A completeness theorem for a class of synchronization objects

(Extended Abstract)

Yehuda Afek Eytan Weisberger*

Hanan Weisman†

Computer Science Department, Tel-Aviv University, Israel 69978.

Abstract

We study a class of synchronization objects in shared memory concurrent systems, which we call *common2*. This class contains *read-modify-write* objects that commute (e.g. *fetch-and-add*), or overwrite (e.g. *swap*). It is known that this class is contained in the consensus number 2 class of objects [Her91a], and most of the commonly used objects with consensus number 2 are included in it. We show that any object in the *common2* class can implement any other object in the class, in a system with an arbitrary number of processes. In fact we show that the objects in *common2* are implementable from any object with consensus number 2.

The *common2* class is in particular interesting since the strongest objects most contemporary processors support are in this class, e.g. either *test-and-set* or *swap* while concurrent operating systems constructs installed on any machine may usually rely on any other primitive in this class.

Additional implications of our result are: (1) The existence of fault-tolerant self implementations of objects in *common2*, (2) improvements in the efficiency of randomized constructions of several objects in *common2* from read/write registers, and (3) low contention constructions of objects in *common2*.

*also with Motorola Semiconductor Israel Ltd.

†also with Scitex Corporation Israel Ltd.

1 Introduction

In his seminal paper [Her91a], Herlihy established the existence of a hierarchy of wait-free concurrent data objects that classifies objects according to their ability to solve k-consensus, that is, consensus among k asynchronous processes. An object has consensus number k if any number of this object and of read/write registers can be used to implement a k-consensus protocol, but cannot be used to implement a $k+1$-consensus protocol. Thus objects with higher consensus number cannot be deterministically implemented by employing objects with lower consensus numbers. At the bottom level are the weakest objects with consensus number 1, e.g. read/write atomic registers, while at the top are objects such as compare-and-swap, whose consensus number is ∞. In [Her91a, Plo88] it is shown that any wait-free shared object can be implemented from any object in the top level of the hierarchy, that is they have presented a universal construction for any sequentially specified wait-free object.

The synchronization primitives in the second level of the hierarchy include many synchronization primitives that are commonly used in practice, such as, *test-and-set*, *fetch-and-add*, *fetch-and-increment*, *queue*, and *swap*. Systems like Encore-Multimax, Sequent-Symmetry, and SGI's MIPS-based multiprocessor support either *test-and-set* or *swap* [Ber91]. In addition, many algorithms for queue management and primitives of concurrent operating system are based on either one of these objects [HW90, GLR83, PS85], but not necessarily on the one supported by the system employed. This situation raises the question whether these software functions and algorithms can run on any of those machines, e.g. can a package that is based on *fetch-and-add* [GLR83] run on a system that supports only *test-and-set* [Ber91]. Yet, aside from very trivial implemen-

tations, no implementation of objects at level 2, from any other at that level, have been provided (trivial ones are *test-and-set* from *swap*, or *fetch-and-increment* from *fetch-and-add*). Herlihy posed the following open question: "Can *fetch-and-add* implement any object with consensus number 2 in a system of three or more processes?" [Her91a]. Although the universal construction implements any sequentially specified, n-process shared object, it is necessarily based on objects with consensus number n.

A *read-modify-write* (RMW) object, over a set of functions F and a register r, is generically defined as follows:

read-modify-write(f,r)**: return**(v) $\quad \{f \in F\}$
$v := r$
$r := f(r)$
return(v)

Let F be a set of functions indexed by an arbitrary set S. The set F is commuting if for all values v and all i and j in S, $f_i(f_j(v)) = f_j(f_i(v))$. The set is overwriting if either $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$. E.g., *fetch-and-add* is a classical example of an object that applies commuting functions (add 5, add 2, ...) while *swap* and *test-and-set* apply overwriting functions.

Consider the class of objects, called *common2*, that contains (1) any *read-modify-write* object that applies commutative functions, and (2) any *read-modify-write* object that applies overwrite functions. In [Her91a] it is shown that *common2* is contained in the class of objects with consensus number 2. The class *common2* includes most of the commonly used objects with consensus number 2.

In this paper we give a completeness theorem for the *common2* class of objects. That is, we provide a reduction from any object in *common2* to any other object in the class. Moreover, we show that any object with consensus number 2 can implement any object in *common2* (in polynomial number of steps per operation). We have thus resolved the above open question with respect to the class *common2*.

Our result has three additional implications: First, an open problem of [AGM+92] and [JCT92] is resolved by showing that there are fault tolerant implementations for the n processes objects in *common2* from any set that contains some faulty objects. This is the result of combining the constructions of this paper with the constructions for two processes given in [AGM+92, JCT92].

The second implication of our result are new randomized implementations for the objects in *common2* from read/write registers. This is the result of combining the constructions given herein with the known randomized constructions of 2-process consensus and *test-and-set* [ADS89, Asp90, AH90, SSW90, Her91b, AGTV92]. Some of those constructions are more efficient than what is previously known, e.g. $O(n^2)$ steps for *fetch-and-increment*, and $O(n^3)$ for *swap* and *fetch-and-add*, as oppose to $O(n^4)$ for either one in [Her91b].

The third implication of our result are low contention implementations of objects in *common2*. This follows from the fact that the basic building block of all our constructions are 2-process synchronization objects. Thus, if any of those are implemented by a critical section (e.g. in a system with only read/write registers) then, within our implementation, at most two processes will contend on each critical section. Moreover, since all our implementations are wait-free, the slow down, or failure of one critical section will not affect the progress of all processes aside from the two connected with the faulty section.

Low contention constructions were also provided in [AHS91, HSW91], where *fetch-and-increment* is constructed by a *counting network* of *balancers*. A *balancer* is an object whose consensus number is two, and which essentially behaves as *fetch-and-increment mod 2*. The contention level of our implementations is different than that of the counting networks. Though each balancer in the counting network has only two ports, the two processors that access each of these ports are different at different points in the execution, while in our implementations always the same process accesses each port of a two processor building block. Furthermore, we conjecture that it is impossible to adapt the concept of counting networks to implement functions such as *fetch-and-add* (e.g. by replacing each balancer with a more sophisticated object).

