The Instancy of Snapshots and Commuting Objects *

Yehuda Afek Eytan Weisberger †
Computer Science Department,
Tel-Aviv University, Israel 69978.

Abstract

We present a sequence of constructions of commuting synchronization objects (e.g., fetch-and-increment and fetch-and-add) in a system of n processors from any two processor synchronization object whose consensus number is two or more [Her91a]. Each implementation in the sequence uses a particular type of shared memory snapshot, as a building block. Later implementations in the sequence are based on higher quality snapshots. The first implementation of a fetch-and-increment uses the standard atomic snapshot concept, introduced in [AAD+93, And90], while the last construction in the sequence, of fetch-and-add, is based on the immediate snapshot concept introduced in [BG93b]. This last construction also yields an implementation of a stronger snapshot which we call Write-and-snapshot. In addition this work solves an open question of Borowsky and Gafni by presenting an implementation of a multi-shot immediate snapshot object.

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

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 ([Jay93]).

*A preliminary version of this paper was presented in PODC '93.
†Motorola Semi-Conductors Israel.

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 (which are supported by Encore-Multimax, Sequent-Symmetry, and SGI's MIPS-based multiprocessor [Ber91]). In addition, many algorithms for queue management and primitives of concurrent operating system are based on either one of these objects [HW87, GLR83, PS85], but not necessarily on the one supported by the system employed. This situation raises the question: Does an implementation of an object (e.g., queue) that is based on n-processor fetch-and-add [GLR83] implies the existence of an implementation of the same object based on 2-processor test-and-set [Ber91] and read/write registers. Yet, aside from very trivial implementations, no implementation of objects at level two, 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 having consensus number two 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**$(v)\{f \in F\}$
  $v := r$
  $r := f(r)$
  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.

In this paper we present constructions of any read-modify-write shared synchronization object that supports *commutative* operations in a system with an arbitrary number of processors from any other two processor synchronization objects whose consensus number is two or more.

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 two 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, an 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 two objects is totally different. They cannot return as a response to an access a total order of previous accesses in the linearization order (because that would imply infinite consensus number). Yet, at least the objects in the *common2* class of synchronization objects that we have introduced in [AWW93] 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]). The *common2* 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 two class of objects [Her91a] and any object in this class can be implemented from any other object in the class.

From the universal construction of [Her91a, Plo88] it follows that any consensus number k object is universal in a system with k processes. Thus, any two process object can be implemented from any consensus number two object. Our implementations are based only on two process objects. By this transformation, each of our implementations can use any consensus number two object as the base object.

All the constructions in this paper use n process test-and-set and read/write registers as their base objects. The construction of the n process test-and-set register from any two process object with consensus number two or more can be found in [AWW93, Wei94]. In this paper we present a sequence of constructions each of which combines a particular type of snapshot algorithm with a linear array of test-and-set objects, as follows:

Single-use fetch-and-increment **object**: Constructed from n n-process test-and-set objects. Each *fetch&increment* operation takes at most $O(n^2)$ primitive operations on the base objects of the implementation (when considering the implementation from 2-process objects) (Section 3.1).

Non-linearizable multi-use fetch-and-increment **object**: This construction uses an unbounded array of n-process test-and-set objects in conjunction with atomic snapshots. The time complexity of this construction is unbounded (Section 3.2).

Linearizable multi-use fetch-and-increment **object**: In this construction we replace the regular atomic snapshot by a new type of snapshots, called proximate-snapshot, obtaining a linearizable implementation of fetch-and-increment register in which the time complexity of each operations is $O(n^2)$ (Section 3.4).

Single-use fetch-and-add **object**: This construction uses n test-and-set registers in conjunction with the single use (i.e., one shot) immediate-snapshot of Borowsky and Gafni [BG93b]. The time complexity of the resulting construction is again $O(n^2)$ operations on either two processor two consensus objects or read/write registers (Sections 4 and 4.2).

Multi-use fetch-and-add **object**: This construction is based on a multi-shot immediate-snapshot in conjunction with an unbounded array of test-and-set registers. The complexity of the resulting algorithm is $O(n^3)$ operations on read/write registers and $O(n^2)$ *test&set* per operation (Sections 4 and 4.3). An important by product of this construction is the proof that the construction of multi-use immediate-snapshot from read/write registers is possible.

The space complexity of the multi-use fetch-and-increment and multi-use fetch-and-add constructions are unbounded in two ways; In the number of registers used in the constructions and in the size of the registers assumed. In Section 5 we describe a method by which the number of registers used in the constructions can be made bounded. However, the size of the registers cannot be bounded since there is no bound on the number of *fetch&increment* operations applied to the object. That is, the size of the constructed multi-use object is inherently unbounded. Although we managed to bound the space complexity of the constructions still, the main point is in the possibility of constructing

any commuting synchronization object from any other object whose consensus number is 2 and not in the complexity of the construction.

2 Model definitions and notations

We assume the standard model of shared memory as in [Her91a]. A *concurrent system* consists of a collection of *processes* that communicate through *typed shared memory objects*. Processes are *sequential* that is, each process applies a sequence of *operations* to objects in the system. An operation on an object is defined by two events: (1) *invocation* to the object, and (2) *response* by the object. There are no *fairness* assumptions concerning the processes, i.e., a process may halt or operate in different speeds. In particular, processes can not detect whether other processes have halted.

2.1 State Machine and Sequential Specification

More formally, we model an object *type* using a Mealy state machine [Mea55]. We use the notations of [AGMT92] to define a sequential specification via the Mealy machine.

An object $\mathcal{O}$ shared by n processes can be specified via a state machine in which the state transitions are labeled by the invocations and responses of operations performed by the processes.

Definition 1 *A* sequential specification, *is a quintuple* $\{states, start, I, R, \delta\}$ *where:*

1. *states is a (finite or infinite) set of* states.
2. $start \subseteq states$ *is a set of* initial states,
3. $I = (Inv_1, ..., Inv_n)$ *is an n-tuple of sets, where each* Inv_i *is a set of symbols denoting the* operation invocations *by process* i. *Let* $Inv = \cup_i Inv_i$.
4. $R = (Res_1, ..., Res_n)$ *is an n-tuple of sets, where each* Res_i *is a set of symbols denoting the* operation responses *for process* i. *Let* $Res = \cup_i Res_i$.
5. *Define the set of operations by process* i *on* $\mathcal{O}$ *to be* $Op_i = Inv_i \cdot Res_i$, *all the two-symbol strings of invocations and responses by* i, *and let* $Op = \cup_i Op_i$. *Then* $\delta \subset states \times Op \times states$ *is the* transition relation.

This state machine denotes a set of (finite and infinite) strings, obtained by concatenating the edge labels along the state transitions. The *sequential runs* of $\mathcal{O}$ are the (finite and infinite) prefixes of these strings. (Taking prefixes allows runs to end with a pending invocation, that is, an inv_i by some process i with no succeeding response, res_i, by i).

Operations on shared objects are required to be total: for every state $s \in states$, every process i, and every $inv_i \in Inv_i$, there exists $res_i \in Res_i$ and $s' \in states$ such that $(s, inv_i \cdot res_i, s') \in \delta$.

The sequential specification allows operations to be specified as atomic state transitions. However, in asynchronous concurrent systems operations have duration and can overlap in time. This is modeled by allowing the interleaving of invocations and responses by different processes, so that between the invocation and response by one process may be any number of invocations or responses by other processes. Thus, concurrent runs of the object are modeled as elements of $(Inv + Res)^\infty$.

Specific correctness conditions constrain these runs by relating them to those in the sequential specification.

For example, the *linearizable* [HW87] runs of an object $\mathcal{O}$ are specified as follows. Given a string $\alpha \in (Inv + Res)^{\infty}$, define a partial order, $\prec_\alpha$ on the events in α, (the distinct occurrences of symbols in α), such that $a \prec_\alpha b$ if and only if either (1) both a and b are invocations or responses events of the same process (have the same subscript) and a appears before b in α, or (2) a is a response that precedes the invocation b, in α. Then α is linearizable if there is a sequential run β of $\mathcal{O}$, containing exactly the events in α and such that the total order $\prec_\beta$ is an extension of $\prec_\alpha$.

2.2 Implementations

It suffices to consider an implementation of an n-process high-level object $\mathcal{O}$ from a set of primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$ as a set of procedures for each process. Invocations and responses of these procedures are identified with those in the specification of $\mathcal{O}$. The procedures are allowed to do local computation, and communicate only by making invocations and receiving responses from the primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$. Hence, runs of the implementation of $\mathcal{O}$ consist of sequences containing invocations and responses to the high-level object $\mathcal{O}$, local steps of the procedures, and invocations and responses to the primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$.

The runs are constrained in the obvious way to respect the control flow of the individual procedures, allowing arbitrary, asynchronous interleaving between the threads of distinct processes. Moreover, the subsequences of invocations and responses to each primitive object must satisfy the specification of that object. Given these constraints, the subsequences of high-level invocations and responses to $\mathcal{O}$ must in turn satisfy the specification of $\mathcal{O}$.

The implementations are required to be *wait free*, thus any single high-level operation *op* (procedure invocation) must terminate, regardless of the steps taken by any other high-level operations, provided the local actions and low-level operations of *op* are allowed to progress [Lam77].

This informal notion of implementation may be made precise using any of several formalisms [HW87, Her91a, Lam86].

The step (time) complexity of an implementation of a high-level operation is the *worst case* number of atomic (primitive) operations each process performs when completing the operation on the (high-level) object. Define $C(i, op, r)$ as the step complexity of operation *op* of process P_i in run r.

An implementation I of an object A is *bounded wait free* if there exists N such that, for every run r of I, for every operation *op* of r and for every $i \leq n$, $C(i, op, r) \leq N$.

Two characteristics of objects are used in this work:

Definition 2 *The* access number *of an object is the maximum number of processes that can perform an operation on the object concurrently.*

Definition 3 *A* single-use *object is one in which each process may perform at most one operation; in a* multi-use *object each process may repeatedly access the object an arbitrary number of times.*

We pay special attention to two objects fetch-and-increment and fetch-and-add. The sequential specification of fetch-and-increment applies the function $f(r) = r + 1$ in the generic RMW of the introduction. The fetch-and-add object applies the function $f(val, r) = r + val$ in the generic RMW of the introduction.

