ELSEVIER

Theoretical Computer Science 269 (2001) 163–201

Theoretical Computer Science

www.elsevier.com/locate/tcs

Towards a practical snapshot algorithm, ☆, ☆☆

Yaron Riany[a], Nir Shavit[a,b,*], Dan Touitou[a]

[a] *Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel*
[b] *Sun Microsystems Laboratories, One Network Drive, Burlington MA 01803-0902, USA*

Received 15 July 1999; revised 13 July 2000; accepted 30 October 2000
Communicated by M. Mavronicolas

Abstract

An *atomic snapshot memory* is an implementation of a multiple-location shared memory that can be atomically read in its entirety without preventing concurrent writing. The design of wait-free implementations of *atomic snapshot memories* has been the subject of extensive theoretical research in recent years. This paper introduces the *coordinated-collect* algorithm, a novel wait-free atomic snapshot construction which we believe is a first step in taking snapshots from theory to practice. Unlike previous algorithms, it uses currently available multiprocessor synchronization operations to provide an algorithm that has only O(1) update complexity and O(n) scan complexity, with very small constants. We evaluated the performance of known snapshot algorithms for a collection of benchmarks on a simulated distributed shared-memory multiprocessor. Our empirical evidence suggests that *coordinated-collect* will outperform all known wait-free, lock-free, and locking snapshot algorithms in terms of overall throughput and latency.

Keywords: Shared memory multiprocessors; Wait-free synchronization; Memory snapshots; Atomic operations; Compare and swap

1. Introduction

An *atomic snapshot memory* [2, 6] is an implementation of a multiple-location shared memory that can be atomically read in its entirety. The ability to collect such an instantaneous view is a powerful tool for designing concurrent data structures, as it

☆ A preliminary version of this work appeared in the Proc. Third Israel Symp. on Theory and Computer Systems (ISTCS '95), Tel-Aviv, January, 1995, pp. 121–129.

☆☆ Part of this work was performed at Tel-Aviv University and at M.I.T. with funding from the Israel Science Foundation under grant 03610882, the Israeli Ministry of Science, and NSF grants 9225124-CCR and 9520298-CCR.

* Corresponding author.
E-mail address: shanir@cs.tau.ac.il (N. Shavit).

PII: S0304-3975(00)00412-6

greatly reduces the need to argue about inconsistent views of memory. Snapshots have been widely used in theoretical work [1, 4, 12, 14, 15, 23, 26], and offer a yet untapped potential for practical use in applications such as check-pointing, generating concurrent backups, or interactive debugging of multiprocessor programs. Snapshots can also serve as building blocks of state-of-the-art fault-tolerant real-time applications such as a multiprocessor server of a radar tracking system, where multiple sensors generate updates concurrently with multiple requests for consistent system states. The design of asymptotically efficient implementations of an *atomic snapshot memory* has been the subject of extensive and highly creative research in recent years [2, 6, 8, 11, 13, 18, 21, 30].

An atomic snapshot memory is an abstract data-type equivalent to a memory partitioned into n segments, one for each processor. There are two types of operations on the object, a *scan* and an *update*. In an update operation, a processor writes the contents of its associated segment, while in a scan, it obtains an instantaneous "global picture" of all n segments. Snapshots should be fault-tolerant and non-interfering. That is, applications (for example, programs being check-pointed) on the system should have minimal disruption or loss of performance as a result of ongoing snapshots, and in the extreme should continue to run even in the face of severe timing anomalies. An atomic snapshot implementation is *wait-free* if the execution of any implemented scan or update operation completes within a bounded number of machine instructions independently of the pace of other processors [24]. Fault-tolerance and non-interference are the major advantages of wait-free methods over standard lock-based implementations.

This paper takes a practical look at the question of providing wait-free implementations of atomic snapshots on multiprocessor architectures. A snapshot implementation that is to be practical should have the following properties:

- The complexity of performing an *update* operation should be within a small constant of that of a simple "write" to memory, since the typical user does not want to sacrifice the speed of updating memory to support efficient snapshots.
- Register sizes and hardware synchronization primitives should conform with ones available on multiprocessor architectures.
- Memory contention should be minimized by distributing and load-balancing work, otherwise good asymptotic complexity will not result in good performance.

1.1. Coordinated collecting

The main contribution of this paper is in introducing the *coordinated-collect* algorithm, a novel atomic snapshot construction which we believe is a first step in taking snapshots from theory to practice. It uses *Load-linked*/*Store-conditional* operations [19, 28, 32] to provide a multi-scanner algorithm[1] that uses real-world registers (each containing at most O(1) values, where a value is typically at least the size of a processor identifier) with only O(1) update complexity (in fact, at most *four* operations), O(n) scan complexity, and O(n^2) space complexity.

[1] A multi-scanner algorithm is one in which concurrent scan operations by different processors are allowed.

Table 1
A comparison of atomic snapshot algorithms

Snapshot algorithm		Primitive used	Update complexity	Scan complexity	Register size	Space complexity
Lock free		r/w register	$O(1)$	∞	$O(1)$	$O(n)$
Block update		r/w register	∞	$O(n)$	$O(1)$	$O(n)$
Anderson [6]		r/w register	$O(2^n)$	$O(2^n)$	$O(1)$	$O(n^3 \log n)$
Aspnes and Herlihy [8]		r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$
Afek et al. [2]	Unbounded	r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
	Bounded	r/w register	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$
Kirousis et al. [30]	One scanner	r/w register	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Dwork et al. [21]	Weak snapshot	r/w register	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$
Chandra Dwork [18]		LL/SC	$O(n)$	$O(n)$	$O(n)$	∞
Attiya and Herlihy	Version 1	T&S	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	∞
Rachman [11]	Version 2	dyn. T&S	$O(n)$	$O(n)$	$O(n)$	∞
Attiya and Rachman [13]	Unbounded	r/w register	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	∞
Coordinated collect		LL/SC	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$

Though one might think that the use of strong primitives like *Load-linked/Store-conditional* would allow us to readily modify the elegant snapshot algorithms in the literature [2, 6, 8, 11, 13, 18, 21] to achieve similar complexity, it turns out that this is not the case (see the summary in Table 1). These multi-scanner snapshot protocols have an algorithmic structure in which each updater and/or scanner collects a view of memory in its register, and then processors try to agree which of the views to return. This leads to a situation where, even with the added power of a *Load-linked/Store-conditional* operation to speed up the view-agreement process, the complexity of an update remains an unacceptable $\Omega(n)$, and the registers used in the algorithms are required to hold simultaneously $\Omega(n)$ values.

Our presentation begins with the introduction of a new single-scanner protocol[2] – a greatly simplified version of the innovative single-scanner protocol of Kirousis et al. [30]. We build the *coordinated-collect* multi-scanner algorithm based upon our single-scanner protocol. The algorithm has updaters perform the same $O(1)$ sequence of operations as in our single-scanner algorithm, but uses a novel collection methodology to allow multiple scanners to return coherent views of memory. Instead of deciding on one of many collected views as in previous algorithms, *coordinated-collect* has all the active scanners distribute the work and 'help' each other to collect values from the n registers into a pre-agreed shared view area. This allows us to achieve an $O(n)$ scan complexity without increasing the update complexity. The helping process is tailored to maintain low contention by load-balancing processors over the shared view locations.