The uniqueness of our implementations is that we implement objects that can be accessed by more processes than their consensus number. To understand why the implementation of n-process objects with consensus number 2 from similar two process objects is more difficult than the analogous implementations between objects with consensus number n, one should understand the difference in the computational power of the two classes. Intuitively, any n-consensus object has the capability to return in the response to each access a total order of any subset of preceding accesses, such that the returned order is consistent with the linearization of the accesses. That is, which process accessed the object first, which process second, third etc. The capability of consensus number 2 objects is totally different. They cannot return as a response to an access a total order of previous accesses in the linearization order. Yet, at least the objects in *common2* have the capability either to return in a response to an access the unordered subset of all previous accesses, or to return the access that is linearized just before this access. (Read/write registers cannot give any of these orders atomically, but can give an order on the operations in a non-atomic manner [DS89]).

From the universal construction of [Her91a, Plo88] it follows that any consensus number k object is univer-

sal in a system with k processes. Thus, any consensus number 2 object for two processes can be implemented from any other. Our implementations are based only on 2 process objects, hence, by the above transformation, each of our implementations can use any consensus number 2 object as the base object.

In this paper we give a sequence of implementations, where basic tools developed in one implementation are used as building blocks in later implementations. For ease of exposition we present the implementations in increasing level of difficulty. We start with the relatively simple implementations of n process *test-and-set* (Subsection 3.1), and n process *fetch-and-increment* objects from 2 process swap registers (Subsection 3.2 and 3.3). We then continue to implement n process RMW for commutative functions and n process RMW for overwrite functions. The RMW for commutative functions is described in terms of *fetch-and-add* (Subsection 3.4) and the RMW for overwrite functions in terms of *swap* (Subsections 3.5 and 3.6). The implementation of each *fetch-and-add* or *swap* operations requires $O(n^3)$ steps on read/write registers and 2 process swap objects. Each of the *swap* and *fetch-and-increment* is presented in two steps, first a *single-use* implementation is presented, and then a *multi-use* that is based on the single-use construct. In a single-use object each process is allowed to make at most one access, while in the multi-use there is no restriction on the number of operations performed by a process. The multi-use implementations of the *swap* is only sketched in this abstract due to space limitations.

2 Model definitions, and notations

We follow the model and the notation of [Her91a]. A system consists of n processes ($P_1, P_2, \ldots, P_n$ each a sequential thread) communicating through shared data structures, called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate that object. Some steps of the sequential threads of each process are operations on the objects. In each operation the process issues a request to make the operation, and subsequently it receives a response which could be just a signal if no value should be returned by the object, or the value returned by the object as a response to that request. Specification and proof of correctness for all the objects, those used, and those implemented are based on the *linearizability* condition of Herlihy and Wing [HW90].

Definition 1 *A* single-use *object is one in which each process may perform an operation at most once.*

In a multi-use *object each process may repeatedly access the object an arbitrary number of times.*

Notation: An object foobar that may be accessed by at most 2 processes is denoted *2P-foobar*. We denote a single-use object *foobar* with *SU_foobar*. Single-writer/multi-reader atomic registers are denoted *swmr*, multi-writer/multi-reader *mwmr* and, single-writer/single-reader *swsr*.

3 Implementations

3.1 The test-and-set and tournament objects

The *tournament* object is a generalization of the *test-and-set* object. While the *test-and-set* object returns to each `test&set` operation either 0 or 1, the *tournament* object returns to one process (the winner) 0 and to any other process the identity of another process. The specification of the *single-use* tournament is such that, the *id* returned by each of the operations impose an acyclic partial order on the operations.

More specifically, consider a single-use tournament. Each of the n processes that share the *tournament* object accesses it through the operation `compete`. Let $P_i \mapsto P_j$ if the response to the `compete` operation by P_i was j. Then, the relation $\mapsto$ defines an acyclic partial order on the processes and only one operation, the winning operation, is responded by a 0. Moreover, in the sequential specification, the winning operation is linearized before any other operation, and if $P_i \mapsto P_j$ then, the `compete` operation of P_i is linearized after that of P_j. In the journal version of the paper this specification is naturally extended to the multi-use by adding the `reset`. However, all the constructions in this abstract may need only the single-use version.

To implement the *tournament* object a competition is carried out between the processes such that only one process wins and all the others lose. Following [PF77, AGTV92] processes contest each other in a binary tree tournament. Only one process can win the tournament, and it is the winner in the implementation, while all other processes lose. In each level of the tournament we match the winners of the previous level in pairs (according to a fixed binary tree) that compete each other. However, the winner of one subtree cannot know the identity of the winner of the sibling subtree, even if such exist (such knowledge would imply a stronger primitive). Thus, to contest the winner of one subtree a process contests *all* the processes in that subtree. Therefore, our solution places a 2 process swap (*2P_swap*) register between any pair of processes. To contest in a *2P_swap* each process swaps a certain value. The process that read *null* is the winner while

the loser reads the value swapped in by the winner. A process loses as soon as it loses in any contest.

The problem with this solution is that one process could lose, while the eventual winner has not yet started to play. This kind of a scenario violates the linearizability condition of n-process tournament. To overcome this problem we follow [AGTV92] and use an additional register, called *gate*. Each process that starts a `compete` operation prevents any process that has not yet started, from starting, by closing the *gate*. That is, each process starts by reading the *gate*. If it is *null*, it writes its identity into the gate and continues, otherwise, it returns with the non *null* value it read from the gate.