3 Proximate Snapshots and Fetch-and-increment

3.1 Single-use-fetch-and-increment

The implementation of n process single-use-fetch-and-increment from n process test-and-set is very simple (see Figure 1): there is a linear array T of n test-and-set registers. To perform *fetch&increment* each process starts performing *test&set* on the array starting from $T[1]$ until it wins. a process that won in $T[j]$ $(j = 1, \ldots, n)$ returns $j - 1$.

```
operation single-use-fetch&increment
    Shared T[1..n]: array of n-process test-and-set ;
    for j = 1 to n if test&set(T[j]) = 0 then return (j-1)
```

Figure 1: Single-use-*fetch&increment*.

Correctness: The implementation is wait-free, since the *test&set* operations are wait-free and each *fetch&increment* 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 x the operations that would return k, $k < x$ have already started, by the linearizability of test-and-set.

3.2 Non linearizable multi-use fetch-and-increment

The single-use-fetch-and-increment suggests a multi-use implementation by using an infinite array of test-and-set registers. While it may be possible to make this solution space 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.

To overcome this problem we suggest a different approach. Instead of ascending from the test-and-set register at some location up to the location at which the *fetch&increment* operation wins in a *test&set*, a process approximates a position in the linear array of test-and-sets such that, it is guaranteed to find an unset test-and-set register while descending down from that location.

The problem of approximating such a position is best captured through the *write and snapshot* problem.

Definition 4 *The* write and snapshot *problem is to update (write) a register in a shared array of single-writer-multi-reader (*swmr*) registers and then to take a snapshot of the array such that a bounded number of other updates and snapshots have taken place between the update and the snapshot.*

The straightforward approach, using an ordinary snapshot algorithm, does not provide any bound on the number of updates that may take place between the update of a processor and its subsequent snapshot. On the other hand, in the fetch-and-increment object that we implement here, a process updates a shared counter and gets back the value of the counter immediately after (before) its update. The *write and snapshot* problem in the read/write model was first raised by Borowsky and Gafni [BG93a], and Saks and Zaharoglou [SZ93]. Borowsky and Gafni has studied this problem and

defined the *immediate snapshot* problem for which they provided a one-shot solution in [BG93b]. However, here we need a multi-shot *write and snapshot* that would bound the number of other updates between the two operations.

Let us first consider the straightforward solution, using an ordinary snapshot scan algorithm [AAD+93, And93, AR93], where there is no bound on the number of updates between the write of a processor and its subsequent snapshot. In this solution each process maintains a private counter, called *my_inc*, in a swmr register (see Figure 2). In each count operation a process adds one to its local *my_inc*. Then, to approximate the global counter it snapshot scans the array of *my_inc* registers, and sums up their values.

```
operation Multi-use-fetch&increment                                  {Non linearizable}
      {at process P_i}
      Shared   T[1..∞]: array of n-process test-and-set ;
               my_inc[1..n]: array of integers, initialized to 0; swmr snapshot memory ;
      local    entry, S, M ;
      my_inc[i] := my_inc[i] + 1
      S := snapshot-scan{ my_inc registers}
      entry := Σ_{c_j ∈ S} c_j
      for j = entry downto 1 if test&set(T[j]) = 0 then return (j-1)
```

Figure 2: Non linearizable multi-use *fetch&increment*.

Clearly this construction is wait-free (though unbounded, since each operation depends on the number of operations that preceded it) because each operation cascades down from some location towards the beginning of the array until it wins a *test&set* operation.

The key observation of this subsection, which is the basis for subsequent subsections, is that every operation is guaranteed to find, and to set, an unset test-and-set register in the cascading process described in the last line of code in Figure 2. This fact is proved in the following lemma. However, the implementation is not linearizable since it is easy to construct scenarios in which an operation starts and finishes with value k while another operation that starts after that, returns with a value smaller than k.

Lemma 3.2.1 *Every fetch&increment operation (Figure 2) of any process P_i terminates after winning in a test&set operation.*

Proof: Assume to the contrary and consider an operation of process P_i that starts with $entry = h$ and losses in *test&set*$(T[1])$. Then P_i must have failed in all the test-and-set registers between $T[h]$ and $T[1]$. Since in each *fetch&increment* operation a process wins at most one test-and-set register, it must be that more than h *fetch&increment* operations have accessed the array in between $T[1]$ and $T[h]$.

Consider a run α of the implementation in which the lemma is violated. Define α_l to be the shortest prefix of α in which there is an $l > 0$ such that more than l operations accessed the test-and-set registers in the interval $T[1]$-$T[l]$. Note that by the end of α_l there might be several different values $l_1, l_2, \ldots l_j$ such that for each i more than l_i operations accessed the test-and-set registers in the interval $T[1]$-$T[l_i]$. Let l' be the largest value in $l_1, l_2, \ldots l_j$. That is, at least $l' + 1$ operations have accessed the test-and-set array between $T[1]$ and $T[l']$. Let L denote this set of $l' + 1$ operations.

Let *op* be the last operation in L to update its *my_inc* register in the first line of code in Figure 2. The value of *entry* of *op* must be at least $l'+1$. Thus, *op* must have accessed $T[l'+1]$ and failed in α_l. Thus at least $l'+2$ operations have accessed the test-and-set registers between $T[1]$ and $T[l'+1]$ (L and the winner of $T[l'+1]$), in contradiction to the definition of l'. ∎

Since the non linearizable fetch-and-increment of Figure 2 is introduced here mostly as a pedagogical step, we will not give a full proof of its correctness. However note that (1) Each test-and-set may be won by at most one process, thus a value is returned by at most one non linearizable *fetch&increment*, (2) consider a complete run, that is a run all of whose operations have terminated. Let k be the total number of *fetch&increment* operations that have started in the run. Then, no operation has started with *entry* $> k$. By Lemma 3.2.1, each of the operations returns with a correct value, and hence each value in $0 \ldots k-1$ is returned by the pigeon hole principle.

3.3 Proximate Snapshot

As indicated above, the problem of the construction in Figure 2 is that a *fetch&increment* operation *op* may be delayed for a long time after increasing its *my_inc* and before taking the snapshot. In that time interval many other operations may take place, each winning a test-and-set register and leaving an unset test-and-set down in the array. When *op* eventually resumes its operation, its snapshot would return an entry point which is much larger than the entry it would have gotten had it taken the snapshot soon after it had incremented the *my_inc*. This causes the construction to be both non linearizable and unbounded in time complexity.

To alleviate this problem we introduce a new type of *write and snapshot*, called proximate-snapshot.

Definition 5 *In the* proximate-snapshot *of process P, P performs an update and takes a* proximate-snapshot *that returns an atomic snapshot such that:*

1. *The update is an atomic write operation and the* proximate-snapshot *is also an atomic operation.*
2. *At most $n-1$ other updates may take place between P's update and its subsequent* proximate-snapshot*, i.e., the returned snapshot contains at most $n-1$ updates that followed P's update.*
3. *At the point at which P's* proximate-snapshot *returns no other* proximate-snapshot *operation has returned an earlier snapshot that includes P's last update. (In other words, all the snapshots returned by other* proximate-snapshot *and that are serialized before P's* proximate-snapshot *either do not include P's last update or return a snapshot that is not earlier than the snapshot returned by P's* proximate-snapshot*.)*

The basic idea behind the proximate-snapshot given in Figure 3 is that fast processes would help processes that are much slower to obtain an early snapshot. To this end each process keeps several copies of snapshots that it saw while doing *update and snapshot* operations. A process whose snapshot took place many steps (of other processes) after its corresponding update, would find among the copies of snapshots retained by the fast processes, an older snapshot which is closer to its update.

The shared data structures used by the implementation are:

my_inc[1..*n*] a swmr register, each counts the number of updates that the corresponding process has performed.

$my_minSS[1..n, 1..n]$, a 2-dimensional array of snapshot vectors. Each process maintains n snapshots, one for each other process.

In addition, the *my_inc* registers are augmented so that the snapshot algorithm of [AAD+93] or of [AR93] may be applied.

Since we construct the proximate-snapshot here only as a building block for the construction of fetch-and-increment, we will present here a simple version in which we assume that the update operation always increments a register called *my_inc*. Generalizing the proximate-snapshot presented here to a proximate-snapshot that makes arbitrary updates is a straightforward process which does not fit the current flow of the paper. Henceforth we assume that each process alternates between two operations, `update` and `Proximate-snapshot`, such that the `update` of process P_i always increments the value in register $my_inc[i]$. That is, no process ever performs two updates without taking a `Proximate-snapshot` between them.

The *proximate-snapshot* operation of process P_i has three steps. First the process takes a regular snapshot of the array *my_inc* (Line 1). Second, P_i checks the snapshot taken in the previous step to see if any process P_j has incremented its $my_inc[j]$ in between the last two snapshots taken by P_i. If P_i finds that $my_inc[j]$ has been updated, then it retains a copy of the new snapshot, in $my_minSS[i, j]$ in case P_j will ever need it (Line 2-6). Note that my_minSS is first calculated on a local copy, my_minSS', and then my_minSS is updated for all the processes in one atomic step (Line 5).

Third, P_i scans all the snapshots that other processes hold for it together with the new snapshot it took, to find out which of them is the earliest that contains the current value of $my_inc[i]$ (which is the most recent update of P_i) (Lines 7-13). To determine which of two snapshots is earlier the summation of their entries is compared. The snapshot with the smaller summation is clearly the earlier. To ensure the third property of proximate-snapshot we have to embed the scan in a repeat loop. Embedding the scan in a *repeat* loop ensures that the returned minimal snapshot was observed twice (and was the minimum in both) before being returned. This ensures that the minimal snapshot returned was not changed during the first time it was observed (collected). Hence, there is a point in which the process has finished reading the minimal snapshot and in this point this is still the minimal snapshot for that process (this property is used in the proof of Lemma 3.4.4).