[2] The version we present uses unbounded time stamps (60-bits or more will suffice in practice) but can be easily bounded using a sequential time-stamp system [29, 34].

1.2. A comparison of atomic snapshot algorithms

The second contribution of this paper, in Section 6, is a comparison of the performance of several single- and multi-scanner algorithm snapshot techniques, including our own, on a simulated distributed shared-memory multiprocessor using the well accepted Proteus Parallel Hardware Simulator [16, 17] of Brewer et al. Our choice of algorithms for simulation was driven not only by their asymptotic complexity, but also by the feasibility of implementing them on multiprocessor machines.

The first two compared methods are an algorithm that blocks updates during a scan and a lock-free algorithm that never blocks updates but does not guarantee scan termination in the face of repeated updating. Of the known wait-free methods, we chose to implement the unbounded-register versions of the algorithms of Afek et al. [2] and Attiya and Rachman [13], and the consensus-based algorithm of Chandra and Dwork [18]. The first two algorithms use n-valued read/write registers to have processors agree among collected views, and the last uses n-valued registers and an agreement mechanism which we implemented using the powerful *Load-linked/Store-conditional* operation. We did not implement the intricate *test&set* based algorithms of Attiya et al. [11] which achieve asymptotically efficient agreement among views using an unbounded number of *test&set* registers. Transforming them into bounded algorithms would introduce a substantial overhead in space and in memory contention, making them inferior to the *Load-linked/Store-conditional* based agreement scheme which we tested. Given that the above algorithms assume the availability of atomic n-value registers, we tested them both under the (unrealistic) assumption that such operations are available in hardware, and under the (more realistic) one that each n-valued read operation takes at least n local operations. We found that their performance was only slightly improved by assuming n-valued registers were available in hardware.

We found that our single-scanner and multi-scanner *coordinated-collect* algorithms outperform all known algorithms both in throughput and latency. Surprisingly, their update throughput is as good as that of the lock-free method which lets updates succeed at the price of very low scan throughput.[3] The scan throughput of our algorithms remains consistently high as the number of processors increases, even though the size of the collected views grows linearly. However, it has an associated overhead and generates a certain level of contention which prevents it from reaching the throughput of the blocking algorithm (one which blocks all updates during a scan).

In summary, we believe our work is an example of using current multiprocessor synchronization operations to develop snapshot algorithms that are more "realistic" in terms of register size and the complexity of update operations. Our hope is that this and future improvements in the performance of this important building block will help in advancing wait-free data structures from theory to practice.

[3] In the lock-free algorithm, the scanner repeatedly collects the contents of the registers. If it reads the contents of the registers twice, and no register has been changed, it returns the collected values as a result.

The paper is structured as follows. Section 2 presents the model. In Section 3 we present our single-scanner algorithm. Section 4 presents our multi-scanner algorithm. The proofs of both algorithms can be found in Section 5. Finally, Section 6 presents a performance analysis of the algorithms.

2. Snapshots and the shared memory model

Our computation model follows [12, 13, 27]. A *concurrent system* consists of a collection of *n processors*. Processors communicate through shared data structures called *objects*. Each object has a set of primitive *operations* that provide the only means to manipulate that object. Each processor P is a sequential thread of control [27] which applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations where an operation A *precedes* another operation B if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. A *sequential history* is a history in which each invocation is followed immediately by its corresponding response. The *sequential specification* of an object is the set of *legal* sequential histories associated with it. The basic correctness requirement for a concurrent implementation is *linearizability* [27]: every concurrent history is "equivalent" to some legal sequential history which is consistent with the real-time order induced by the concurrent history. In a linearizable implementation, operations appear to take effect atomically at some point between their invocation and response. In our model every shared memory location L of a multiprocessor machine memory is a linearizable implementation of an object which provides every processor P_i with the following set of sequentially specified operations (see [24, 25] for details):

$Read^i(L)$ reads location L and returns its value.

$Load\text{-}linked^i(L)$ reads location L and returns its value. Marks L as read by P_i.

$Store\text{-}conditional^i(L, v)$ if location L is marked as read by P_i, the operation writes the value v to L, erases all existing marks on L and returns a *success* status. Otherwise it returns a *failure* status.

$Write^i(L, v)$ writes the value v to location L. Erases all existing marks on L.

Since the *Load-linked* and *Store-conditional* operations on some machines have a different semantics than those described above, we discuss in Section 6 how one can implement them using existing synchronization primitives.

2.1. Atomic snapshots

An *atomic snapshot* object is a concurrent shared object which allows each processor P_i, where $0 \leqslant i \leqslant n-1$ to perform two types of operations on the object: $Update^i(r)$ and $Scan^i$. Each $Update^i$ operation has input r from some given range of *values*. Each $Scan^i$ returns a vector $view[0..n-1]$ of n values.

We require that a *correct* implementation of an atomic snapshot object meet the following sequential specification:

Definition 2.1. In every finite or infinite sequential history $h = o_1 o_2 o_3 \ldots$, if $o_k = Scan^i$ where $Scan^i$ returns $view[0..n-1]$, then for all j such that $0 \leqslant j < n$, the last $Update^j(r)$ operation (last meaning with highest subscript) in the subsequence $o_1 \ldots o_{k-1}$, has $r = view[j]$. If $o_1 \ldots o_{k-1}$ does not contain any $Update^j(r)$ operations then $view[j] = empty$.

In terms of our implementation, proving correctness amounts to showing that each possible concurrent history of the object's execution fills the following conditions:

(P1) The induced real-time order on the implemented *Scan* and *Update* operation executions can be extended to a total order, such that

(P2) the totally ordered history is *legal*, that is, meets the sequential specification.

We also require the implementation to be *wait-free*, requiring the execution of any *Scan* or *Update* operation to complete within a bounded number of machine instructions [26].

3. The single-scanner algorithm

The following algorithm is a greatly simplified variant of the Kirousis, Spirakis, and Tsigas single-scanner snapshot algorithm [30]. The latter algorithm is based on the innovative idea of letting the scanner change, in each scan, the memory location where updaters write their values. This allows updaters to keep writing to memory without disrupting an ongoing scan operation's attempt to collect a snapshot view. However, the price paid in [30] is a rather complicated scheme to eliminate the need to search back through an unbounded number of possible locations in which updates were recorded. The key to our algorithm, presented below, is a simple pragmatic structure that eliminates the need for such a search with almost no overhead and achieves optimal asymptotic time complexity, $O(n)$ for a scan and $O(1)$ for an update. Moreover, as can be seen from the code in Fig. 1, the constants involved are very small. The algorithm uses a sequential time-stamp system [29], a mechanism for maintaining order among events using sequence numbers. In practice, implementing the "unbounded" sequence numbers we use in the algorithm requires a register `curr_seq` of 60 bits or more (which on current multiprocessors will take hundreds of years before it overflows) though one can readily replace it by a more theoretical bounded sequential time-stamp system [20, 29, 34] (one that never overflows) to achieve a simple *bounded* read/write register single-scanner protocol.