The code for the `compete` operation is given in Figure 1.

```
shared:  tour = [swap[i,j]: 1 ≤ i < j ≤ n;
{data structures of tournament}
          gate: mwmr register];
operation compete(tour) returns(id);
    {For P_i}
    if tour.gate ≠ null return (tour.gate);
{ Gate is closed - lost to id in gate}
    tour.gate := i; { Closes the gate.}
    for level = 1 to log n do
    for every process P_j in P_i's
        sibling subtree in level level do
        t := Swap(swap[min(i,j), max(i,j)], i);
        if (t ≠ null) return (j);
    { Lost to P_j.}
    od od
    return (0); { Won the competition}
end compete
```

Figure 1: Operation `compete`

The description above implements a single-use tournament. Its generalization to an implementation of a bounded multi-use test-and-set (with reset) follows the technique of [AGTV92] (using sequence numbers within the swap register) and are thus left out from this extended abstract. For the proof of correctness (that also follows that citation) we only give the key lemma:

Definition 2 *Let $P_k \mapsto P_i$ if P_ks* `compete` *operation returns i.*

Note the following simple observations:

1. There is at most one winner in each tournament (the process that returns 0). It follows since each winner must have won all the other processes.
2. If a process loses at level l, it does not compete at any level l', such that $l' > l$.

Lemma 3.1.1 *The relation $\mapsto$ defines an acyclic partial order between the operations on the tournament.*

Proof: Assume to the contrary that there is a cycle of processes related by $\mapsto$, such that, $P_1 \mapsto P_2 \ldots P_{k-1} \mapsto P_k \mapsto P_1$. For $k = 2$ the proof is straightforward. We prove it for $k \geq 3$. Let us label each of the $P_i \mapsto P_j$ arrows by P_i's value of the variable *level* when P_i lost and broke out of the for-loop, the label is 0 if P_i lost in the gate. Clearly no two consecutive $\mapsto$ arrows in the cycle are labeled by 0. We claim that, if $P_i \mapsto P_{i+1}$ is labeled by $l \neq 0$ then $P_{i-i} \mapsto P_i$ is labeled by $l' < l$. Since P_{i+1} won over P_i in level l, P_{i-1} could not have proceed to level l and to contest P_i there, thus P_{i-1} lost to P_i in a level smaller than l. ∎

Theorem 1 *The code in Figure 1 correctly implements a tournament object.*

Proof outline: By Lemma 3.1.1 there is exactly one winner. The use of the gate insures linearizability. Wait-freedom easily follows from the code. ∎

3.2 Single-use fetch-and-increment

The sequential specification of *fetch-and-increment* applies the function $f(r) = r+1$ in the generic RMW in the introduction. The implementation of n process single-use fetch-and-increment from n process test-and-set is very simple: there is a linear array T of n test-and-set registers. To perform a fetch-and-increment each process starts performing test-and-set on the array starting from $T[0]$ until it wins. A process that won in $T[i]$ $(i = 0, \ldots, n-1)$ returns i.

```
operation single-use fetch-and-increment
    Shared T[0..n-1]: n-process T&S ;
        for i = 0 to n-1 do
            if T&S(T[i]) = 0 then return (i) od
```

Figure 2: Single use fetch-and-increment.

The single-use fetch-and-increment gives a multi-use implementation by using an infinite array of test-and-set registers. While it is easy to make this solution bounded, it is not wait-free. Fast processes can repeatedly win an arbitrarily long sequence of registers in the array while a slow process trails behind.

Correctness: The implementation is wait-free, since the T&S's are wait-free and each operation performs at most n of those. Each process returns a unique value from $\{0, \ldots n-1\}$ since only one process wins in each location. The linearizability follows since when a process returns j the operations that would return k, $k < j$ have already started, by the linearizability of test-and-set.

3.3 Multi-use fetch-and-increment

This subsection presents a bounded wait-free multi-use fetch-and-increment. We give in detail an unbounded space implementation, its transformation into a bounded space implementation is relegated to the journal version of the paper. Assume, as in the end of the previous subsection, an infinite linear array of test-and-set registers $T[0,\ldots]$. The idea behind the wait-free implementation is that fast processes leave "holes" (untouched test-and-set registers) in the array for the slow processes that trail behind.

More specifically, each process approximates a position p in the array, such that it is guaranteed to find an unset register between $T[p]$ and $T[p-K]$ (K will be shown to be bounded by n). Then the process will start performing test-and-set on the array starting from $T[p]$ going down in the indices until it wins. Still the implementation is such that in any complete run a consecutive interval of registers from $T[0]$ is set, i.e., no holes are left behind in a complete run.

Consider the following unbounded number of operations solution: Each process in the implementation maintains a swmr (single-writer/multi-reader) register, called *my_inc* in which it updates the total number of increments it had performed so far. In the first step of an operation a process increments its *my_inc*. To estimate a "good" position from which to start test-and-setting, each process then takes a snapshot of all the *my_inc* values, and takes the sum of the values in the snapshot as its starting position. This guarantees that each process will find an unset register in a finite number of steps, but still there is no bound on the number of test-and-set registers it will have to test before winning one. The reason being that between the time a process increments its *my_inc* and the time it takes a snapshot, an unbounded number of operations can take place by other processes leaving a "hole" for that process an unbounded number of registers away from its eventually entrance point.

To overcome this problem each process keeps a copy of a "relevant" snapshot for each other process (all snapshots are only of the *my_inc* values). A process computes its entrance point according to a set of snapshots one of which is guaranteed to be in close vicinity (in the run) to the point in which the process has incremented its *my_inc*.

The implementation is described in Figure 3. The shared data structures used by the implementation are as follows:

$T[0,\ldots]$ An infinite array of test-and-set registers (in the journal version of the paper it is folded into a bounded size array).

$my\_inc[1..n]$ a swmr register, each counts the number of increments that the corresponding process has started.

$my\_minSS[i]$, $i = 1,\ldots n$, a swmr register one per process, each containing a list of snapshots, at most one snapshot for each other process.

```
operation Fetch-and-increment
    {at process P_i}
    Shared  T[1..∞]: n-process T&S ;
            my_inc[1..n]:
                swmr snapshot memory ;
            my_minSS[1..n]:
                swmr ; initialized to null ;
    local   entry, S, M ;
    my_inc[i] := my_inc[i] + 1
    S := snapshot-scan{ my_inc registers}
    my_minSS[i] :=
            relevantSS{S ⋃ my_minSS[i]}
    M := minSS{⋃_j my_minSS[j], i}
    entry := sum(M)
    while T&S( T[entry]) ≠ 0 do
            entry := entry − 1 od
    return (entry)
```

Figure 3: Multi-use fetch-and-increment.