Claim 3.3.1 *For every two processes P_i and P_j, P_i never updates the value in $my_minSS[i, j]$ twice, such that $my_minSS[i, j][j] = l$ in both updates.*

Proof: At line 4 a process P_i changes the value of another process P_j, only when it observed a change in the value of P_j in the snapshot S. Therefore for a given l it will update the value of $my_minSS[i, j]$ at most once. ∎

Claim 3.3.2 *A Process P_i in its l-th proximate-snapshot operation performs the* **repeat** *loop (Lines $7-13$) at most $n-1$ times.*

Proof: P_i remains in the loop only if it finds that another process P_k had updated its value in $my_minSS[k, i]$ and $my_minSS[k, i][i]$ is equal to l (Line 10). From Claim 3.3.1 it follows that at most $n-1$ such updates could occur (one for each another process). ∎

Lemma 3.3.3 *The complexity of the proximate-snapshot operation is $O(n^2)$.*

Proof: The cost of Line 1 is $O(n^2)$ [AAD+93, And93]. From claim 3.3.2 Line 9 can be repeated at most $n-1$ times, thus the **repeat** costs at most $O(n^2)$. Each of the other lines $(2-6)$ cost at most $O(n)$. ∎

```
Shared my_inc[1..n] : array of integers; swmr snapshot memory ;
       my_minSS[1..n, 1..n]: array of snapshots; swmr ; initialized to null ;
                                          {A 2-dimensional with n snapshots for each process.}
local   S, sum, prev − sum ;
operation Update(my_inc[i] register)                                          {at process P_i}
   my_inc[i] := my_inc[i] + 1

operation Proximate-snapshot(my_inc[1..n])                                    {at process P_i}
1: S := snapshot-scan{ my_inc[1..n]}                          {Take a regular snapshot}
                                   {Make a temporary copy of the two-dimensional array my_minSS}
2: my_minSS' := my_minSS
                                   {Check if processes have updated since the previous snapshot by P_i}
3: for j := 1 to n
4:    if S[j] > my_minSS'[i, j][j] then my_minSS'[i, j] := S
5: my_minSS := my_minSS'                                  {Atomically update my_minSS}
            {Next, read all snapshots kept for P_i to find the earliest that contains P_i's last update}
6: sum := Σ_{c_k ∈ S} c_k
7: repeat
8:    prev-sum := sum
9:      for j := 1 to n
10:   if S[i]=my_minSS[j, i][i] and sum > Σ_{c_k ∈ my_minSS[j,i]} c_k then
11:       S := my_minSS[j, i]
12:       sum := Σ_{c_k ∈ S} c_k
13: until (sum = prev-sum)
14: return (S)
```

Figure 3: The proximate-snapshot construction

Definition 6 *PSS_i^j is the snapshot returned by the j-th* proximate-snapshot *operation of process i.*

Definition 7 *w_i^j is the j-th write operation by process i to my_inc[i], that is: my_inc[i] := j.*

Definition 8 *$atomic_i^j$: Consider the sequence of all the write operations to the shared variables my_inc in Figure 3. Since the writes are atomic, this is a total order. We then define $atomic_i^j$ to be the index of w_i^j in this sequence.*

Note that $atomic_i^j$ is the summation of the *my_inc* registers values immediately after the j-th write by process i.

Lemma 3.3.4 *For every i and j,*

$$atomic_i^j \leq \left(\sum_{c \in PSS_i^j} c \right) < atomic_i^j + n$$

Proof: PSS_i^j is a regular atomic snapshot that was taken after w_i^j. Hence all the values in this snapshot are at least as large as they were at the time of the w_i^j operation. Therefore their sum is at least as large as $atomic_i^j$, and the left hand part of the inequality holds.

To prove the right hand side of the inequality we claim that there are at most $n-1$ w_k^l operations, $k = 1, \ldots, n$, $l = 1, \ldots$ after w_i^j and before PSS_i^j was first taken as an atomic snapshot by some process (not necessarily process i). Assume by way of contradiction that there are more than $n-1$ such operations. Thus there must be at least one process index, t such that there were two w_t operations (w_t^l, and w_t^{l+1}) after w_i^j and before PSS_i^j was taken. Thus, after w_t^l and before w_t^{l+1} process P_t must have performed a complete *proximate-snapshot* operation, denoted pss_t^l. During the pss_t^l operation process P_t computes a snapshot in which w_t^l was observed and hence also w_i^j was observed. Therefore after executing Line 5 of pss_t^l *my_minSS*$[t, i]$ must contain a snapshot that includes w_i^j. Since by the assumption PSS_i^j is after w_t^{l+1} process P_i starts executing procedure Proximate-snapshot after process P_t has finished the pss_t^l operation. Therefore, when process P_i reaches Line 10 the condition in that line must be satisfied for some process index, and S must contain a snapshot whose value for process P_t is at most l. This contradicts the assumption that PSS_i^j has a value larger than l for t. ∎

Definition 9 *$ll8_i^j$ is the last execution of Line 8 in the j-th execution of proximate-snapshot operation by process i.*

Lemma 3.3.5 *The implementation in Figure 3 satisfies property 3 of Definition 5.*

Proof: Serialize each operation pss_i^j at $ll8_i^j$. Let pss_r^l and pss_q^k be two *proximate-snapshot* operations such that $w_q^k \in PSS_r^l$ and PSS_r^l is earlier than PSS_q^k. To complete the proof of the lemma we claim that $ll8_q^k$ was performed before $ll8_r^l$. P_r computes PSS_r^l according to a snapshot of P_g (g might be equal to r), which had updated its *my_minSS* (Line 5 in Figure 3) to contain a snapshot that corresponds to PSS_r^l. Therefor P_r performs $ll8_r^l$ after P_g's execution of Line 5. On the other hand, PSS_r^l could be also a valid value for PSS_q^k, since $w_q^k \in PSS_r^l$. But PSS_q^k is later than PSS_r^l, so it follows that P_q had performed $ll8_q^k$ before the execution of Line 5 by P_g. ∎

From the above lemmas we get:

Theorem 1 *The code in Figure 3 correctly implements a wait free proximate-snapshot operation and each operation in it terminates in at most n^2 accesses to the shared memory.*

3.4 Multi-use fetch-and-increment

Here we combine the proximate-snapshot of the previous subsection with the unbounded array of test-and-set registers as in Subsection 3.2 to get a wait-free linearizable multi-use-fetch-and-increment. We give in detail an unbounded space implementation, a method to transform it into a bounded number of registers implementation is outlined in Section 5.

The idea is exactly as in Subsection 3.2. Since here we replace the regular snapshot by an proximate-snapshot the implementation becomes bounded wait-free and linearizable as proved next.

operation Multi-use-*fetch&increment*
{at process P_i}
Shared $T[1..\infty]$: array of n-process test-and-set ;
$my_inc[1..n]$: and $my_minSS[1..n, 1..n]$; for the proximate-snapshot operation;
local entry, S ;
$my_inc[i] := my_inc[i] + 1$ {This is the update from Figure 3}
$S :=$ proximate-snapshot{ $my_inc[1..n]$ }
$entry := \sum_{c_j \in S} c_j$
for $j = entry$ **downto** 0 if *test&set*(T[j]) = 0 **then return** (j-1) {See Lemma 3.4.1}

Figure 4: Multi-use *fetch&increment*.

The implementation is described in Figure 4. The shared data structures include those from Section 3.3 together with:

$T[1..\infty]$ An infinite array of test-and-set registers.

3.4.1 Correctness

Definition 10 *$start_i^j$: is the value of the local variable entry of P_i in its j-th* multi-use-*fetch&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 11 *inc_i^j is the value returned in P_i's j-th* multi-use-*fetch&increment operation. That is, the index of the* test-and-set *register which P_i wins when $my_inc[i] = j$ minus one.*

Definition 12 *pss_i^j (fai_i^j) is the j-th proximate-snapshot (fetch&increment) operation of process P_i.*

Lemma 3.4.1 *Every* multi-use-*fetch&increment operation of any process P_i terminates with $entry > 0$.*

Proof: The algorithm in figure 4 differs from the non-linearizable solution in Figure 2 only in the calculation of *entry*. Since the proximate-snapshot operation returns a "legal" snapshot vector, the proof of Lemma 3.2.1 carries over verbatim. ∎

Lemma 3.4.2

$$atomic_i^j \leq start_i^j < atomic_i^j + n$$

Proof: Follows from Lemma 3.3.4 since $start_i^j = \sum_{c \in PSS_i^j} c$. ∎

Lemma 3.4.3 *If in a run of the system there is a process P_i and a j, such that:*

$$atomic_i^j > t \geq inc_i^j$$

then in this run there must be a process P_l, $l \neq i$, and an m such that:

$$atomic_l^m \leq t < inc_l^m \leq start_l^m$$

Proof: By way of contradiction, assume such P_i, j, t and that for any operation by process P_r and any k such that $atomic_r^k \leq t$, $inc_r^k \leq t$. Then at least $t+1$ (these operations and the j-th operation of P_i) access the array $T[1..\infty]$ between 1 and t. By Lemma 3.4.1 all the processes terminate. Thus $t+1$ processes terminate in locations 1 to t of the array. Therefore two processes must have won the same test-and-set register, contradiction. ∎

Lemma 3.4.4 *Consider a run of the system in which there are r, l, q and k such that:*

$$atomic_q^k \leq start_r^l < start_q^k$$

then, process P_q had performed pss_q^k *before process P_r performed* pss_r^l.