The algorithm is designed to allow carrying the *update* operation structure without change over to the case of multiple scanners. The code for *scan* and *update* operations appears in Fig. 1.

The algorithm uses a shared array $memory[0..n-1]$ of records, each having a `high` and a `low` field, and a shared variable `curr_seq` which holds a current time-stamp

```
Shared data structures:
  curr_seq: integer;
  type value: integer or Empty or Null;
       Register: record [ val: value, seq:integer ];
       memory: array [0..n-1] of record [high:Register,low:Register];

shared data structures initialization:
  for all k=0..n-1
       memory[k].high = memory[k].low  = [Empty, 0];
  curr_seq   = 0;

Scan()
variables
  view: array [0..n-1] of value;
  high_r, low_r: register; j: integer;
begin
       curr_seq := curr_seq+1;
       for j := 0 to n-1
           high_r := memory[j].high;
           if (high_r.seq < curr_seq)
                view[j] := high_r.val
           else
                view[j] := memory[j].low.val;
       return view;
end; {Scan}

Update(val)   {for processor Pi}
variables
   seq : integer; high_r: register;
begin
       seq := curr_seq;
       high_r := memory[i].high;
       if (seq <> high_r.seq)
            memory[i].low := high_r;
       memory[i].high := [val,seq]
end; {Update}
```

Fig. 1. The single-scanner algorithm code.

(sequence number). The basic idea is to let the single-scanner set a new time-stamp as its first step and then collect updaters' values, while the first operation in each update is to read this time-stamp.

The scanner classifies the collected updaters' values according to its newly created time-stamp, and returns only those values that were written by update operations which did not read its time-stamp. These updates must have started and performed the read of `curr_seq` before the scan, and so all their associated values *could have existed* in memory at the point in the execution when the increment of the time-stamp was performed (note that the fact that the values could have existed implies linearizability [27]). To guarantee that such a value is found for each updater, even if scanning is

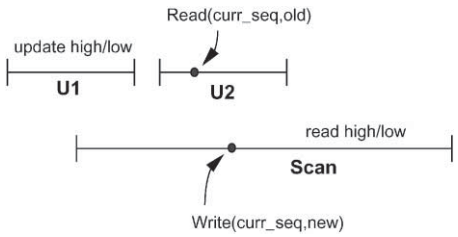

Fig. 2. Ordering updates and scans.

concurrent with updating we keep, in an additional memory location (the low field), the latest value updated with a time-stamp preceding that of the current/latest scan. Therefore, in case an updater "sees" a scan which has started after its last update operation it modifies the low and high fields of its update location. First, it updates the low field to hold its previous update value which should be available for concurrent scans, and then the high field to hold the new update value.

To prove the correctness of this construction we must show that each possible history of this algorithm is equivalent to a legal history [27]. A detailed formal correctness proof of the algorithm follows from that of the *coordinated-collect* algorithm and is presented in Section 5. In a nutshell, given a concurrent history of the single-scanner algorithm, we order every *Scan* operation S and *Update* operation U in the following way: If U reads `curr_seq` after S increments it, then U is ordered after S. If U reads `curr_seq` before S increments it, then if S reads `memory[i].high` before U writes into it, U is ordered after S otherwise U is ordered before S. Two update operations, U_1 and U_2 are ordered as follows: if there exists a *Scan* operation S, such that U_1 is ordered before S and U_2 is ordered after S, then U_2 is ordered after U_1. Otherwise U_1 and U_2 are ordered according to the order of their respective read operations on `curr_seq`.

Clearly, this total order extends the partial order induced by the concurrent history. The only case for which this is not immediate is for two update operations U_1 and U_2 that are totally ordered in the concurrent history. Assume, without loss of generality, that U_1 precedes U_2, and let us see why they are consistently ordered with respect to scan operations. As depicted in Fig 2, if U_2 is ordered in the total order before some scan operation S, then U_2 must have read the content of `curr_seq` before S incremented it. This implies that U_1 updated its registers in the `memory` array before S incremented `curr_seq` and therefore before S read the `memory` array. Therefore, we order U_1 before S.

4. The coordinated-collect algorithm

To achieve an O(1) update complexity, we build the multi-scanner algorithm around our single-scanner algorithm. As in the single-scanner algorithm, we keep a `memory`

array and a curr_seq time-stamp, and have updaters perform the sequence of operations in Fig. 1. We begin our multi-scanner design by noticing that a solution based on simply incrementing the curr_seq in every scan will force updaters to maintain a set of O(n) last-updated values (one for each ongoing scan). This will affect not only the update complexity but the scan complexity as well. On the other hand, incrementing the curr_seq only once for a collection of concurrent scans, will lead to a situation where updaters can switch values of high/low memory locations in the middle of a scan. Establishing an order among concurrent scans will thus not be possible. Take, for example, an execution in which two update operations U_1 and U_2 on locations i and j, respectively, are executed concurrently with two scan operations S_1 and S_2. Assume that both U_1 and U_2 have read curr_seq before it was incremented. In that case we may have a concurrent history in which S_1 read memory[i].high before U_1 wrote in it and read memory[j].high after U_2 wrote in it, while S_2 read memory[i].high after U_1 wrote in it and read memory[j].high before U_2 wrote in it. Consequently, S_1 should have collected the value written by U_1 but not the value written by U_2, and S_2 should have collected the value written by U_2 but not the value written by U_1. This creates a situation in which S_1 and S_2 cannot be ordered consistently. In short, to make our efficient single-scanner algorithm work for multiple scanners, we must ensure that there are *no concurrent scans*.

Since we cannot prevent scans from actually taking place concurrently, our approach is to have them emulate a sequence of *virtual* scan operations whose execution intervals do not overlap. These sequential virtual scans together with the regular single-scanner updates form executions. In these executions the values collected are identical to the values collected by the scanner in the single-scanner algorithm. The key to our multi-scanner construction is to guarantee that each of the concurrent scan operations completely overlaps at least one virtual scan and returns its value. The total linearization order on concurrent scan and update operations is defined by first ordering the updates relative to the virtual scans and then ordering each of the concurrent scans according to the order of its overlapped virtual scan, that is, as if it occurred at some point within the virtual scan interval.

To create the sequence of non-overlapping virtual scans, we let all scanners that execute concurrently share a special variable curr_index that points to a pre-chosen view: an array of n locations in which the current virtual scan's snapshot "view" of memory will be collected. Scanners start a new virtual scan operation only after the current one completes. To provide wait-free behavior, each scanner must guarantee that the view is collected even if all other scanners that share it have halted. This means that (in the worst case) it must execute the collection code of the single-scanner scan operation for all memory locations. Scanners working concurrently on the same view might still read different values from memory, and so we must make sure that only one snapshot view is actually written to the shared view array. We do so using a *Load-linked* and *Store-conditional* operation to guarantee that each view location is written by only one of the concurrent scanners. The reason for using a *Load-linked* and *Store-conditional* operation as opposed to a simple write is that

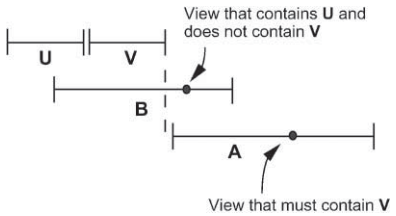

Fig. 3. Returning an incorrect view.

otherwise, delayed update operations that started before the current virtual scan may cause concurrent scanners to overwrite view locations. In such cases, delayed scanners may overwrite (and return) some already collected (and returned) snapshot. Similarly, scanners running independently would eventually return snapshot results that cannot be ordered consistently.