In addition, each process keeps a local copy of a snapshot and the *my_inc* registers have all the necessary fields to take a snapshot of them [AAD+90].

The sum of a snapshot, $sum(ss)$, is the summation of the n *my_inc* values in ss. For a given set of snapshots L, we define $max(L,i)$ as the maximum value of process P_i's *my_inc* in all the snapshots in L. $minSS(L,i)$ is then defined, as the snapshot in L with the minimum sum among those in which P_i's value is $max(L,i)$. We define the function `relevantSS`:

$$\texttt{relevantSS}(L) = \{minSS(L,i) | i = 1,\ldots,n\}.$$

that is, $my\_minSS[i]$ is a set of snapshots, containing for each process P_j the earliest snapshot in which P_j was observed by P_i with the maximal $my\_inc[j]$ that P_i ever observed. Each process estimates its entry point after reading all the entries in the array my_minSS and taking the minimum value snapshot in which it appears with its current *my_inc* value.

Correctness

Definition 3 inc_point(i, j)*: Consider the sequence of all the write operations of the shared variable my_inc in (Figure 3). Since the writes are atomic, this is a total order.* inc_point(i, j) *is the index of the write operation,* $my\_inc[i] := j$ *in this sequence.*

Definition 4 start(i, j)*: is the value of the local variable entry that* P_i *calculates in the operation*

`multi-use-fetch-and-increment`. *when* $my\_inc[i] = j$ *(i.e., in its j's increment operation). This is the index of the first test-and-set register that* P_i *tries to set in its j's increment.*

Definition 5 inc(i, j) *is the value returned in* P_i*'s j's increment operation. That is, the index of the test-and-set register which* P_i *wins when* $my\_inc[i] = j$.

Lemma 3.3.1 . *Every fetch-and-increment operation of any process* P_i *terminates with entry* ≥ 0.

Proof: If process P_i that starts with initial entry at h and fails to find an unset test-and-set when $entry = 0$ then it must be that more than h $fetch-and-increment$ operations have accessed the array in the interval between 0 and h. We consider the shortest prefix of a run in which there is an l such that more than l operations accessed the registers in the interval 0-l. Let l' be the largest l for which more than l operations touched the first l registers. At least one of the l' operations must have their first entry point above l' and it must have lost at $l'+1$, a contradiction to the definition of l' is thus reached. ∎

Lemma 3.3.2
$inc\_point(i,j) \leq start(i,j) < inc\_point(i,j) + n$

Proof: The first entry point is a summation of a snapshot that includes the $my\_inc[i]$. By the atomicity of snapshot and the my_inc registers the left side of the inequality holds. It is bounded to be at most n registers away from the inc_point since otherwise some process must have done two such increments. The snapshot taken by that process in the first of these two increments, includes the most recent value of $my\_inc[i]$, and is kept in the appropriate $my\_relevatSS$. ∎

Lemma 3.3.3 *If in a run of the system there is a process* P_i *and a j, such that,* $inc\_point(i,j) > t \geq inc(i,j)$ *then in this run there must be a process* P_l, $l \neq i$, *and an m such that,* $inc\_point(l,m) \leq t < inc(l,m) \leq start(l,m) < t+n$

Proof outline: If for all processes P_r and w such that $inc\_point(r,w) \leq t$ get $inc(r,w) \leq t$, then at least $t+1$ (those processes and P_i) access the array between 0 and t, contradicting Lemma 3.3.1, and hence the lemma follows. ∎

Lemma 3.3.4 *Consider a run of the system in which there is a process* P_i *and a j, such that,* $inc(i,j) > inc\_point(i,j)$ *and a process* P_l, *such that* $inc\_point(i,j) \leq start(l,x) < inc(i,j)$. *Then, in this run, process* P_l *in its x's increment operation had not written its* $my\_minSS[l]$ *in the operation* `multi-use-fetch-and-increment`, *before process* P_i *had started to perform the minSS computation of its j's increment operation.*

Proof outline: If P_l had written its $my\_relevant[l]$ before P_i started to perform $minSS$, then P_i would have computed $start(i,j) \leq start(l,x)$. ∎

Lemma 3.3.5 *In any run of the implementation, if an* `fetch-and-increment`(i,x) *starts after an operation* `fetch-and-increment`(j,y) *ends then,* $inc(i,x) > inc(j,y)$.

Proof outline: Assume to the contrary that $inc(i,x) < inc(j,y)$. Since `fetch-and-increment`(i,x) starts after `fetch-and-increment`(j,y) ends, it follows that $inc\_point(i,x) > start(j,y) > inc(i,x)$. From Lemma 3.3.3 it follows that there is a process P_{k_1} and a b_1 such that, $inc\_point(k_1,b_1) \leq start(j,y) < start(k_1,b_1)$, by taking t in that lemma to be $start(j,y)$. If $inc\_point(i,x) > start(k_1,b_1)$ then this argument can be repeated to show that there is a P_{k_2} and a b_2 such that $inc\_point(k_2,b_2) \leq start(k_1,b_1) < start(k_2,b_2)$. This argument is repeated until we arrive at a P_{k_l} such that $inc\_point(i,x) \leq start(k_l,b_l)$, and $inc\_point(k_l,b_l) \leq start(k_{l-1},b_{l-1}) < start(k_l,b_l)$. By Lemma 3.3.4 P_j finished to write its $my\_minSS[j]$ in the operation multi-use-fetch-and-increment after P_{k_1} had started to perform $minSS$ computation of its b_1's increment operation. Apply this lemma repeatedly to get: that $P_{k_{z-1}}$ finished to write its $my\_minSS[k_{z-1}]$ after P_{k_z} had started to perform $minSS$ computation of its b_z's increment operation, for $z = 2, \ldots, l$. P_{k_l} started his $minSS$ computation after $inc\_point(i,x)$ hence P_j wrote $my\_minSS[j]$ after $inc\_point(i,x)$ in contradiction to the assumption. ∎

From the above lemmas we get:

Theorem 2 *The code in Figure 3 correctly implements fetch-and-increment and each operation in it terminates in at most* n^2 *accesses to the shared memory, and at most n* `test&set` *operations on the test-and-set array.*

3.4 Multi-use Fetch-and-add

The *fetch-and-add* object applies the function $f(v,r) = r+v$ in the generic RMW of the introduction. The implementation of the fetch-and-add object, presented in this paper, is a general one, it enables each operation to return an unordered set of operations linearized before it. Thus, it can be used to implement any commutative function, e.g. fetch-and-multiply.