Proof: From the third property of proximate-snapshot (see Definition 5), it follows that $start_r^l$ was taken after $start_q^k$ was taken, hence the lemma. ∎

Lemma 3.4.5 *In any run of the implementation, if a* fai_i^x *starts after an operation* fai_j^y *finishes then, $inc_i^x > inc_j^y$.*

Proof: Assume to the contrary that $inc_i^x < inc_j^y$. Since fai_i^x starts after fai_j^y ends, and since $start_j^y \geq inc_j^y$ it follows that $atomic_i^x > start_j^y > inc_i^x$. From Lemma 3.4.3 it follows that there is a process P_{k_1} and a b_1 such that, $atomic_{k_1}^{b_1} \leq start_j^y < start_{k_1}^{b_1}$, by taking t in Lemma 3.4.3 to be $start_j^y$. If $atomic_i^x > start_{k_1}^{b_1}$ then this argument can be repeated to show that there is a P_{k_2} and a b_2 such that $atomic_{k_2}^{b_2} \leq start_{k_1}^{b_1} < start_{k_2}^{b_2}$. This argument is repeated until we arrive at a P_{k_l} such that $atomic_i^x \leq start_{k_l}^{b_l}$, and $atomic_{k_l}^{b_l} \leq start_{k_{l-1}}^{b_{l-1}} < start_{k_l}^{b_l}$. By Lemma 3.4.4, pss_j^y was performed after $pss_{k_1}^{b1}$. Apply this lemma repeatedly to get that $pss_{k_{z-1}}^{b_{z-1}}$ was performed after $pss_{k_z}^{b_z}$, for $z = 2, \ldots, l$. P_{k_l} started its $pss_{k_l}^{b_l}$ operation after w_i^x, since $start_{k_l}^{b_l} > atomic_i^x$. Hence P_j performed its pss_j^y operation after w_i^x in contradiction to the assumption that P_i starts after P_j finishes. ∎

Lemma 3.4.6 *For every $x \geq 1$ there are at most n* fai_t^l *operations with $atomic_t^l > x$ such that $inc_t^l \leq x$ (each might access the array $T[1..\infty]$ between 1 and x)*

Proof: Since $start_t^l = PPS_t^l$, it follows from Lemma 3.4.2 that $atomic_t^l \leq start_t^l < atomic_t^l + n$. Therefore from the x fai_k^m operations with $atomic_k^m \leq x$, at most n operations might have computed $start_k^m > x$, and the others, at least $x - n$ operations, compute $start_k^m \leq x$. Since by Lemma 3.4.1 at most x *fetch&increment* operations access the array $T[1..\infty]$ between 1 and x, and at least $x - n$ of them have $atomic_k^m \leq x$, it follows that at most n fai_t^l operations with $atomic_k^m > x$ access the array between 1 and x. ∎

Lemma 3.4.7 *In a single fetch&increment operation, a process losses in less than* $3n$ *T&S registers in the array* $T[1..\infty]$.

Proof: Assume to the contrary that there are t and l such that during the fai_t^l operation, P_t losses in $3n$ $T\&S$ registers. Since P_t starts in $T[start_t^l]$ it follows from Lemmas 3.4.2 and 3.4.6 that it must have lost to two *fetch&increment* operations of the same process P_k, fai_k^{m1} and fai_k^{m2}, such that $atomic_k^{m1} < atomic_k^{m2} < atomic_t^l$. The operation fai_k^{m1} was completed before the beginning of the execution of fai_t^l. Since fai_t^l lost to fai_k^{m1} it would terminate with a inc_t^l which is smaller than inc_k^{m1} in contradiction to Lemma 3.4.5. ∎

Definition 13 *x is a hole in a run α if x was not returned by any fetch&increment operation in α while there is a value y, $y > x$ that is returned by an operation in α.*

Claim 3.4.8 *Let β be a complete finite run of z fetch&increment operations, then there are no holes in β, that is, all the integer values from 0 to $z-1$ are returned in β.*

Proof: For every fai_i^t in β, $start_i^t \leq z$ since $start_i^t$ is a "real" snapshot of the *my_inc* registers. The usage of the test-and-set bit guarantees that such operation terminates with a unique value in entry. The claim now follows from Lemma 3.4.1 and the "pigeon hole" principle. ∎

Claim 3.4.9 *Consider an infinite run α in which the value m is returned by some fetch&increment. Then in any prefix α' of α in which:*

1. *m has been returned, and*
2. *There is a set $S = \{s_1, s_2, \ldots s_k\}$ of holes $s_i < m$ $i = 1, 2, \ldots k$,*

there is a set $O = \{o_1, o_2, \ldots o_k\}$ of fetch&increment operations s.t.:

1. *No operation $o_i \in O$ has finished in α', and*
2. *there is a suffix β' such that $\alpha'\beta'$ is a legal finite run in which every $o_i \in O$ terminates with a value $s_i \in S$.*

Proof: Consider an α' as in the statement of the claim. We construct for this α' a β' that satisfies the claim by simply scheduling all the operations that has not yet terminated to take enough steps (in any order) until they all terminate. By Claim 3.4.8 this suffix satisfies the claim. ∎

Theorem 2 *The code in Figure 4 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.*

Proof: From Lemmas 3.4.1, 3.4.7 and 3.3.3 it follows that the implementation is wait-free, with a complexity of $O(n^2)$ primitive operations per *fetch&increment* operation. To prove the correctness of the implementation we define a total order α on the *fetch&increment* operations and prove that this total order is an extension of the natural partial order β. The partial order β is induced on the operations by the relation $<_\beta$ which is defined as follows: $fai_q^j <_\beta fai_p^l$ iff fai_q^j finishes before fai_p^l starts. The total order is defined by the relation $<_\alpha$ which is: $fai_q^j <_\alpha fai_p^l$ iff $inc_q^j < inc_p^l$. By Lemma 3.4.5 the total order α agrees with the partial order β. Together with Claim 3.4.9 this completes the proof. ∎

4 Atomic-write-and-snapshot and Fetch-and-add

While proximate-snapshot is sufficient for the construction of fetch-and-increment it does not suffice to implement a fetch-and-add object. The difference between the two being that in fetch-and-increment a process returns from an operation with the number of operations that took place (are linearized) before it whereas, a process returning from a *fetch&add* operation essentially identifies the exact set of operations that is linearized before it. This additional strength of fetch-and-add requires a more sophisticated implementation. Here we show that it is enough to strengthen the snapshot primitive used within the implementation of fetch-and-increment. That is, while in the non linearizable fetch-and-increment a regular snapshot was enough, and in the linearizable fetch-and-increment proximate-snapshot was sufficient, here for the fetch-and-add construction we need the atomic-write-and-snapshot. The atomic-write-and-snapshot is based on the immediate-snapshot defined by Borowsky and Gafni [BG93b] which is a restricted case of proximate-snapshot.

Let us next define the atomic-write-and-snapshot, then show how it is used to generate fetch-and-add in Subsection 4.1. Then in Subsection 4.2 we present an implementation of `single-use-atomic-write-and-snapshot` from test-and-set registers that gives us also `single-use-fetch-and-add`, and in Subsection 4.3 we implement `multi-use-atomic-write-and-snapshot` from test-and-set registers that produces as a by product `multi-use-fetch-and-add`.

The atomic-write-and-snapshot is a restriction of the immediate-snapshot (or the participating set problem) of Borowsky and Gafni. In the participating set algorithm several processes that have accessed the object concurrently may return (observe) the same set of participating processes. Here we augment their algorithm with test-and-set registers to obtain the new object, called atomic-write-and-snapshot:

Definition 14 *The atomic-write&snapshot operation by process p atomically writes a value in a register in a shared array of* swmr *registers, and returns an atomic snapshot scan of the array. That is, the i^{th} operation of process p writes i into the shared memory and returns a vector R_p^i of indices such that:*

1. Self-Containment: $R_p^i[p] = i$.
2. Atomic Snapshot: *For all p, q, i, j, either $R_p^i \leq R_q^j$ (element by element) or $R_p^i \geq R_q^j$.*
3. Atomic Write&Snapshot: *For all p, q, i, j, if $R_p^i[q] = j$ then $R_q^j[p] < i$.*

Note that it follows that any atomic-write&snapshot operation that is later in the serial order "observes" all the atomic-write&snapshot operations that are ordered before it. That is, the snapshot returned is an accurate picture of the array instantly after the write operation. This is the best one can hope for in the write and snapshot *problem from Subsection 3.2, i.e., this is an* atomic write&snapshot. *No other update or snapshot may take place in between the update and the snapshot of one processor.*

4.1 Implementing Fetch-and-add from Atomic-write-and-snapshots

The definition of the atomic-write-and-snapshot object enables each operation to return an unordered set of operations linearized before it, which is essentially what is necessary for fetch-and-f, Figure 5. In the implementation of fetch-and-add (replacing f in the figure by summation) process P_i keeps

a local variable $part_i$ with the sum of all of its inputs. To perform *fetch&add* P_i first adds $input_i$ to its private sum, $part_i$, and then calls atomic-write-and-snapshot with $part_i$ as an input. Then, summing up all the parts in the returned snapshot gives P_i the total value of all the processes' at the time it has posted its new part in the atomic-write-and-snapshot.

operation *fetch&f(input$_i$)* {Multi-use; f is any commutative function}
  Shared the necessary objects for the multi-use-atomic-write-and-snapshot operation ;
    $part[1..n]$: array of swmr registers; initially 0
  local M: set of at most n data values ;
    tmp, *prev*: temporary local result ;
    $prev := part[i]$
    $tmp := f(part[i], input_i)$ {in *fetch&add* $:= part[i] + input_i$ }
    $M :=$ atomic-write-and-snapshot$(part[i] := tmp)$;
    M' := M with the i-th entry replaced by $prev$;
    return $f(M')$ {The returned value excludes the application of $f(., input_i)$
      e.g., in *fetch&add*, $f(M') = \sum_{part \in M'} part$ }

Figure 5: The implementation of *fetch&f(input$_i$) from* atomic-write-and-snapshot

4.2 Single-use-Atomic-write-and-snapshot

The implementation of single-use-Atomic-write-and-snapshot is based on the algorithm of Borowsky and Gafni for the one-shot immediate-snapshot. The one-shot immediate-snapshot is also an operation in which a process writes a value to the immediate-snapshot-shared-memory and returns a snapshot. However, in the immediate-snapshot the operation is not atomic. Instead, in each execution of the immediate-snapshot, the operations are partitioned into sets of operations with a total order between the sets. Operations that are not concurrent belong to different sets, and the total order between the sets extends the natural $after$ relation [Lam78]. The specification of immediate-snapshot requires that the snapshot that operation o returns contains the values written by immediate-snapshot operations that are together with o in the same set and immediate-snapshot operations that are in preceding sets. Single-use-Atomic-write-and-snapshot is however a restriction of immediate-snapshot in the sense that here each operation should be in a set of its own. That is, a write and a snapshot operations that are executed as one atomic operation. Therefore, the code of [BG93b] is augmented with *test&set* operations. At the point where an operation should return the immediate snapshot in [BG93b] it first *test&set* that particular snapshot. If the *test&set* was successful, then this snapshot is returned only by that operation. Otherwise, the process performing the atomic-write-and-snapshot tries to capture a smaller snapshot that is contained in the snapshot it failed to capture. This is done by continuing the iterations of the immediate-snapshot algorithm, until finally winning the test-and-set in one of the snapshots.