Having constructed a mechanism that ensures that each scanner will collect a snapshot view, we turn to guaranteeing that it can return a virtual scan operation that it completely overlaps. The problem is similar to that noticed by Afek et al. [2]. If the view returned by scanner A was collected by another concurrent scanner B, the values U that are returned by A might be incorrect. As depicted in Fig. 3 the reason is that there may be updates V completely after the updates U and completely before A's scan, that are "missed" by returning the values in U. The solution is to use a variant of the *double collect* technique [2], namely, participate in two virtual scans and return the view collected in the later one. A scanner participates in a virtual scan collection only after getting a shared view pointed to by `curr_index` and verifying that it is still not fully collected with snapshot values. Since by definition virtual scans are never concurrent, the second of them is completely within the scan interval.

However, there can still be a situation in which a scanner infinitely often fails to have access to the view pointed to by `curr_index` and also fails to participate in virtual scans of two chosen views pointed by `curr_index`. This may happen if it is delayed between the agreement on the current view and the view collection, and meanwhile, all other scanners complete the collecting process using the chosen view. We overcome this problem by adding an array $A[0..n-1]$ indexed by `curr_index`. The ith entry of `A` is used by scanner P_i to post its proposed `view`. The new role of `curr_index` is to cycle among the `A` entries. As we prove below, every scanner is ensured to have its proposed view (its entry in `A`) chosen within n virtual scan rounds. This means that if a scanner fails to participate in a virtual scan n times, its own view must have been chosen as some virtual scan's returned view at some earlier point, hence it is filled with snapshot collected values (see Fig. 5).

We now turn our attention to the possibility of having delayed scanners, ones that have not completed reading the view of the virtual scan they decided to collect. If we had unbounded space available, we could simply never re-use a view area, and a

```
type value: integer or Empty or Null;
     index: integer or Null;
     Register: record [val: value; seq: integer];
     View : record [version: integer; sequence:
     index; regs: array[0..n-1] of value];
     Ind : record [value: integer; state: {fill, advance}];

shared data structures:
  A: array [0..n-1] of index;
  curr_index: Ind;
  curr_seq: integer;
  memory: array [0..n-1] of record [high:Register; low:Register];
  views : array [0..2n-1] of View;

shared data structures initialization:
  for all k = 0..n-1
        A[k] = Null;
        memory[k].high = memory[k].low = [Empty, 0];
  curr_index = [0,advance]; curr_seq = 0;

private data structures
  current_view: {0, 1};
  result: array [0..n-1] of value;
  fill_count: integer;

private data structure initialization
  current_view = 0;
```

Fig. 4. Data types and variables of the coordinated-collect algorithm.

delayed scanner could participate in a virtual scan and collect the contents of the shared view at its own pace. However, we observe that any view space that has been filled must be kept for no more than n virtual scan rounds before it can be recycled and used in a new scan. The reason is that after n rounds, every potentially delayed processor will have its own proposed view filled. Therefore, each processor can initially allocate two different view spaces that will be alternately proposed in its `A` entry. Each time it has to propose a view, the processor will pick its last unused view, invalidate it for delayed processors, reinitialize its fields, and write the view address in its `A` entry. Then it will attempt to set its proposed view for the next virtual scan by advancing the `curr_index` pointer and setting it to point to this view. Concurrent scans will be able to agree on one of their proposed views by using *Load-linked*/*Store-conditional* operations to set `curr_index`. Having set `curr_index`, all scanners will try to participate in the collection of the view. If a scanner is successful twice, it will return the view's contents.

Note that allocating only one view per processor does not suffice. This is because we may have a situation in which each time a processor helps some other processor to complete a scan, the helped processor initiates a new scan operation and reuses its view before the helping processor has a chance to copy the view's content. In such a case, the helping processor may have to help $n-1$ other processors before ensuring

```
Scan()
variables
 helped_view, k: integer;
 index: Ind;
 first : boolean;
1: first := True; fill_count := 0;
   forever  do
2:  if (A[i] = Null)
3:      if first
           index := curr_index;
4:          if ((index.value mod n) <> i) or
               (index.state <> fill)
5:              first := False;
6:              current_view = 1-current_view;
7:              Init(2*i + current_view);
8:              A[i] := 2*i + current_view;     {Propose the view}
9:      else
           return views[2*i + current_view].regs;
10:  index := LL(curr_index);
     k := index.value mod n;
11:  helped_view  := A[k];
12:  if (index.state = fill)         {If a collection process is in progress}
13:      if (helped_view <> Null)
14:          if (help_fill(helped_view, index)
               = SUCC and fill_count = 2)
               return result;
        else                    {The view has been filled already}
15:          SC(curr_index,[index.value,advance]);
    else
16:      if (A[(index.value+1) mod n] <> Null)  {If there is a pro-
posed view in the next entry}
17:          SC(curr_index,[index.value+1,fill]);   {Advance and begin
                                                     a collection process}
        else
18:          SC(curr_index,[index.value+1,advance]);        {Advance to
                                                             the next Entry}


Init(x: integer)
19: views[x].version := views[x].version + 1;
20: views[x].sequence := Null;
   for k := 0 to n-1
21: views[x].regs[k] := Null;
```

Fig. 5. *Scan* by processor i in the coordinated-collect algorithm.

that its own view has been filled. This will cause the complexity of the scan operation to be $O(n^2)$.

The following section provides a more detailed description of the algorithm's code. The multi-scanner algorithm's pseudo-code for a scan operation appears in Figs. 5 and 6, while the code of the update operation is exactly as in the single-scanner case of Fig. 1.

```
Help_fill(x: integer, index: Ind)
22: version := views[x].version;
23: if (curr_index <> index)
        return FAIL;
    {If the collection process is still active}
24:  if (LL(views[x].sequence) = Null)
        if (version <> views[x].version)
            return FAIL;
        else
            SC(views[x].sequence,curr_seq+1);

25:  seq = views[x].sequence;
     if (LL(curr_seq) = seq-1)
        if (version <> views[x].version)
            return FAIL;
        else
            SC(curr_seq,seq);
26:  for j := 0 to n-1 do
     {For each updated shared register follow the single
     collect method}
27:      high_r := memory[j].high;
         low_r := memory[j].low;
28:      if (LL(views[x].regs[j]) = Null)
             if (version <> views[x].version)
                 return FAIL;
             else
29:             if (high_r.seq < views[x].sequence)
30:                 SC(views[x].regs[j],high_r.value);
                else
31:                 SC(views[x].regs[j],low_r.value);
32:  if (LL(A[index.value mod n]) = x)
33:      if (curr_index = index)
             SC(A[index.value mod n],Null);
34:  fill_count := fill_count+1;
35:  if (fill_count < 2)
         return SUCC;
     for  j := 0 to n-1 do       {Copy the shared view
                                 registers into private space}
         result[j] := views[x].regs[j];
     if (views[x].version <> version)
           return FAIL;
     else
           return SUCC;
```

Fig. 6. *Help_fill* by processor i in the coordinated-collect algorithm.

