number of processes in P'_{xor}, and thus appear unmodified after all the processes in P'_{xor} apply their operations. Because all the other bits of X are modified afterwards by $and()$ or $or()$ operations, the state of X is the same as if the processes in P'_{xor} hadn't executed their operations, a contradiction.

- *For each process p_i not in P_{xor}, there must be at least one bit of X such that the only process not in P_{xor} that can modify it is p_i.*

The reasoning is the same as in Lemma 9. Consider the execution in which first, a process $p_i \notin P_{xor}$ executes one step, imposing a value $v_i \neq v_{xor}$, and then crashes. Afterwards, a process $p_j \in P_{xor}$ executes one step, followed by all the other processes (if any) that also execute one step. If there is no bit such that the only process not in P_{xor} that can modify it is p_i, then this execution is indistinguishable by all the other processes from the execution in which p_i crashes before executing its step and in which v_{xor} is imposed, a contradiction.

There must then be at least $|P_{xor}|$ bits modified only by the processes in P_{xor} and at least $n - |P_{xor}|$ bits modified only by the processes not in P_{xor}. To solve consensus for n processes, AND/OR/XOR registers must then have at least n bits, which concludes the proof of the lemma. ◀

▶ **Theorem 13.** *Shared AND/OR/XOR registers of x bits have consensus number exactly x.*

Proof. The proof follows from Corollary 7 (lower bound) and Lemma 12 (upper bound). ◀

5.4 Other variants

Until now, we have only considered shared AND/OR and AND/OR/XOR registers. This section briefly discusses other variants. Shared AND (resp. OR or XOR) registers offer, in addition to the *read*() operation, a single operation *and*() (resp. *or*() or *xor*()). Shared OR/XOR (resp. AND/XOR) offer *or*() (resp. *and*()) and *xor*() operations, but not the *and*() (resp. *or*()) operation.

Shared AND, OR and XOR registers

Operations of a single type (*and*(), *or*() or *xor*()) are commutative: the effect of various *and*() (resp. *or*() or *xor*()) operations on the state of an object, when only these operations are issued, does not depend on the order in which they have been issued.

Consider, as in Lemmas 9 and 12, an execution E in which the next step by process p_i imposes the decision of v_i, and the next step by process p_j imposes the value $v_j \neq v_i$. Neither p_i nor p_j can distinguish the extension of E in which p_i takes a step first, then p_j, from the extension in which p_j takes its step before p_i. This implies that shared AND, OR and XOR registers cannot solve consensus, even for two processes. Consequently, their consensus number is 1.

Shared OR/XOR and AND/XOR registers

The algorithm presented in Figure 3 uses *or*() and *xor*() operations, but not the *and*() operation. It can easily be modified to use *and*() operations instead of *or*() operations. Lemma 12 trivially applies if the objects only offer *or*() and *xor*() (or *and*() and *xor*()) operations, and thus shared OR/XOR and AND/XOR registers have the same consensus number as shared AND/OR/XOR registers.

6 Conclusion

Bitwise operations are a common tool in sequential computing but, until now, they had not been considered in the distributed context. This paper studied their synchronization power by presenting the following contributions.

- The concept of shared AND/OR and AND/OR/XOR registers. A shared AND/OR register contains an array of x bits, to which it allows applying atomically bitwise AND and OR operations. A shared AND/OR/XOR register additionally offers the XOR operation.
- A wait-free algorithm that solves consensus for $\lfloor \frac{x+1}{2} \rfloor$ processes using x-bits shared AND/OR registers. This algorithm constitutes a lower bound on the consensus number of shared AND/OR registers.

- A modification of the previous algorithm that solves consensus for x processes using x-bits shared AND/OR/XOR registers. This algorithm constitutes a lower bound on the consensus number of shared AND/OR/XOR registers.
- A proof that x-bits shared AND/OR registers cannot solve consensus for more than $\lfloor \frac{x+1}{2} \rfloor$ processes. This constitutes an upper bound on the consensus number of shared AND/OR registers, and thus proves that the previous bound is tight.
- A proof that x-bits shared AND/OR/XOR registers cannot solve consensus for more than x processes. This constitutes an upper bound on the consensus number of shared AND/OR/XOR registers, and thus proves that the previous bound is tight.

These results show that shared AND/OR registers have consensus number $\lfloor \frac{x+1}{2} \rfloor$ and that shared AND/OR/XOR registers have consensus number x. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems.

Acknowledgements. The author would like to thank Dmitry Kozlov for his help in the proof of Lemma 11.

References

1 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 218–227, 2006. doi:10.1145/1146381.1146415.

2 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pages 159–170, 1993. doi:10.1145/164051.164071.

3 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 91–100, 1993. doi:10.1145/167088.167119.

4 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. doi:10.1006/inco.1993.1043.

5 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

6 Eli Gafni and Petr Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011. doi:10.1007/s00446-011-0142-8.

7 Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. The synchronization power of coalesced memory accesses. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):939–953, 2010. doi:10.1109/TPDS.2009.135.

8 Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 276–290, 1988. doi:10.1145/62546.62593.

9 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

10 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

11 Damien Imbs and Michel Raynal. A note on atomicity: Boosting test&set to solve consensus. *Information Processing Letters*, 109(12):589–591, 2009. doi:10.1016/j.ipl.2009.02.004.

12 Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997. doi:10.1145/263867.263888.

13 Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, 4:163–183, 1987.

14 Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 159–175, 1989. doi:10.1145/72981.72992.

15 Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000. doi:10.1137/S0097539796307698.