Definition 15 *Let* wss$_i$ *be the single* write-and-snapshot *operation of process i; denote by* snap$_i$ *the value returned by* wss$_i$.

Definition 16 *S_p is the set that is assigned into S in the last execution of Line* 4 *in the code in Figure 6, by P_p in operation* wss$_p$.

```
operation single-use-atomic-write-and-snapshot(input_i)
    Shared T[1..n]: array of n-process test-and-set ;
           level[1..n]: array of swmr registers; initialized to n + 1;
           value[1..n]: array of swmr registers;
    local S: set of at most n integer numbers ;
    1:     value[i] := input_i;
    2:     while TRUE do
    3:         level[i] := level[i] - 1;
    4:         S := {j | level[j] ≤ level[i]};
    5:         if |S| ≥ level[i] then
    6:             t := test&set(T[level[i]])
    7:             if t = 0 then
    8:                 return {value[j] | j ∈ S}
           od
```

Figure 6: Single-use-atomic-write-and-snapshot

Lemma 4.2.1 *For every $1 \leq x \leq n$ there are at most x write&snapshot operations that might access the array $T[1..n]$ in the interval 1 to x.*

Proof: By way of contradiction. Consider a run α of the implementation in which the lemma is violated. Define α_l to be the shortest prefix of α in which there is an $0 < l < n$ such that more than l *write&snapshot* operations accessed the test-and-set registers in the interval $T[1]$-$T[l]$ in α. Note that by the end of α_l there might be several different values $l_1, l_2, \ldots l_j$ such that for each i $1 \leq i \leq j$, more than l_i operations accessed the test-and-set registers in the interval $T[1]$-$T[l_i]$. Let l' be the largest value in $l_1, l_2, \ldots l_j$. That is, at least $l'+1$ operations have accessed the test-and-set array between $T[1]$ and $T[l']$ in α_l. Let L denote this set of $l'+1$ operations. Let wss_r be the last operation in L to write $level[r] := l'+1$ in Line 3. Then P_r must have seen $|S| \geq l'+1$ in Line 5 and has performed $test\&set(T[l'+1])$. Since by assumption P_r accessed $T[l']$, it must have lost in $test\&set(T[l'+1])$. Therefore at least $l'+2$ *write&snapshot* operations accessed the array $T[1 \ldots n]$ between 1 and $l'+1$, the $l'+1$ operations in L, and the operation that won in $T[l'+1]$ which is necessarily not a member of L. This is a contradiction to the assumption that l' is the largest index that was violated in α_l. ∎

Definition 17 *For a run α of the algorithm we define $participate_l = \{r | level[r] \leq l$ at some point in $\alpha\}$, for $1 \leq l \leq n$.*

Corollary 4.2.2 *For every l_1 and l_2, $l_1 < l_2$ if $q \in participate_{l_1}$ then $q \in participate_{l_2}$ and therefore $participate_{l_1} \subset participate_{l_2}$. Therefore, for every $l_1 \neq l_2$ either $participate_{l_1} \subset participate_{l_2}$ or, $participate_{l_1} \supset participate_{l_2}$.*

Claim 4.2.3 *Consider a run of the algorithm in which process P_p wins the test&set on $(T[l])$, then the set S_p equals $participate_l$, and for every other process P_q either $S_p \subset S_q$ or $S_p \supset S_q$.*

Proof: From Lemma 4.2.1 it follows that $|participate_l| \leq l$ and since in Line 5 $|S| \geq l$ it must be that S_p is identical to $participate_l$. Since for every other process q S_q is identical to $participate_{l'}$, $l' \neq l$ it follows from Corollary 4.2.2 that either $S_p \subset S_q$ or $S_p \supset S_q$. ∎

Lemma 4.2.4 *Let* snap_p *and* snap_q *be the sets that were returned by two write&snapshot operations. Then, either* $\text{snap}_p \subset \text{snap}_q$ *or* $\text{snap}_p \supset \text{snap}_q$.

Proof: Let x be the index of the $T\&S$ register that P_p won, and y be the $T\&S$ register that P_q won. Assume, w.l.o.g. that $x > y$. From Claim 4.2.3 it follows that $S_p = participate_x$ and $S_q = participate_y$ and that $S_q \subset S_p$. Therefore, $snap_q \subset snap_p$. ∎

Lemma 4.2.5 *In any run of the implementation, if* P_p *starts its write&snapshot operation after* P_q *finishes its write&snapshot operation then,* $\text{snap}_p \supset \text{snap}_q$.

Proof: Let y be the index of the $T\&S$ register that was won by P_q and let S_y be the set S that P_q saw in its last **while** iteration in Line 4. The index p is not in S_y, since P_p have not started its operation when S_y was taken. It follows that $snap_q \not\supset snap_p$ and therefore from Lemma 4.2.4 it must be that $snap_p \supset snap_q$. ∎

Theorem 3 *The code in Figure 6 correctly implements a single-use* write-and-snapshot *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.*

Proof: From Lemmas 4.2.1 it follows that the run of the implementation terminates after at most n iterations of the **while** loop. The cost of the execution of Line 4 is $O(n)$, all the other lines have constant cost, and therefore the complexity of the implementation is $O(n^2)$ with at most n operations on the test-and-set array, and it is obviously wait-free. To prove the correctness of the implementation we define a total order α on the *write&snapshot* operations and prove that this total order is an extension of the natural partial order β. The partial order β is induced on the operations by the relation $<_\beta$ which is defined as follows: $wss_q <_\beta wss_p$ iff wss_q finishes before wss_p starts. The total order is defined by the relation $<_\alpha$ which is: $wss_q <_\alpha wss_p$ iff $wss_q \subset wss_p$. By Lemma 4.2.5 the total order α agrees with the partial order β. ∎

4.3 Multi-use-Write-and-snapshot

Overview: The single-use solution of the previous section cannot be extended as is to the multi-use because it requires knowing ahead of time, the total number of operations that would ever be applied (and the complexity of the multi-use operation would linearly depend on that number). A multi-use immediate-snapshot algorithm would probably solve the problem, however no such algorithm existed before this work and the algorithm here provides the only solution that we know of to that problem. The implementation of Multi-use-write-and-snapshot presented here is a non-trivial extension of the previous section combined with the techniques of the proximate snapshot (Section 3.3).

Following [BG93b] the single-use solution was constructed around an algorithm to snapshot the names of the processes whose values should be returned ("participating set" in the jargon of [BG93b]). Then, these names were used as indices to the shared-memory from which the values of the immediate snapshot were taken (i.e., indirect addressing). Here we use a similar idea. Each process owns an unbounded[1] linear array of registers each of which can hold one value (the array is called *values* in the code) and a pointer to the array (called *my_inc* in the code). Instead of

[1] In the next section we show a technique by which this array can be bounded.

overwriting the values in a single register, each process writes a new value by appending it in the next vacant place in its array and then incrementing its pointer by one (Lines 1 and 2 in the code in Figure 7). After writing a value in this way each process performs a write-and-snapshot on the array of pointers (Lines 3 to the end in Figure 7). The multi-use-write-and-snapshot then terminates by returning the values of each process by indirect addressing with the snapshoted pointer values. Henceforth, the rest of the overview considers the multi-use-write-and-snapshot of the pointer values (that are incremented by one in each operation).

Intuitively the algorithm presented here works as follows: The algorithm groups the *write&snapshot*(v) operations into disjoint sets of operations. The operations in each set are concurrent. Within each set the single-use-write-and-snapshot is used to order the operations in a legal order. While each operation locates the set with which it has to do the single-use-write-and-snapshot it also informs later sets of its new value and ensures that no earlier set observes the new value.

Let us first provide a high level description of the write-and-snapshot algorithm on the pointers values (each value is monotonically incremented by its process, like a counter). Imagine an unbounded linear array of cells. Associate with each cell a single-use-write-and-snapshot object, and n read/write words, one per process, in which each may register a pointer value. In each cell each process is registered with some pointer value. In each multi-use-write-and-snapshot operation a process increments its pointer value by one and finds one cell. The cell a process finds in an operation is higher in the array than the one it had found in the previous operation. However, in finding a cell, a process ensures two things: (1) that it is registered with the previous pointer value in all the cells from the one it had found in the previous operation (or the zero cell, if this is the first time) to the current one (exclusive) and (2) that in all higher cells it would be automatically registered with its new pointer value (until the next operation). After entering a cell a process performs a single-use-write-and-snapshot with all the other operations that have entered this cell. Then, from the set obtained by the single-use plus the registration information associated with the cell it computes a set of pointer values to return as a snapshot. Thus the algorithm ensures that if in some operation process p enters a cell that is lower than a cell in which some operation of process q enters, then p is already registered in the cell that is captured by q, with p's new pointer value.