4.1. Implementation details

As mentioned earlier, the algorithm uses an array $A[0..n-1]$ of indexes in `views`, an array of views (see Fig. 4). The shared variable `curr_index` contains a `value` field and a `state` field. The `value` field is an index into `A` and the `state` field indicates whether the view specified by `A[curr_index.value]` is subject to an ongoing collection process (which we call a `fill`). If not, the `curr_index` should be advanced (i.e.

advance). Each view space contains: an array regs[0...$n-1$] in which a virtual scan's snapshot "view" of memory will be collected, a sequence field on which processors will dump the content of curr_seq, and a version field which contains an integer representing the number of times this view was proposed. A processor invalidates a view for a delayed processor by incrementing the version field of the view by one.

The algorithm executes as follows. A scanner repeatedly tries to participate in collecting a view until it has taken part in two view collections or until a view it has proposed after the beginning of the current scan operation is filled for the second time. Note that the algorithm uses a flag first to ensure that the scanner will not return its own view after the first fill (line 3). The Null value is stored in the scanner A's entry at either the initialization of the snapshot object or after the scanner's proposed view has been filled.

In detail, a scanner first checks whether no view is currently proposed in its entry (line 2). If this is the case, then if the last proposed view was filled after the scan began, the scanner terminates and returns the content of its own view as the result (line 9). Otherwise the scanner proposes a new recycled view (lines 6–8). The scanner then reads the contents of curr_index and the next view to be filled according to curr_index (lines 10 and 11). If the view was not filled, the scanner tries to participate in an ongoing view collection. This can be done only if the collecting process has not terminated yet (line 13). In this case the scanner calls the Help_fill procedure (line 14). Otherwise, the scanner changes the index status to advance (line 15). If there is no ongoing collection performed on the view pointed to by the curr_index, the scanner increments curr_index and sets its status to:

- fill if there is some proposed view in the next index (line 17), and to
- advance (line 18) otherwise.

A scanner participating in a coordinated collection on some view (procedure Help_fill in Fig. 6), first reads the content of the version field of the view (line 22). Since the owner of the view has to increment the version field before initializing the view, the scanner will be able to check before each coordinated step that the view was not invalidated by its owner. This prevents a delayed scanner from collecting a view after it was reinitialized and proposed again by its owner. After reading the version (line 22), the scanner verifies that the curr_index has not changed (line 23). If curr_index has changed, the view is already filled and the processor exits the procedure.

As in the single-scanner algorithm there is a time-stamp which is incremented at the beginning of each view collection. The time-stamp is incremented in the following way: the scanners first agree on the content of curr_seq by writing its previous value to the filled view (line 24), then they update curr_seq according to the value stored in the view (line 25). In this way we ensure that curr_seq is incremented only once. In the view collection (lines 26–31) scanners agree on the contents of the read register by writing only to "Empty" registers (using *Load-linked* and Store-conditional). Finally, the scanners complete the view collection by writing a Null value in the entry in A indexed to by curr_index (lines 32–33).

5. Correctness proof

By the locality property of linearizeable objects [27] it is a valid technique to ignore any specific implementation details of operations like *Read*, *Write*, *Load-linked* and *Store-conditional* , and to assume that these operations occur as atomic actions sometime within their corresponding operation interval. We may therefore assume that an execution of the coordinated collect algorithm is, in fact, a (possibly infinite) sequence of events $r = e_1,\ e_2, \ldots$ where each event is one of the form:

$W^i(var, val)$ Processor i performs a *Write* operation on variable *var* with value *val*.
$R^i(var, value)$ Processor i performs a *Read* operation on variable *var* which returns value *val*.
$LL^i(var, val)$ Processor i performs a *Load-linked* operation on variable *var* which returns value *val*.
$SC^i(var, value, success)$ Processor i performs a successful *Store-conditional* operation on variable *var* with value *val*.
$SC^i(var, value, failure)$ Processor i performs a unsuccessful *Store-conditional* operation on variable *var* with value *val*.

Sometimes we use the notation $R^i(variable, \Phi(variable))$ or $LL^i(variable, \Phi(variable))$ for some predicate Φ, as a short form for $R^i(variable, value)$ or $LL^i(variable, value)$ and Φ (value) $= true$. In other words, $R^i(variable, \Phi(variable))$ means that processor i reads the content of *variable* and gets a value that satisfies Φ. When the subscripts are clear from the context or unimportant we will at times omit them. For every execution $r = e_1,\ e_2 \ldots$ we refer to the content of every shared memory location *loc* after every event e as $Content(r, e, loc)$, subject to the specification of the primitives used. We use $*$ to replace a value unimportant in a given context. We use $SC^i(var, value)$ as a shorthand for $SC^i(var, value, success)$.

Definition 5.1. For every execution r, the *real-time* order of events in r (denoted as "$\rightarrow$") is the total order such that for two events e_i and e_j in r, $e_i \rightarrow e_j$ *iff* $i < j$. The real-time order of subsequences is the partial order $\overset{seq}{\rightarrow}$ such that for two subsequences $s = e_{j_1} \ldots e_{j_l}$ and $s' = e_{j'_1} \ldots e_{j'_{l'}}$, $s \overset{seq}{\rightarrow} s'$ iff $e_{j_l} \rightarrow e_{j'_1}$.

For simplicity we usually write $s \rightarrow s'$ instead of $s \overset{seq}{\rightarrow} s'$.

Let $Update^i(v)$ be a call to the procedure `Update` by processor i with input v. Then given an execution r, we denote by $U^i(v)$, the sequence of events in r performed by i during $Update^i(v)$. Similarly, given $Scan^i(view[0 \ldots n-1])$ a call to procedure `Scan` by i that returned $view[0 \ldots n-1]$, let $S^i(view[0 \ldots n-1])$ be the subsequence of events in r performed by i during $Scan^i(view[0 \ldots n-1])$.

In order to prove the linearizability of our algorithm we have to show that for every execution, we can provide a total ordering $\Rightarrow$ between the calls to Update and Scan, which is consistent with the partial order between the event subsequences created by the calls, and to show that this total order induces a legal history. Informally, the proof

will be structured as follows. We first show that although many processes may execute a scan operation concurrently, there is at most one "scaning process" in progress at any moment. We then attribute to every such "virtual" scan a vector $view[0 \ldots n-1]$ as its returned value. Our next step is to show that one can linearize the sequence of virtual scans and the sequences of updates to create a legal history. To complete the proof we show that every "actual" scan operation can be matched with a returned view of a "virtual" scan that is contained within its execution interval.

The following two claims hold directly from the algorithm and the specification of the *Load-linked* and *Store-conditional* operations.