3.4.1 The wigwag object

We introduce a new primitive, called *wigwag* that serves as a key building block in the implementations of n process *fetch-and-add*. It is very similar to the trap object introduced in subsection 3.5.1 and is also used by processes to signal each other. The wigwag object is constructed from atomic read/write registers and hence its

consensus number = 1. Intuitively, in the wigwag object some processes signal others whether they have passed through with some value, and some processes may try to prevent others from passing through with a specific value. The difference between the trap and the wigwag is that here it is possible that process p has blocked process q, but p cannot be certain about it. The object has two operations: `wig-Pass` and `wig-Block` (see Figure 4):

1. `wig-Pass`(w, i, val) - in which process P_i tries to signal all other processes that it is passing though wigwag w with value val. The operation returns false or true, whether some other process has already prohibited P_i from passing through w with value val or not (by writing a different value in their corresponding register).

2. `wig-Block`$(w, i, vec[1..n])$ - in which P_i signals each other process P_j to stop if it tries to wig-Pass through wigwag w with a value different from $vec[j]$. The operation returns a boolean vector, $bvec[1..n]$ indicating for each other process P_j whether P_j might have already wig-passed w with a different value than $vec[j]$, or not. (if $bvec[j] =$ true then P_j is certainly blocked in w, but if $bvec[j] =$ false then P_j may be blocked but is not certain to be blocked).

```
operation wig-Pass(w, i, val):
                                  (returns(boolean));
  for k := 1 to n do
      w.WWPass[k, i]:= val ; od
  for k := 1 to n do
      if (w.WWBlock[k, i] ≠ null)∧
          (w.WWBlock[k, i] ≠ val)
      then return (false) ;
  od
  return (true);

operation wig-Block(w, i, vec[1..n]):
                              returns(bool − vector);
  for k := 1 to n do
      w.WWBlock[i, k]:= vec[k] ; od
  for k := 1 to n do
      bvec[k] := (w.WWPass[i, k] = null)∨
          (w.WWPass[i, k] = vec[k]) ;
  od
  return (bvec[1..n]);
```

Figure 4: The wig-Pass and wig-Block operations, of the *wigwag*

The *wigwag* is implemented by two, two dimensional arrays of *swsr* atomic registers, $WWBlock[i, j]$, and $WWPass[i, j]$, $1 \leq i, j \leq n$. All registers are initialized to *null*. In operation wig-Pass(w, i, val) process P_i writes val to $WWPass[1..n, i]$ of w and then reads $WWBlock[1..n, i]$. If all the reads have returned either *null* or val then it returns true (successfully passed the wigwag), otherwise it returns false. In operation wig-Block$(w, i, vec[1..n])$ process P_i writes $vec[j]$ to $WWBlock[i, j]$ $j = 1, ..., n$ and then reads $WWPass[i, 1..n]$. If the read of $WWPass[i, j]$ returns either *null* or $vec[j]$ then $bvec[j]$ is returned true.

3.4.2 The fetch-and-add construction

Overview Essentially the fetch-and-add works as follows: Assume we have an unbounded linear array of abstract cells. In each cell each process is registered with some value, and during the run at most one process wins each cell, i.e., becomes the winner of that cell. In each fetch-and-add operation a process wins a higher indexed cell. Before winning a cell, a process ensures that it is registered in all the cells from the previous one it won to the current one (exclusive) with the summation of all its inputs excluding the current input. After winning a cell a process returns the summation of all the values with which processes are registered in the cell it won. In all the cells above the last one won by p, p is registered by default, with the summation of all the inputs to all the fetch-and-add operations it had initiated (including the last one).

In our implementation each process counts the number of operations it performs in a swmr register $my\_inc[i]$. The total sum of its first k operations is recorded in another swmr register $total[k, i]$. The unbounded array of abstract cells is arranged in an unbounded array of rows, called *floors*, each *floor* (row) contains n cells. That is, the first n cells in the first floor, cells $n + 1$ to $2n$ in the second floor, etc. Each cell is a wigwag object ($WW[., .]$ in Figure 5) and an n-process test-and-set object ($TST[., .]$). With each floor a unique snapshot of the my_inc array ($SS[.]$ in the Figure) is associated.

In its first step each process increases its own my_inc counter and updates its $total$ variable that holds the summation of all the inputs to it operations so far. Then it computes its entry level in a similar way to the *fetch-and-increment* operation, using the $minSS$ mechanism, and writes the entry's snapshot in $SS[entry]$. The process starts descending from its $entry$ floor and finds after at most n steps a floor with a registered snapshot, and that snapshot contains its previous my_inc value. From that level the process tries to climb back to its $entry$ floor passing through all the wigwags of all the floors on its way. A process may be stopped on its way up in two ways: Either another process stops it by blocking a wigwag, or it reaches the last wigwag of its entry floor (level). From the place in which it was stopped it starts

a seek phase to find a unique place in the floor in which it was stopped. In the seek phase it finds a place where it can win the test-and-set, and be sure that other processes would not pass through that place with values that can change the return value of its *fetch-and-add* operation. In each step of the seeking phase it tires to block the current wigwag, then it writes the index of its own place (level) in the floor in a variable called *level* and then reads the other processes positions in the floor. The process finds now a group (LEFT) of processes that are assured to be in its level or bellow, or processes that it has blocked in the current wigwag. When the process figures out that the size of the group, LEFT is equal to its current level, it is sure that no other processes (fetch-and-add operations) will join this group. The process then tries to win the current T&S. If it wins it returns the summation of all the fetch-and-add operations that are accounted in LEFT. When it losses the T&S operation it repeats the above procedure in the previous level (i.e., one level below).