More specifically, each process p starts its operation by writing its input and incrementing the pointer ($my_inc[p]$) by one (Lines 1 and 2 in Figure 7). Since all the updates of the pointers are atomic write operations they are totally ordered and may be sequentially enumerated by an outside observer. Moreover, each $my_inc[]$ pointer can be viewed as a counter since it is incremented by one in each operation, therefore, the enumeration of the atomic increment points (of the $my_inc[]$ pointers) associates with each updating point the summation of the pointers values at this point. We call this summation of a point, the *floor* of the point, and with each such point we associate one cell (with a single-use-write-and-snapshot as described above). That is, the floor of one point is larger by one than its previous. In other words, had we taken a snapshot of the $my_inc[]$ pointers values instantly after an increment then the summation of its entries is equal to the floor of this update. In the implementation we maintain an unbounded array of snapshots that reserves one slot for each such point (called SS[$\cdot$] in the code), this array parallels the array of cells, i.e., it associates one snapshot with each cell (these snapshots should not be confused with the ones returned by processes that exit from a cell).

Clearly, many of these snapshots may not be observed and thus their corresponding slots in the array remain vacant throughout the algorithm execution. However, the first snapshot each process obtains in its operation (in Line 3) is a real snapshot produced by a procedure as in [AAD+93, And93] that is called from within proximate-snapshot. Hence, this snapshot is one of those enumerated above

and the process writes it into $SS[floor]$. Since these are real instantaneous atomic snapshots, if two processes write a snapshot to the same location in the array, the two snapshots are identical.

After obtaining an initial snapshot with total value f, and writing it into $SS[f]$ process p starts the search for the floor (cell) in which to perform single-use- write-and-snapshot. The snapshots that are recorded in the array SS are used by processes to obtain values for processes that they do not observe in the single-use-write-and-snapshot. Clearly, any atomic snapshot that would be recorded in higher floors (higher than f) would contain the new or later $my_inc[p]$ pointer values of p and thus would be observed by processes that exit the algorithm from these floors. Also, if $my_inc[p]-1$, the pointer value in the previous operation of p appears in some snapshot in $SS[l]$, in floor l lower than f, then all the operations that exit from floor l and below would observe one of the previous values of $my_inc[p]$ but not the new. All that remains for p to do is to record its new $my_inc[p]$ pointer value in all the floors below f and in which it does not find itself recorded in a snapshot with the previous pointer value. Since we used the proximate-snapshot to compute f it is guaranteed that p finds a floor in which it is recorded with its previous pointer value after checking no more than $2n$ floors from f down (see Lemma 4.4.3). In each floor that p is descending through, it performs two tasks: first it records its new $my_inc[p]$ pointer value, and second, if it observes itself in the atomic snapshot that is associated with that floor with its previous pointer value, $my_inc[p]-1$, then it participates in a single-use-write-and-snapshot that is associated with that floor. The key to our algorithm is that both tasks can be achieved by a single-use-write-and-snapshot. Thus, when arriving at a new floor the process performs single-use-write-and-snapshot to first record its new value in this floor, and this operation is used in case it turns out that it has to exit from that floor.

Note that the snapshots that are returned in the end of each operation are not necessarily snapshots that were returned by a proximate-snapshot or recorded in SS. The algorithm serializes the atomic point of each multi-use-*write&snapshot* operation at the point in which it has executed the last single-use-*write&snapshot*. More specifically, a process p stops descending through the cells if it observes its previous $my_inc[p]$ in a cell. A process observes itself with value $my_inc[p]-1$ in a cell if one of the processes that is returned in its single-use-*write&snapshot* operation in that floor has observed the snapshot associated with this floor and p appears in this snapshot with $my_inc[p]-1$. (note that the observed process could be p itself, Line 18 in Figure 7). At this point p has to compose the snapshot it returns from the *write&snapshot* operation. For each process q, p should choose one of two values to be included in the snapshot it returns, either the value with which q appears in the snapshot associated with floor f, or the value with which q is registered in floor f (in $inc[f,q]$). For each process that p observes in the single-use-*write&snapshot* it chooses the value with which that process is registered. For all other processes it chooses the value with which they appear in the associated snapshot (in $SS[f]$).

The formal code of the algorithm is given twice in Figures 7 and in Figure 8. The proofs use the code given in Figure 7. The reason for presenting Figure 8 is that the code there is modular and uses the single-use-write-and-snapshot as a building block, thus being shorter. However, for the proofs we found it more appropriate to use the full expanded version without the modular calls.

4.4 Correctness

Definition 18 *In this section we make the following definitions:*

wss_p^i *denotes the i-th write&snapshot operation of process p.*

```
operation atomic-write-and-snapshot(input)                                  {at process P_p}
    Shared   SS[0..∞]: array of snapshot vectors initialized to ⊥, SS[0] is initialized to [0, 0, ...] ;
             T[1..∞, 1..n]: array of n-process test-and-set ;
             my_inc[1..n]: array of integers; in swmr snapshot memory initialized to 0;
             value[1..n, 1..∞]: array of swmr registers ;
             level[1..n, 1..∞]: array of swmr registers initialized to n + 1;
             flag[1..n, 1..∞]: array of swmr registers ;
             inc[1..n, 1..∞]: array of swmr registers initialized to 0;
    local    floor, exit_floor, PSS, S, snap;
 1: value[p, my_inc[p] + 1] := input
 2: my_inc[p] := my_inc[p] + 1
 3: PSS := proximate-snapshot(my_inc[1..n])
 4: floor := Σ_{j∈PSS} j
 5: SS[floor] := PSS
 6: while TRUE do
 7:       floor := floor − 1
 8:       exit_floor := FALSE
 9:       flag[p, floor] := (SS[floor] ≠ ⊥)
10:       inc[p, floor] := my_inc[p]
11:       while not exit_floor do
12:             level[p, floor] := level[p, floor] − 1
13:             S := {j | level[j, floor] ≤ level[p, floor]};
14:             if |S| ≥ level[p, floor] then
15:               b := test&set(T[floor, level[p, floor]])
16:               if b = 0 then
17:                 exit_floor := TRUE
18:                 if (∃j ∈ S s.t., flag[j, floor] = TRUE) and (SS[floor, p] < my_inc[p]) then
19:                   for h := 1 to n do
20:                       if h ∈ S then R[h] := inc[h, floor]
21:                         else R[h] := SS[floor, h]
22:                       snap[h] := value[h, R[h]]
23:                   od
24:                   return (snap[1..n])
25:                 fi
26:               fi
27:             fi
28:       od
29: od
```

Figure 7: The code of multi-use-Write-and-snapshot

```
operation atomic-write-and-snapshot(input)                                  {at process P_p}
    Shared  SS[0..∞]: array of snapshot vectors initialized to ⊥, SS[0] is initialized to [0, 0, ...] ;
            my_inc[1..n]: array of integers; in swmr snapshot memory initialized to 0;
            value[1..n, 1..∞]: array of swmr registers ;
    local   floor, flag, PSS, S, R, snap ;
    value[p, my_inc[p] + 1] := input
    my_inc[p] := my_inc[p] + 1
    PSS := proximate-snapshot(my_inc[1..n])
    floor := Σ_{j∈PSS} j
    SS[floor] := PSS
    while TRUE do
        floor := floor − 1
        flag := (SS[floor] ≠ ⊥)
        S := single-use-atomic-write-and-snapshot_floor((p, flag, my_inc[p]))
        if ∃(j, f_j, inc_j) ∈ S, f_j == TRUE and SS[floor, p] < my_inc[p] then do
          forj := 1 to n do
            if ∃(j, f_j, inc_j) ∈ S then
              then R[j] := inc_j
              else R[j] := SS[floor, j]
            snap[j] := value[j, R[j]]
          return snap
        od
    od
```

Figure 8: An alternative modular code for the mulit-use-Write-and-snapshot

$input_p^i$ is the input to wss_p^i.

$snap_p^i$ is the snapshot vector returned by wss_p^i.

pss_p^i is the snapshot returned by the proximate-snapshot in Line 3 *in the execution of* wss_p^i.

$entry_p^i$ is the entry floor of wss_p^i*, i.e., the summation of* pss_p^i *(Line* 4*).*

$exit_p^i$ is the last value of floor in the execution of wss_p^i*, i.e., the floor where* wss_p^i *exits with a value (Line* $19-24$*).*

$S_p^{f,i}$ is the set that is assigned into S in the last execution of Line 13 *while the value of $floor = f$ for process p in operation* wss_p^i*. $S_p^{f,i}$ corresponds to the set returned by the single-use-write&snapshot in the floor at which* wss_p^i *finishes the operation.*

$capture_{(f,t)}$, $f \geq 1$, $1 \leq t \leq n$ is a pair process name, p and operation index, i such that the last $T\&S$ cell won by process p in its ith operation (wss_p^i*) is $T[f,t]$. $capture_{(f,t)} = \perp$ if there is no write&snapshot operation such that the last $T\&S$ cell won by the process performing the operation is $T[f,t]$. A pair $capture_{(f,t)} \neq \perp$ uniquely defines an operation.*

$R_{(f,t)}$ is the vector R (of my_inc indices) of the $capture_{(f,t)}$ operation in line 24*. This vector is the output of write&snapshot(my_inc), which is then used as pointers to retrieve the values of* snap_p^i *which is then the output of write&snapshot($input_p^i$).*

Lemmas 4.4.1, 4.4.2, 4.4.3, and 4.4.4 prove the complexity of the algorithm while the rest of the proof shows that it is a correct implementation of *write&snapshot*.