Claim 5.1. *Let r be an execution of the coordinated-collect algorithm. Then each of the value transitions of the curr_index variable in r, can be categorized as having one of the following forms*:
1. [*value*,`fill`] *to* [*value*,`advance`],
2. [*value*,`advance`] *to* [*value*+1,`fill`],
3. [*value*,`advance`] *to* [*value*+1,`advance`].

Claim 5.2. *Let r be an execution of the coordinated-collect algorithm. Then each of the value transitions of the* `curr_seq` *variable in r has the form: x to x* + 1.

Definition 5.2. For every execution and every value x, if $SC(curr\_index, [x, fill])$ and $SC(curr\_index, [x, advance])$ are contained in that execution, we define the *filling interval of x* denoted I_x, as the (unique) subsequence in the execution starting with a $SC(curr\_index, [x, fill])$ and ending with $SC(curr\_index, [x, advance])$ if $SC(curr\_index, [x, advance])$ exists. We call such filling intervals *closed*. If $SC(curr\_index, [x, advance])$ does not exist, I_x extends to the end of the execution. In that case we say that I_x is an *open* filling interval. Sometimes we use the term *interval* as shorthand for *filling interval*.

An execution may thus contain many closed intervals and its last interval can be either closed or open. Note that from our definition, an execution is not "covered" by intervals, rather, they are scattered throughout it.

Definition 5.3. Given an interval I_x, we denote $SC(curr\_index, [x, fill])$ (the starting event of I_x) by $Start_x$. If I_x is a closed interval we denote $SC(curr\_index, [x, advance])$ (the ending event of I_x) by End_x.

From the above definitions and Claim 5.1 we can deduce:

Claim 5.3. *If two intervals I_x and I_y are included in an execution r, then I_x and I_y do not overlap, and if $x < y$ then $I_x \rightarrow I_y$.*

In the following lemma, we show that during an interval I_x there is only one view on which scanners may potentially operate.

Lemma 5.4. *For every execution r and every interval I_x contained in r, the following properties are satisfied*:

(a) *Content*$(r, Start_x, \texttt{A}[x \bmod n]) \neq \texttt{Null}$.

(b) *During I_x, the content of* $\texttt{A}[x \bmod n]$ *changes at most once, and to* Null. *If I_x is a closed interval then* $\texttt{A}[x \bmod n]$ *is updated to* Null *during I_x.*

Proof. We show that the properties hold for every interval I_x contained in the execution by induction on x. Assume that the properties hold for all the intervals I_y where $y<x$, and let us prove that they hold for I_x.

Proof. (a) Assume by way of contradiction that the property does not hold. By the algorithm the sequence of operations executed by processor p which has set *curr_index* to [x,fill] is the following:

$$LL^p(curr\_index, [x-1, advance]) \rightarrow R^p(A[x, \bmod n], A[x, \bmod n] \neq Null) \rightarrow SC^p(curr\_index, [x, fill]).$$

In that case, there is some processor q that updated $\texttt{A}[x \bmod n]$ to Null after R^p ($\texttt{A}[x \bmod n]$, $\texttt{A}[x \bmod n] \neq \texttt{Null}$) and before $SC^p(curr\_index, [x, fill])$. Since by the algorithm, $\texttt{A}[x \bmod n]$ is set to Null only by processes executing procedure Help_fill the sequence of events executed by q (lines 32 and 33) must be

$$LL^q(A[y \bmod n], V) \rightarrow R^q(curr\_index, index) \rightarrow SC^q(A[y \bmod n], Null)$$

for some y s.t. $y \bmod n = x \bmod n$ and some view V. Since $SC^q(\texttt{A}[y \bmod n], \texttt{Null}) \rightarrow Start_x$ and since q has read *curr_index* before $Start_x$, by Claim 5.1, $y<x$. Now, since q has executed lines 10–12 in Scan before calling Help_fill, index = [y,fill]. Therefore by Claim 5.1, the only possible interleaving between p and q's events is

$$LL^q(A[y \bmod n], V) \rightarrow R^q(curr\_index, [y\, fill]) \rightarrow LL^p(curr\_index, [x-1, advance]) \rightarrow R^p(A[y \bmod n] \neq Null) \rightarrow SC^q(A[y \bmod n], Null).$$

Clearly, I_y is a closed interval, and $Content(r, End_y, \texttt{A}[y \bmod n]) = \texttt{Null}$) by the induction hypothesis on property (b). Since by Claim 5.1 $End_y \rightarrow LL^p(curr\_index, [x-1, advance])$, it follows by the specification of the *Store-conditional* operation, that $SC^q(\texttt{A}[y \bmod n], \texttt{Null})$ should have failed, a contradiction.

(b) We show that (1) during I_x no processor ever writes in $\texttt{A}[x \bmod n]$ a value that differs from Null, and (2) if I_x is a closed interval there is some processor which updates $\texttt{A}[x \bmod n]$ to Null during I_x. To prove (1), assume by way of contradiction

that some processor writes a non-Null value in A[$x \bmod n$] during I_x. Since by the algorithm the only processor which may write into A[$x \bmod n$] a value which differs from Null is processor $o = x \bmod n$, o has executed (lines 2, 4 and 8):

$$R^o(A[o], Null) \rightarrow R^o(curr\_index, curr\_index\_value \bmod n \neq o \vee curr\_index\_state \neq fill) \rightarrow W^o(A[o], V')$$

for some view V'. Since we assume that $W^o(A[o], V')$ occurs during I_x and since by Property (a), at $Start_x$, A[$x \bmod n$] $\neq$ Null we may deduce that

$$Start_x \rightarrow R^o(A[o], Null) \rightarrow R^o(curr\_index, curr\_index.value \bmod n \neq o \vee curr\_index.state \neq fill) \rightarrow W^o(A[o], V') \rightarrow End_x.$$

However, by Claim 5.1 and by Definition 5.2 during I_x, curr_index $= [x, fill]$ and therefore $R^o(curr\_index, curr\_index.value \bmod n \neq o \vee curr\_index.state \neq$ fill) cannot occur – a contradiction. To prove part (2), let u be the processor that performed $SC(curr\_index, [x, advance])$. By Claim 5.1 and by the algorithm (lines 10 and 15), u executed:

$$Start_x \rightarrow LL^u(curr\_index, [x, fill]) \rightarrow R^u(A[x \bmod n], Null) \rightarrow SC^u(curr\_index, [x, advance]).$$

By this equation and since by property (a) at $Start_x$, A[$x \bmod n$] $\neq$ Null, the claim holds. □

Definition 5.4. An interval I_x during which A[$x \bmod n$] changes to Null, is a *completed* interval.

Clearly, by Lemma 5.4 any closed interval is a completed interval.

Definition 5.5. For every completed interval I_x, we denote by $Done_x$ the event in I_x, in which by the previous lemma, A[$x \bmod n$] changes to Null. For a given execution r and a processor o, we denote all the intervals I_x s.t. $x \bmod n = o$ as *o's intervals*. We also denote all the write events performed by o during the calls to procedure Init as o's *initialization events*.

Clearly $Done_x \rightarrow End_x$, since by Lemma 5.4 A[$x \bmod n$] changes to Null *during* I_x.