Correctness

Definition 6 *FA_i^k the k'th fetch-and-add operation of process i.*

Lemma 3.4.1 *Consider the sequence of values of the variable r in the last while loop in each fetch-and-add operation of some process. Then this sequence is monotonically increasing.*

Definition 7 *$LEFT(i,r,t)$ is the group LEFT that process P_i computes after writing $level[r,i] := t$.*

Lemma 3.4.2 *For any r and t and process P_i, $|LEFT(i,r,t)| \leq t$.*

Proof: The size of $LEFT(i,r,t)$ is at most n so the lemma holds for $t = n$. Assume to the contrary and consider the shortest prefix of a run in which there is an i, r and t, $(t < n)$ such that $|LEFT(i,r,t)| > t$. There must be at least one process P_z in $LEFT(i,r,t)$ that had already written $level[r,z] = t+1$ in the past (otherwise P_i would not compute $LEFT$ at this point). Let P_j be the last process in $LEFT(i,r,t)$ that wrote $level[r,j] = t+1$. For every $P_l \in LEFT(i,r,t)$ either P_j had written: $WW[r,t+1].WWBlock[j,l] := SS[r][l]$ or P_j read $level[r,l] \leq t+1$. Thus P_j had calculated $|LEFT(j,r,t+1)| \geq t+1$ and must have tried to win $TST[r,t+1]$. P_j lost the $T\&S(TST[r,t+1])$ (because it is in $LEFT(i,r,t)$) to another process, say P_q. P_q had written $level[r,q] = t+1$ before it tried to win $T\&S$, it is not a member of $LEFT(i,r,t)$, so either P_j had calculated $|LEFT(j,r,t+1) > t+1$ or P_q had calculated $|LEFT(q,r,t+1) > t$ in either case this calculation was terminated before the calculation of $LEFT(i,r,t)$ started - a contradiction. ∎

```
operation Fetch-and-add(input)     {at process P_i}
  Shared array TST[1..∞, 1..n]: of n-process T&S ;
         array WW[1..∞, 1..n]: of wigwag ;
         array my_inc[1..n]: swmr snapshot memory;
         array SS[1..∞]: of mwmr registers ;
       {each records an n-tuple snapshot of the my_inc
                snapshot memory, only one value will
                              ever be written in each.}
         array level[1..∞, 1..n], total[1..∞, 1..n]:
                                   of swmr (integer);
         array my_minSS[1..n]: of swmr: init null;
  local  entry, blocked, S, M, faa := 0 ;
  total[my_inc + 1, i] := total[my_inc, i] + input ;
  my_inc[i] := my_inc[i] + 1 ;
  S := snapshot-scan{ my_inc registers} ;
  my_minSS[i] := relevantSS{S ⋃ my_minSS[i]} ;
  M := minSS{⋃_j my_minSS[j], i} ;
  entry := r := sum(M) ;
  SS[entry] := M ;
  while SS[r][i] ≠ my_inc[i] − 1 do r := r − 1 od
  blocked := false ;
  while ¬blocked ∧ (r < entry) do
    r := r + 1 ;
    t := 0 ;
    while ¬blocked ∧ (t ≤ n) do
       t := t + 1 ;
       blocked := ¬ wig-Pass(WW[r, t], i, my_inc[i] − 1) ;
    od
  od
  while true do
    bvec := wig-Block(WW[r, t], i, SS[r]) ;
    level[r, i] := t ;
    LEFT := {j|bvec[j] ∨ (level[r, j] ≤ level[r, i]) } ;
    if |LEFT| ≥ t ;
    then if (T&S(TST[r, t]))
          then
            for k = 1 to n do
                if (k ∈ LEFT) ∧ (k ≠ i)
                then faa := faa + total[SS[r][k], k] ;
                else  faa := faa + total[SS[r][k] − 1, k] ;
            od
            return(faa)
    t := t − 1 ;
  od
```

Figure 5: Multi-use fetch-and-add.

Lemma 3.4.3 *Every fetch−and−add−operation can terminate after its process wins a T&S.*

Proof: We consider a run in which there is a process P_j that fails to win $TST[r,1]$. Thus, by Lemma 3.4.1, there were at least two processes (P_j and say P_q) in either $LEFT(j,r,1)$ or $LEFT(q,r,1)$. A contradiction to lemma 3.4.2.

Theorem 3 *Let x be the input value of the FA_i^k operation, in which P_i wins test-and-set $TST[r,t]$. Consider another process P_j in FA_j^h that wins test-and-set $TST[l,m]$.*
If ($l = r$ and $m < t$) or ($l < r$), then FA_j^h operation will not include x in its summation. If however ($l = r$ and $m > t$) or ($l > r$), then the output of FA_j^h will include x.

Proof: For $l > r$, $SS[l][i] \geq k$, P_i does not pass there (through the wigwags) with k, so each process that wins in these floors returns P_i k'th value (x). For $l < r$, either $SS[l][i] < k$ or P_i succeeds to propagate $k-1$ through all of floor l wigwags. For $l = r$, P_i passed through the wigwags $1..t-1$ with the value $k-1$ and never wrote $level[r,i] \leq t$. It means that P_i was not in the $LEFT$ group of the processes that win in $TST[l,1..t-1]$ and they do not include x in their summation. At the time that process P_j wins $TST[l,m]$, $m > t$, it is sure that P_i is included in $LEFT(j,l,m)$ group, since if P_i is not included in $LEFT(j,l,m)$ then there will be some other process P_w such that $|LEFT(w,l,m)| > m$ a contradiction to Lemma 3.4.2. ∎

3.5 Single-use swap

In the sequential specification of the *swap*, each operation returns the input of the operation preceding it. Thus each complete run defines a total order on the operations.