Lemma 4.4.1 *For every $d > n+1$, when a process writes $SS[d] = PSS$ in Line* 5*, there must be at least one entry in the $SS[]$ array in the interval between $d-(n+1)$ and $d-1$ that already contains a snap-shot in it. Therefore, at the time that $SS[d]$ is updated, the largest consecutive sequence of empty ($= \perp$) entries in the array $SS[]$, between* 1 *and d is less than $n+1$.*

Proof: For every process p in its wss_p^i operation the incrementing of the value of the $my_inc[p]$ register in Line 2 is identical to the **Update** operation in the *proximate-snapshot* operation in Section 3.3, the definition and characteristics of $atomic_p^i$ stand here too. Let T be the set of all the wss_p^i *write&snapshot* operations such that $d-(n+1) \leq atomic_p^i \leq d-1$. Since $|T| > n$ there must be two operations of the same process $wss_q^{l_1}$ and $wss_q^{l_2}$ in T. The first operation $wss_q^{l_1}$ had terminated before $wss_q^{l_2}$ has started, so it had assigned $floor = f$ in Line 4 such that $f < atomic_q^{l_2}$ since its PSS is a "real" snap-shot that was taken before the beginning of $wss_q^{l_2}$. From the same argument $f \geq atomic_q^{l_1}$ and therefore $SS[f]$, $d-(n+1) \leq f < d-1$ is already set when p updates $SS[d]$. ∎

Claim 4.4.2 *Let* wss_p^i *be a write&snapshot operation whose $atomic_p^i = f$. Then for every g s.t., $SS[g] \neq \perp$:*

1. *If $g \geq f$ then, $SS[g,p] \geq i$.*
2. *If $g < f$ then, $SS[g,p] < i$.*

Proof: Follows from the definition of $atomic_p^i = f$ (Definition 8) and the fact that the values in the array $my_inc[]$ are incremented by one in each operation. The snapshots are of this array. Thus the summation of the values in each snapshot equals the value of $atomic_q^j$ such that the write $my_inc[q] := j$ is the last such write that is serialized before the atomic snapshot. ∎

Lemma 4.4.3 *A process performs at most* $2n$ *iterations of the big* **while** *loop (Lines* $6 - 29$ *).*

Proof: Consider process p in its pss_p^i operation. From the definition of the *proximate-snapshot* operation it follows that $entry_p^j \leq atomic_p^i + n$. According to Lemma 4.4.1 there must be an f, $atomic_p^i - n \leq f < atomic_p^i$ s.t., $SS[f] \neq \perp$. Following Claim 4.4.2, $SS[f,p] < i$ and therefore $exit_p^i \geq atomic_p^i - n$ so $entry_p^i - exit_p^i < 2n$. ∎

Lemma 4.4.4 *Each write&snapshot operation terminates in at most* $O(n^2)$ *accesses to the shared memory, and at most* $2n^2$ `test&set` *operations on the* test-and-set *array.*

Proof: From Lemma 4.4.3 it follows that the run of the implementation terminates after at most $2n$ iterations of the big **while** loop (Lines $6 - 29$). The cost of the execution of the internal **while** loop in Lines $11 - 27$ is $O(n)$, since the **for** loop in Lines $19 - 23$ is executed only once (in the last iteration). All the other lines have constant cost, and therefore the complexity of the implementation is $O(n^2)$ with at most $2n^2$ operations on the test-and-set array, and it is obviously wait-free. ∎

Claim 4.4.5 *For every* wss_p^i *operation,* $SS[\text{exit}_p^i, p] = i - 1$.

Proof: The values of $entry_p^i$ are monotonically increasing in i, since they are a sequence of summations of serial snapshots that where taken on the $my_inc[]$ registers. From the condition in Line 18 it follows that $SS[exit_p^i, p] < i$. Clearly, when wss_p^i finishes Line 5, $SS[entry_p^i, p] = i$ and for $entry_p^{i-1} < j < entry_p^i$ $SS[j,p] \geq i - 1$ and $SS[entry_p^{i-1}, p] = i - 1$. Thus, in the last execution of Line 18 the Lemma must be satisfied. ∎

Lemma 4.4.6 *Consider a run of the algorithm in which* $capture_{(f,t)} = (p,i)$ *and* $capture_{(f,t')} = (q,j)$ *then:*

1. *if* $t' \geq t$ *then* $R_{(f,t')}[p] = i$
2. *if* $t' < t$ *then* $R_{(f,t')}[p] = i - 1$

The proof of this lemma (given below) follows the proofs of the single-use-write-and-snapshot since all the processes that reach a floor, access a single-use-write-and-snapshot object that is associated with that floor.
Proof:

1. If $t' \geq t$ then it follows from lemma 4.2.4 that $i \in S_q^{f,j}$ and therefore $R_{(f,t')}[p] = inc[p,f] = i$.
2. If $t' < t$ then $i \notin S_q^{f,j}$ and from the code and from claim 4.4.5 it follows that $R_{(f,t')}[p] = SS[f,p] = i - 1$.

∎

Lemma 4.4.7 *Consider a run of the algorithm in which* $capture_{(f,t)} = (p,i)$, $capture_{(f',t')} = (q,j)$ *and* $f < f' < entry_p^i$ *then* $R_{(f',t')}[p] = i$.

Proof: Since wss_q^j terminates in floor f', there must be a process d such that $flag[f',d] = TRUE$ and $d \in S_q^{f,j}$. Since wss_p^i did not terminate with $floor = f'$ but rather with a smaller floor, and since process d had set $flag[f',d] = TRUE$ before decreasing $level[f',d]$, it must be that either:

1. $SS[f',p] = i$ and then $R_{(f',t')}[p] = i$ or,
2. $d \notin S_p^{f',i}$ and then by lemma 4.2.4 $S_p^{f',i} \subset S_q^{f',j}$ and it follows that $p \in S_q^{f',j}$ and $R_{(f',t')}[p] = inc[p,f'] = i$.

∎

Lemma 4.4.8 *For every process p the values* $R_{(f',t')}[p]$ *are monotonically non-decreasing with* (f',t') *(for* (f',t') *such that* $capture_{(f',t')} \neq \perp$*). That is, if* $(f',t') >_{lexicographically} (f",t")$ *then for every process p,* $R_{(f',t')}[p] \geq R_{(f",t")}[p]$.

Proof: Consider two *write&snapshot* operations of process p, wss_p^i s.t., $capture_{(f,t)} = (p,i)$ and wss_p^{i-1} that preceded it.

Let wss_q^j be a *write&snapshot* operation of process q s.t. $capture_{(f',t')} = (q,j)$ then:

1. If $f < f' \leq entry_p^i$ then by lemma 4.4.7 it follows that $R_{(f',t')}[p] = i$.
2. If $f' = f$ then by lemma 4.4.6 if $t' > t$ then $R_{(f',t')}[p] = i$ and when $t' < t$ then $R_{(f',t')}[p] = i - i$.
3. If $entry_p^{i-1} \leq f' < f$ then $R_{(f',t')}[p] = i - 1$ since $SS[f',p] = i - 1$ and since $p \notin S_q^{f',j}$ in these floors.

∎

Lemma 4.4.9 *Consider a run of the algorithm in which* $capture_{(f,t)} = (p,i)$, $capture_{(f',t')} = (q,j)$ *and* wss_q^j *started after* wss_p^i *terminated, then* $(f',t') >_{lexicographically} (f,t)$.

Proof: By contradiction. Assume $(f',t') <_{lexicographically} (f,t)$ then by Lemma 4.4.8 $R_{(f,t)}[q] \geq R_{(f',t')}[q] = j$, i.e., wss_p^i has observed a value written by wss_q^j, which is impossible, contradiction. ∎

Lemma 4.4.10 *Let* $capture_{(f,t)} = (p,i)$ *then for every* $(f',t') < (f,t)$ $R_{(f',t')}[p] < i$

Proof:

Follows immediately from points 2 and 3 in the proof of Lemma 4.4.8. ∎

Theorem 4 *The algorithm in Figure 7 correctly implements a wait-free multi-use write&snapshot.*

Proof: The fact that the algorithm is wait-free immediately follows from Lemma 4.4.4. To proof the Theorem we first define the linearization order on the *write&snapshot* operations in any run, and secondly we prove that the snapshots returned by these operations satisfy the requirements of Definition 14.

We define a total order α on the *write&snapshot* operations and prove that this total order is an extension of the natural partial order β. The partial order β is induced on the operations

by the relation $<_\beta$ defined as follows: $wss_q^j <_\beta wss_p^i$ iff wss_q^j finishes before wss_p^i starts. The total order $<_\alpha$ is defined as follows: $wss_q^j <_\alpha wss_p^i$ iff $capture^{-1}(q,j) < capture^{-1}(p,i)$ (i.e., iff $(f',t') <_{lexicographically} (f,t)$ where $capture_{(f,t)} = (p,i)$ and $capture_{(f',t')} = (q,j)$). By Lemma 4.4.9 the total order α agrees with the partial order β.

Next we prove that the snapshots returned by the *write&snapshot* operations satisfy Definition 14. Requirement (1) of the definition follows from the code that assigns i to $R[p]$ (which is $R_{(f,t)}[p]$). Requirement (2) follows from Lemma 4.4.8. Requirement (3) follows from Lemma 4.4.10.

∎

4.5 Multi-use Immediate-snapshot

An interesting derivative of our algorithm is a multi-shot algorithm for the Immediate-snapshot problem. In [BG93b] Borowsky and Gafni have presented a single-shot implementation of Immediate-snapshot and left the question of a multi-shot implementation open. As was explained before, in the implementation of the multi-use write-and-snapshot we use multi-use immediate-snapshot. Each set of processors that was returned by the immediate-snapshot is then diffracted into individual processes by using test-and-set registers. Since we were unable to employ the multi-use immediate-snapshot as a true black box in the implementation of the multi-use write-and-snapshot, we present the code of the multi-use immediate-snapshot separately in this subsection and in Figure 9. Note that the code is exactly the same as the code given in Figure 7 after omitting Lines 15, 16, and 26 (Line 26 is just a closing "**fi**").

The proof of correctness is essentially the same as the proof of the multi-use write-and-snapshot implementation. The definition of $capture_p^i$ should be changed to:
$capture_p^i$ is the pair of indices $[floor, level[p, floor]]$ of process p when it returns from its i^{th} immediate-snapshot operation.
Then the Lemmas of the previous subsection should be modified to take care of the case $(f',t') = (f,t)$.

5 Bounding the number of registers in the Atomic-write-and-snapshot algorithm

Several entities in the Atomic-write-and-snapshot algorithm are unbounded, these are:

1. The number of registers used (i.e., the number of floors in the array of cells).
2. The number of test-and-set registers used.
3. The size (number of bits) of values written in the different registers, e.g., $floor$, and $my_inc[]$.

In this section we describe how the first two entities may be bounded. Bounding the third entity, the size of the values, was left for further research.