Lemma 5.5. *Let r be an execution. If e is an event in r of the form* $SC(A[o], \text{Null})$, *then e is contained in one of o's intervals.*

Proof. Consider p, the processor that performed e. By the algorithm p executed (lines 10, 32 and 33):

$$LL^p(curr\_index, [x, fill]) \rightarrow LL^p(A[x \bmod n], A[x \bmod n] \neq Null) \rightarrow R^p(curr\_index, [x, fill]) \rightarrow e,$$

where $x \bmod n = o$. By Claim 5.1 $Start_x \rightarrow R^p(\texttt{curr\_index}, [x, fill])$. I_x is either open, in which case it extends to the end of the execution, or it is closed, and then by Lemma 5.4, at End_x $\texttt{A}[x \bmod n] = \texttt{Null}$. It follows that $e \rightarrow End_x$, otherwise $SC(A[o], \texttt{Null})$ would have failed. □

Lemma 5.6. *Given an execution r and a processor o, no initialization event of o is contained in any of o's intervals.*

Proof. Assume by way of contradiction that some initialization event e of processor o is contained in one of o's intervals I_x. Since the `Init` procedure is called only at line 7 of the algorithm, o executed (lines 2–8):

$$R^o(A[o], Null) \rightarrow R^o(curr\_index, curr\_index.value \bmod n \neq o \vee curr\_index.state \neq fill) \rightarrow e.$$

By the assumption $Start_x \rightarrow e$. Since by the definition of o's intervals $x \bmod n = o$, and since only processor o may write non-`Null` values into $\texttt{A}[o]$ (line 8), by the first part of Lemma 5.4, $Start_x \rightarrow R^o(A[o], \texttt{Null})$. In such case, by Claim 5.1 and Definition 5.2,

$$R^o(curr\_index, curr\_index.value \bmod n \neq o \vee curr\_index.state \neq fill)$$

may occur only after End_x, a contradiction. □

Following Lemma 5.4, for every interval I_x:

Definition 5.6. Let I_x's *view* be defined to be the *only* non-`Null` value which was in $\texttt{A}[x \bmod n]$ during I_x.

Definition 5.7. Since by the algorithm (lines 6–8), $\texttt{A}[o]$ contains either $2o$ or $2o+1$ and since only o's initialization events may write to $\texttt{view}[V]\texttt{.version}$, by Lemma 5.6 the value contained in $\texttt{view}[V]\texttt{.version}$ where V (I_x's view), remains unchanged during I_x. This value is I_x's *version* and we denote it as ver_x.

Definition 5.8. We say that a processor executing `Help_fill` *operates* on view V with version *ver* if the parameter `x` passed to the `Help_fill` call contains $= V$ and the value read $\texttt{view}[V]\texttt{.version}$ at line 22 is equal to *ver*.

For every interval I_x, we say that a processor p, executing lines 24–35 in `Help_fill` is helping I_x if the `index`

parameter passed to `Help_fill` call contains $[x, fill]$.

All the events performed by processors while helping I_x are the *helping events* of I_x.

Lemma 5.7. *Every processor helping I_x operates on I_x's view with version ver_x.*

Proof. Every helping processor p of I_x performed the following sequence of operations during the `Scan` (lines 10 and 11) and `Help_fill` (lines 22 and 23):

$$LL^p(curr\_index, [x, fill]) \to R^p(A[x \bmod n], V)$$
$$\to R^p(views[V].version, ver)$$
$$\to R^p(curr\_index, [x, fill]),$$

for some version *ver* and view $V \neq$ `Null`. By Claim 5.1 this whole sequence occurred within I_x. Therefore, by Lemma 5.4 and Definitions 5.7 and 5.8, V and *ver* are I_x's view and version, respectively. □

In the next lemma we show that every completed interval has at least one processor helping it.

Lemma 5.8. *Given a completed interval I_x, let p be the processor that executed $Done_x$. Then p helps I_x and $Done_x$ is one of I_x's helping events.*

Proof. By the algorithm, p executed (lines 10, 11, 32 and 33)

$$R^p(curr\_index, [y, fill]) \to R^p(A[y \bmod n], V)$$
$$\to LL^p(A[y \bmod n], A[y \bmod n] \neq Null)$$
$$\to R^p(curr\_index, [y, fill])$$
$$\to SC^p(A[y \bmod n], Null)$$

for some $V \neq$ `Null` where $y \bmod n = x \bmod n$. By Lemma 5.1, $y \leqslant x$. If $y < x$ then I_y must be a closed interval and by Lemma 5.4 at $SC(curr\_index, [y, advance])$, `A[`$y \bmod n$`]` = `Null` and SC^p(`A[`$y \bmod n$`]`, `Null`) must have failed. Therefore we may deduce that $y = x$. □

Definition 5.9. A *virtual_fill* execution on a view V is a sequence of the form

$$LL^{init}(curr\_seq, seq) \to SC^{init'}(curr\_seq, seq + 1)$$
$$\to coordinated\_fill(V)$$

for some value *seq*, where $coordinated\_fill(V) = \{s_0, \ldots, s_{n-1}\}$, and where each s_i is a sequence of events of one of the forms

$$R^{k_i}(memory[i].high, val_i) \to LL^{k_i}(views[V].regs[i], Null) \to SC^{k_i}(views[V].regs[i], val_i.value),$$

where $val_i.seq < seq + 1$ or

$$R^{k_i}(memory[i].high, high) \to R^{k_i}(memory[i].low, val_i) \to LL^{k_i}(views[V].regs[i], Null) \to SC^{k_i}(views[V].regs[i], val_i.value),$$

where $high.seq = seq + 1$. Denote the vector $val_0.value, val_1.value, \ldots val_{n-1}.value$ as the *result* of the *virtual_fill* execution. Denote *init*, *init*′ and $k_0 \ldots k_{n-1}$ as the *participating* processors of the *virtual_fill* execution.

Definition 5.10. Given a completed interval I_x, let $SeqAgree_x$ be the first helping event of I_x in the execution, that was generated by the execution of line 25 in the `Help_fill` procedure. Let $NewSeq_x$ be the first helping event of I_x in the execution, that was generated by the execution of line 26 in the `Help_fill` procedure. From this definition and by Lemma 5.8, such events exist and

$$Start_x \to SeqAgree_x \to NewSeq_x \to Done_x.$$

Notice that in the above definition we use the notation $SC^{init'}$ even though it may be the same process *init* that performed the operation. We did this in order to avoid the need to prove that the same processor performed both operations.

We now proceed to show that every completed interval contains a *virtual_fill* execution.

Lemma 5.9. *For every execution r, if I_x is a completed interval contained in r and V is I_x's view, then the following properties hold*:

(a) $Content(r, Start_x, l) =$ `Null` *for every location l in* `views[V].regs` *or* `views[V].sequence`.
(b) $Content(r, SeqAgree_x,$ `views[V].sequence`$) = Content(r, Start_x,$ `curr_seq`$) + 1$ *and no helping event of I_x updates* `views[V].sequence` *after $SeqAgree_x$.*
(c) $Content(r, NewSeq_x,$ `curr_seq`$) = Content(r, Start_x,$ `curr_seq`$) + 1$ *and none of the helping events of I_x update curr_seq after $NewSeq_x$.*
(d) $Content(r, Done_x,$ `views[V].regs[`i`]`$) \neq$ `Null` *for all $i = 0, \ldots, n-1$, and no helping event of I_x updates* `views[V].regs[`i`]` *after Done.*

At this point we remind the reader that an execution may contain at most one open interval, and that this interval extends up to the end of the execution.