3.5.1 The trap object

We introduce a new primitive, called *trap* that serves as a key building block in the implementations of n process *swap*. Intuitively, the trap is a synchronization object which some process is trying to *pass* through while some process might try to *block* it. The object has two operations: `pass` and `block`:

1. `Block`(t,j) - by process P_i, where it tries to block process P_j in trap t. The operation returns either *null* (success) or a value, in case P_j has already passed through trap t and registered value v in it.
2. `Pass`(t,j,v) - by process P_i, where P_i tries to pass process P_j in trap t and leave value v in it. The operation returns either *false* or *true* depending on whether process P_j has already blocked P_i in t, or not.

```
operation Block(t, j): returns(value);
        local: v: value;
        v:= Swap(t[j, i], "blocked");
        return (v);

operation Pass(t, j, v): (returns(boolean));
        if Swap(t[i, j], v) = "blocked"
        then return (false);
        else return (true);
```

Figure 6: The Block and Pass operations, of the *trap*

The *trap* is implemented by a two dimensional array of *2P_swap*, $SW[i,j]$, $1 \leq i,j \leq n$. All objects are initialized to *null*. In order to succeed in a Pass(j,v) operation process P_i must Swap its value v in the $SW[i,j]$ *2P_swap* object before P_j has swapped "blocked" in there. In order to block(j) process P_i must Swap "blocked" at $SW[j,i]$ before P_j swaps in a value. The code for both operations is given in Figure 6.

We define a procedure (Figure 7), Pass_all on the *trap* object, which is used in the implementations of the *swap* object.

```
procedure Pass_all(t, v):
                returns(boolean);
        for k := 1 to n do
            if ¬ Pass(t, k, v)
            then return (false); od
        return (true);
```

Figure 7: The Pass_all procedure of the *trap*

3.5.2 The single-use swap construction

To implement the single-use-swap, we use a sequence of n cells ordered from *level* 1 to n. Each process starts the operation by registering its input value in a swmr register it owns. Each process goes over the cells in order, starting from 1, trying to capture one of them. Each cell can be captured by at most one process. Once a cell was captured by a process, that process must identify a process in the cell below it, whose value it will return. However, the eventual winner in the cell below may have not yet determined. We use the *trap* object defined above to solve this problem.

Each level (cell) can be captured only by one process. After failing to capture a cell in a level, processes progress to the next level by passing through a trap object that is associated with the current level. Once a cell is captured, the capturing process goes to the *trap* associated with the level below it, in an attempt to block a process there and to return its value.

```
operation SU_Swap(v) returns(value)
    { For P_i. }
    shared input[1,..,n] array of value
           TOUR[1,..,n] array of tournament;
           TR[1,..,n] array of trap;
    local  t, level, lost_to

    input[i] := v
    {P_i records its input in a shared variable}
    for level = 1 to n do
        t := compete(TOUR[level]);
    {P_i competes at tournament TOUR[level].}
        if t = 0 {P_i won.} then
          return FindValue(level - 1, lost_to)
    {P_i returns a value from trap level - 1.}
    else if ¬ pass_all(TR[level],t)
    {P_i was blocked at trap level.}
          then return FindValue(level, t)
    {P_i returns a value from trap at level.}
           lost_to := t
    od
end SU_Swap

function FindValue(level, id) returns(value)
    if level = 0 then return (null)
    else repeat
      x := Block(TR[level, id)
      if x ≠ null then id := x
    until (x = null)
    return (input[id])
end FindValue
```

Figure 8: The single-use Swap operation

To ensure that at most one process captures each cell, we use the *tournament* object from Subsection 3.1. Note that our implementation of the `compete` operation provides a process P_i that lost at level l with the identity of another process P_j. P_j's `compete` operation at this level was linearized before P_i's operation. Upon losing, process P_i performs a Pass_all$(t_l, v = j)$ procedure, thus registering P_j's identity in $trap_l$. Let P_w be a process that lost to P_i at level l and won the *tournament* at level $l+1$. If P_w has already blocked process P_i in $trap_l$ then P_w returns P_i's value which it finds in a swmr register of P_i, while P_i will now try to block P_j and return P_j's value. Otherwise, if P_i has successfully passed the trap then P_w's block operation has failed and P_w will try to block P_j instead.

The code for the single-use swap operation is given in Figure 8.

Correctness

Definition 8 *$ret(i)$ - The index of the trap with which process P_i called function* `FindValue` *(the value of the first parameter with which* `FindValue` *was called).*

Note that, more than one process may have the same $ret(i)$.

Definition 9 *Let $P_j \prec P_i$ if, the $Block(trap, j)$ operation by P_i has successfully returned null.*

Lemma 3.5.1 *In any complete run, the relation $\prec$ defines a total order between the operations.*

Proof: We need to prove two claims, first, that aside from the one process which returns *null* in the `SU_swap` operation (i.e., won the `compete` operation in *level* 1), for any other process P_i there is exactly one process P_j such that $P_j \prec P_i$. Second, we prove that for any process P_i , such that $P_i \prec P_j$ there is exactly one such process, P_j. The first claim follows immediately from the code, since every process calls FindValue() exactly once.

The second claim is proved by contradiction, that is, assume that $P_i \prec P_j$ and $P_i \prec P_k$. If $ret(j) < ret(k)$ then P_j has blocked P_i in trap $ret(j)$ so P_k could not have *lost to* P_i at *tournament* $ret(k)$ because P_i did not compete there and P_k could not have attempted to block P_i at that trap. Thus, $ret(j) = ret(k)$. Processes perform the first Block operation on a process which satisfy the $\mapsto$ (lost to) relation ,defined in Subsection 3.1. If they fail to block, then they try the process which their winner lost to and so forth. By Lemma 3.1.1 the relation $\mapsto$ is an acyclic order between processes. Since, the $\prec$ relation correlates in reverse to the transitive closure of the $\mapsto$ relation, the claim follows by contradiction. ∎

Definition 10 *Let $\prec'$ be the transitive closure of $\prec$.*

Lemma 3.5.2 *For any run of the algorithm, if $P_i \prec' P_j$ then P_i's operation had started before P_j's operation has ended.*

Proof: Assume to the contrary that $P_i \prec' P_j$ but P_i's operation had started after P_j's operation has already ended. We consider 2 cases: (1) $c = ret(i) < ret(j)$, and (2) $c = ret(i) = ret(j)$. In case (1), when P_i started the operation, the gate at *tournament* c was already closed. Therefore, no process could have *lost to* P_i at *tournament* c i.e. there does not exist a P_k, such that $P_k \mapsto P_i$. Thus, P_i would successfully complete the procedure Pass_all at that trap and proceed to the next tournament at cell $c+1$. P_i can't win the `compete` operation at that cell since the gate there is closed too ($c+1 \leq ret(j)$). Therefore P_i could not have returned a value at trap c - a contradiction.