The basic principle in bounding the number of registers used is that each process can compute a bounded set of floor indices such that only these indices in the different arrays owned by that process may be accessed by any other process. The unbounded size arrays are: $SS[]$, $value[]$, $T[]$, $level[]$, $flag[]$, and $inc[]$. Except for $SS[]$ and $T[]$ all the arrays are single-writer-multi-reader. First we will transform the array $SS[]$ to an equivalent set of arrays that are single-writer-multi-reader.

```
operation atomic-write-and-snapshot(input)                                   {at process P_p}
    Shared   SS[0..∞]: array of snapshot vectors initialized to ⊥, SS[0] is initialized to [0, 0, ...] ;
             T[1..∞, 1..n]: array of n-process test-and-set ;
             my_inc[1..n]: array of integers; in swmr snapshot memory initialized to 0;
             value[1..n, 1..∞]: array of swmr registers ;
             level[1..n, 1..∞]: array of swmr registers initialized to n + 1;
             flag[1..n, 1..∞]: array of swmr registers ;
             inc[1..n, 1..∞]: array of swmr registers initialized to 0;
    local    floor, exit_floor, PSS, S, snap;
 1: value[p, my_inc[p] + 1] := input
 2: my_inc[p] := my_inc[p] + 1
 3: PSS := proximate-snapshot(my_inc[1..n])
 4: floor := Σ_{j∈PSS} j
 5: SS[floor] := PSS
 6: while TRUE do
 7:       floor := floor − 1
 8:       exit_floor := FALSE
 9:       flag[p, floor] := (SS[floor] ≠ ⊥)
10:       inc[p, floor] := my_inc[p]
11:       while not exit_floor do
12:             level[p, floor] := level[p, floor] − 1
13:             S := {j | level[j, floor] ≤ level[p, floor]};
14:             if |S| ≥ level[p, floor] then
17:               exit_floor := TRUE
18:               if (∃j ∈ S s.t., flag[j, floor] = TRUE) and (SS[floor, p] < my_inc[p]) then
19:                  for h := 1 to n do
20:                     if h ∈ S then R[h] := inc[h, floor]
21:                       else R[h] := SS[floor, h]
22:                     snap[h] := value[h, R[h]]
23:                  od
24:                  return (snap[1..n])
25:               fi
27:             fi
28:       od
29: od
```

Figure 9: The code of multi-use-Immediate-snapshot

Then we will describe our method and finally the method will be extended to bound the size of $T[]$ as well.

Instead of holding one multi-writer $SS[]$ array, each process maintains a private single-writer multi-reader copy of the array. Since each entry in the array records a real atomic snapshot, and for the same floor index (which is the summation of the snapshot entries) any process observes the same snapshot, all processes that record a snapshot at some floor value record the same snapshot at this floor value. Thus to read a snapshot at $SS[f]$ a process should read all the f entries in each of the new n arrays. A read that finds a snapshot is serialized at its last atomic read and a read that does not find a snapshot in any of the n arrays is serialized at the first atomic read.

The basic idea in bounding the swmr arrays is that each process may have information that might be necessary by another process in at most $6n$ different indices of the arrays. Moreover, each process can compute at any point in time the set of these $6n$ indices for each other process. This computation is based on the following observations:

1. During a single *atomic-write&snapshot* operation each process p scans a consecutive range of $2n$ indices (see Lemma 4.4.3). From $entry_p^i$ to $exit_p^i$.

2. The starting index in each operation of process p, $entry_p^i$, is a result of a *proximate-snapshot* operation by p.

3. The information that process q writes in an operation and that process p might access is either in the $2n$ entries from $q's$ entry point, $entry_q$, and below or in the $2n$ entries from $p's$ entry point, $entry_p$, and below. The former range is in case that q did not observe the beginning of a new operation by p (i.e., q did not see the new value of $my_inc[p]$) and the later range is in case $q's$ operation observes the last operation of p.

4. If process q writes in its arrays information that process p later accesses then q can compute the index value, $entry_p$, that process p might read. Process q computes this index value by mimicking the *proximate-snapshot* operation of process p. That is, when process q performs *proximate-snapshot* it repeats Lines 7 thru 14 in Figure 3 but on the behalf of process p. That is, the repeated Line 10 would be performed as follows:
   10: **if** $S[p]$=$my_minSS[j,i][p]$ and $sum > \sum_{c_k \in my_minSS[j,i]} c_k$ **then**
   The entry that corresponds to the snapshot that q has obtained in this way (called mimicked $entry_p$) is at distance at most n from the entry point of p (i.e., its summation differs from the entry point of p by at most n). Note that if during this operation process p has advanced its $my_inc[p]$ then p would access array entries which are around or above the entry point of q (above $entry_q - 2n$). Thus if p has advanced then q may abort the mimicking of $p's$ *proximate-snapshot* operation.

5. Since in its operation process p may read array entries from $entry_p$ to $entry_p - 2n$ and the mimicking $entry_p$ point computed by q is at distance at most n from $entry_p$ that process p computed, q has to preserve for p all arrays entries from index mimicked $entry_p + n$ to mimicked $entry_p - 3n$.

6. The maximum number of different indices that process q may have to preserve for other processes is thus $4n^2$ ($4n \times (n-1)$ plus the $2n$ entries below $entry_p$).

By Observation 6 the number of entries each process has to maintain of arrays $SS[], level[], flag[]$, and $inc[]$ is $4n^2$. Let us now explain how to bound the number of entries in the array $values[]$ and how to identify the different entries in the reusable memory.

To bound the number of entries that need to be preserved by process q in array $values[q,.]$ we use the above technique. Since the number of entries that process q has information about and that might still be accessed by some process is $4n^2$ the number of input values that q need to preserve is also bounded by $4n^2$. This follows because with each floor index value that q needs to preserve at most one of $q's$ input values is associated. Thus for each of the $4n^2$ indices that q preserves it also preserves the corresponding input value.

Since each array is now kept in a bounded size memory we need to provide a method by which if a certain index is given then the corresponding entry is found. In order to be able to locate an entry we keep with each entry its original index value in the unbounded array. A trivial method to access an entry in index i in array A would then be to scan the entire memory of array A. However, a more efficient method would be to use a hashing function.

It remains to bound the number of test-and-set registers used. To do this we employ a method that we have developed in [Wei94, AWW93] and in which a collection of $n^2/2$ 2-process swap registers are used to implement the test-and-set registers. In that method (see Section 3.1 in [AWW93], or Chapter 3 in [Wei94]) between each two processes one swap register is placed. To perform a *test&set* operation a process has to swap the index of the test-and-set register it accesses into each of the swap registers it shares with the other processes. If in any of the swaps it detects that that index was already swaped in by the competing process it loses and returns from the *test&set* operation. (The order in which processes perform the swap operations with each other is very important and is defined in [AWW93, Wei94, AGTV92]). Following Observations 4 and 5 above each process q knows a set of $6n$ floor values in which another process p may compete with it. Therefore, when process q swaps into the swap register which it shares with p (in a *test&set* operation), it swaps in the indices of all the test-and-set registers that correspond to these floor values and that it won in the past and the index of the one it tries to win currently. If the swap operation returns a vector that contains the index in which the process is trying to win then the process loses and returns from the *test&set* operation. Otherwise the process wins the *test&set*. Since in the Atomic-write-and-snapshot algorithm with each floor n test-and-set registers are associated, each swap register has to hold at most $6n^2$ indices. It is possible to reduce the number of indices kept in each swap register from $6n^2$ to $6n$ by keeping only one index for each floor, however the many details of this implementation are beyond the scope and space of this paper.

6 Conclusions and Implications

The commuting objects constructed in this paper are part of the *common2* class of objects introduced in [AWW93]: The class of objects, called *common2*, 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 [Wei94, AWW93] we present implementations of any read-modify-write synchronization object that applies *overwriting* functions, in a system with an arbitrary number of processors, from any other shared object whose consensus number is 2 or more.

Thus, together with [Wei94, AWW93] this paper provides 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 Herlihy's open question ("Can fetch-and-add implement any object having consensus number two in a system of three or more processes?") with respect to the class *common2*.

Our result has three additional implications: First, we show that there are fault tolerant implementations for n processes commuting objects from any set of such objects that contains some faulty objects. This is the result of combining the constructions of this paper with the constructions for two processes given in [AGMT92, JCT92].

The second implication of our result are new randomized implementations for commuting objects 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, SSW91, 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 $O(n^3)$ fetch-and-add, as oppose to $O(n^4)$ for either one in [Her91b].

The third implication of our results are low contention implementations of commuting synchronization objects. This follows from the fact that the basic building block of all our constructions are 2-process synchronization objects and read/write registers. 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 [AHS94, 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).

This work 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 and to Hanan Weisman for many enlightening discussions and collaborations. We thank Nir Shavit, and Gideon Stupp for many helpful discussions. In particular Gideon's patients was inparticular inspiring.

References

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[ADS89] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 281–293, Edmonton, Alberta, Canada, August 1989.

[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proc. 11th Ann. ACM Symp. on Principles of Distributed Computing*, pages 47–58, August 1992. JACM.

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 85–94. Springer Verlag, November 1992.

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

[AHS94] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[And90] J. H. Anderson. Composite registers. In *Proc. of the Ninth Ann. ACM Symp. on Principles of Distributed Computing*, pages 15–29, August 1990.

[And93] Anderson. Composite registers. *Distributed Computing*, 6, 1993.

[AR93] H. Attiya and O. Rachman. Atomic snapshot in $o(n \log n)$ operations. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 29–40, August 1993.

[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.

[AWW93] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 159–170, August 1993.

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

[BG93a] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[BG93b] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 41–51, August 1993.

[DS89] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proc. of the 21st Ann. ACM Symp. 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 proc. *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*, 13(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] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. In *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, pages 526–535, October 1991.

[HW87] M. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, January 1987.

[Jay93] P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 145–158, August 1993.

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

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. Technical Report 34, Bell Sys. Tech., September 1955.

[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.

[SSW91] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proc. of 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 351–362, January 1991.

[SZ93] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[Wei94] H. Weisman. Implementing shared memory overwriting objects. Master's thesis, Tel-Aviv University, 1994.