Consider case (2); by observing the code in Figure 8 all the Block operations of a trap in a certain level are done sequentially. That is if P_k has successfully blocked P_l, then process P_l starts it's own Block operation after P_k had finished. Since $P_i \prec' P_j$, P_j was blocked in trap c before P_i was blocked there. Thus, when P_j finished the Block operation the gate at cell c was already closed. Since $P_i \prec' P_j$, there is some process , P_k, such that $P_i \prec P_k$ and according to Lemma 3.5.1 there is a process P_l such that $P_l \mapsto P_i$. Thus, P_i could not have observed a closed gate at *tournament* c. But according to our assumption, P_i starts after P_j had finished, and should have read a closed gate - a contradiction. ∎

Lemma 3.5.3 *For any run α of the algorithm, the order $\prec$ is a correct SU_swap linearization.*

Proof: We now map the total order defined by $\prec$ into a sequential run. The run is correct since, if $P_i \prec P_j$ then according to Lemma 3.5.1 P_j is the only process related in that manner to P_i and according to the code, P_j returns the value that P_i had registered. The run is linearizable since, according to Lemma 3.5.2, if P_i is linearized before P_j then P_j's operation could not have ended before P_i's had started. ∎

From the above lemmas we get:

Theorem 4 *The code in Figure 8 correctly implements a SU_swap object.*

3.6 Multi-use swap

This subsection describes the construction of a multi-use *swap* object from *tournament* and *trap* objects defined in the previous sections. We show an unbounded space complexity implementation, the bounded implementation will be discussed in the journal version.

Using the implementation of the single-use *swap* by extending its array of cells to be unbounded, we face two problems: First, since processes start at the first cell of the array, and the number of operations is unbounded, they are apt to perform an unbounded number of steps. Second, processes can be *slow* when competing in a tournament (thus always lose it), but *fast* when passing through a trap (other processes never succeed in blocking them) and therefore, be *starved* forever. To overcome these difficulties, we modify the previous scheme as follows:

First, a process finds a cell to start its operation, as close as possible to the cell that it eventually wins. This is done by collecting the list of cell indices currently in use by each process. These indices are declared by processes every time they advance to a new cell. The base cell for computation in an operation, is the cell with the maximum index seen in the collect.

Second, as before, processes try to capture a cell. However, once a process fails in a competition, it does not try to compete in the cell above. Rather, it goes to a new set of cells in which fewer processes compete. The linear array is thus extended to a *tree* of infinite *degree* but of *depth* of $n+1$. The nodes of the tree represent the cell indices and are labeled and ordered by a *preorder* scan of the tree. We define *rank* on the nodes by their distance from the leaves, e.g. a leaf is ranked 1 where the *root* is ranked $n+1$. Processes start each operation in a cell of *rank* 1. If more than one process compete in a cell, then only one process wins and the rest proceed to cells of rank 2 and so forth.

Once a process has captured a cell it looks for a value to return in the subtree under the captured cell. The remaining details are omitted due to space limitation.

The time complexity of the operation is bounded since all the primitives used are wait-free objects and the number of compete operations can be shown to be $O(n)$.

4 Conclusions

This abstract leaves several open questions. First, to extend the class *common2* to include more or all the objects with consensus number 2, or to prove that the class *common2* is included but not equal to the class of objects with consensus number 2 (An object has consensus number 2 if 2 processes can reach consensus by using any number of this object and any number of read/write registers). More specifically, can *common2* include objects that may apply either commute functions or overwrite functions (e.g. fetch-and-add with a set operation). The question of the optimality of any of the above constructions in time and/or space is open.

Acknowledgments: We are in debt to Eli Gafni for many insightful remarks. We thank Yishay Mansour, Nir Shavit, and Gideon Stupp for many helpful discussions. In particular Gideon's patients was inspiring.

References

[AAD+90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proc. of the Ninth Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 1–13, August 1990.

[ADS89] Hagit Attiya, Danny Dolev, and Nir Shavit. Bounded polynomial randomized consensus. Extended Abstract, January 1989.

[AGM+92] Y. Afek, D. S. Greenberg, M. Merritt, , and G. Taubenfeld. Computing with faulty shared memory. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *Proceedings of the 6th International Workshop on Distributed Algorithms: Springer-Verlag LNCS*, November 1992.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, pages 281–294, September 1990.

[AHS91] J. Aspnes, Maurice Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proceedings of the 23rd annual ACM Symposium on Theory of Computing*, May 1991.

[Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proc. of the Ninth ACM Symp. on Principles of Distributed Computing*, pages 325–331, August 1990.

[Ber91] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, September 1991.

[DS89] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proc. of the 21st Annual ACM Symposium on Theory of Computing*, 1989.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[Her91a] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[Her91b] M. P. Herlihy. Randomized wait-free concurrent objects. In *Proc. of the Tenth ACM Symp. on Principles of Distributed Computing*, pages 11–22, 1991.

[HSW91] Maurice Herlihy, N. Shavit, and Orli Waarts. Linearizable counting networks. In *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 526–535, October 1991.

[HW90] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[JCT92] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. In *Proc. of the 33rd IEEE Annual Symp. on Foundation of Computer Science*, October 1992.

[PF77] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 91–97, 1977.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus.* PhD thesis, M.I.T., August 1988.

[PS85] J. L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, 1985.

[SSW90] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proc. of SODA 90*, December 1990.