Proof. By induction on x. Assume that properties (a)–(d) are satisfied for all the intervals I_y $y < x$ and let us show that they hold for I_x:

Proof. (a) By Lemma 5.4 and Definition 5.6, $Content(r, Start_x, \mathtt{A}[x \bmod n]) = V$. Let $o = x \bmod n$. By the algorithm, processor o initialized all the fields in `views[V]` before writing V in `A[`$o \bmod n$`]`. By Lemma 5.6 the initialization events did not occur during any of o's intervals. Since by the induction hypothesis, the helping events of previous intervals do not update the views outside the intervals, and since `views[V].regs` can be modified only by helping or initialization events, `views[V]` remains unchanged until $Start_x$.

(b) Consider processor p which executed $SeqAgree_x$. By the algorithm, p either read `views[V].sequence` as $\neq$ `Null` or tried to update it. However, since by property (a) at $Start_x$ `views[V].sequence` = `Null`, and since by the induction hypothesis it will remain so until one of the helping events of I_x updates it, there is one helping processor of I_x that executed $LL(\mathtt{views[}V\mathtt{].sequence}, \mathtt{Null}) \rightarrow R(\mathtt{curr\_seq}, seq) \rightarrow SC(\mathtt{views[}V\mathtt{].sequence}, seq + 1)$ before $SeqAgree_x$. By the induction hypothesis, `curr_seq` is updated only by the helping events of I_x during I_x, and therefore it will remain unchanged at least until $SeqAgree_x$. Therefore,

$$Content(r, SeqAgree_x, views[V].sequence) = Content(r, Start_x, curr\_seq) + 1.$$

Assume by way of contradiction that some helping event e of I_x updated `views[V].sequence` after $SeqAgree_x$. By the algorithm the processor q that performed e executed:

$$LL^q(views[V].sequence, Null) \rightarrow R^q(views[V].version, ver) \rightarrow SC^q(views[V].sequence, *),$$

where by Lemma 5.7, $ver = ver_x$. Clearly $SeqAgree_x \rightarrow LL(\mathtt{views[}V\mathtt{].sequence}, \mathtt{Null})$, otherwise $SC(\mathtt{views[}V\mathtt{].sequence}, *)$ would have failed by the first part of property (b). That means, that `view[V].sequence` was reinitialized by $o = x \bmod n$ after $SeqAgree_x$. By the algorithm, o also incremented `views[V].` version before initializing `views[`V *mbox*`].`*sequence*. In that case $R^q(\mathtt{views[}V\mathtt{].\ version}, ver_x)$ could not have occurred, a contradiction.

(c) We first show that one of the helping events updated `curr_seq` before $NewSeq_x$. Let p' be the processor that executed $NewSeq_x$. Assume that p' did not update `curr_seq` before $NewSeq_x$. By the algorithm (line 25 in the `Help_fill` procedure) this may happen for one of the three following reasons: (1) p' read a value in `curr_seq` that differed from the value it has read in `views[V].sequence` minus one, or (2) p' read a value in `views[V].version` that differs from the content of its private variable `version`, or (3) p' failed in the $SC(\mathtt{curr\_seq}, seq)$ operation.

Case (1): By property (b), `views[V].sequence` remained unchanged from *Seq Agree*$_x$ until at least $Done_x$. Therefore `curr_seq` was updated after $SeqAgree_x$. By the induction hypothesis, during I_x `curr_seq` can be updated only by I_x's helping events.

Case (2): Could not happen since p' is the processor that executed $NewSeq_x$, and since by Lemma 5.6 the `version` field stays unchanged during intervals.

Case (3): If the *Store-conditional* operation failed that means that `curr_seq` was updated following the *Load-linked* operation on `curr_seq`, and by the induction hypothesis, `curr_seq` could be updated only by one of the helping events of I_x.

Let q be the processor that performed the helping event that updated `curr_seq` while according to the algorithm, q performed $R(\texttt{views}[V].\texttt{sequence}, seq) \to SC(\texttt{curr\_seq}, seq)$, after $SeqAgree_x$ and before $NewSeq_x$. Therefore, by the previous property $Content(r, NewSeq_x, \texttt{curr\_seq}) = Content(r, Start_x, \texttt{curr\_seq}) + 1$.

Assume that some processor q' while helping I_x updated $curr\_seq$ after $NewSeq_x$. Processor q' must have executed

$$\begin{aligned} R(views[V].sequence, seq') &\to LL(curr\_seq, seq' - 1) \\ &\to R(views[V].version, ver_x) \\ &\to SC(curr\_seq, seq'). \end{aligned}$$

By Claim 5.2 this could have happened only if V was reinitialized and reused after $NewSeq_x$. However in that case $\texttt{views}[V].\texttt{version}$ must have been incremented and we get a contradiction since $R(\texttt{views}[V].\texttt{version}, ver_x)$ could not have occured.

(d) Consider p'', the processor that executed $Done_x$. By the algorithm, p'' has either read $\texttt{views}[V].\texttt{regs}[i]$ as $\neq \texttt{Null}$ or tried to update it. However, since by property (a) at $Start_x$, $\texttt{views}[V].\texttt{regs}[i] = \texttt{Null}$, and since by the induction hypothesis it remained so until one of the helping events of I_x updated it, there is a helping event of I_x that updated $\texttt{views}[V].\texttt{regs}[i]$ before $Done_x$.

The proof that none of the helping events of I_x will update $\texttt{views}[V].regs[i]$ from this point on continues as for property (b) above: assume by way of contradiction that some processor q helping I_x updated $\texttt{views}[V].reg[i]$ for some $0 \neq i \neq n-1$ after $Done_x$. By the algorithm q executed:

$$\begin{aligned} LL^q(views[V].reg[i], Null) &\to R^q(views[V].version, ver) \\ &\to SC^q(views[V].reg[i], v) \end{aligned}$$

for some value v where by Lemma 5.7, $ver = ver_x$. Clearly $Done_x \to LL(\texttt{views}[V].\texttt{reg}[i], \texttt{Null})$, otherwise $SC(\texttt{views}[V].\texttt{reg}[i], v)$ would have failed by the first part of property (d). That means, that $\texttt{view}[V].\texttt{reg}[i]$ was reinitialized by $o = x \bmod n$ after $Done_x$. By the algorithm, o also incremented $\texttt{views}[V].\texttt{version}$ before initializing $\texttt{views}[V].\texttt{sequence}$. In that case $R^q(\texttt{views}[V].\texttt{version}, ver_x)$ could not have occurred, a contradiction. □

Corollary 5.10. *Given an execution r and an event e in r of the form $SC(l, val)$ where l is one of the fields in* `view[V].regs` *or* `view[V].sequence` *for some view*

V, or `curr_seq` *then if e is contained in some interval I_x, e is one of the helping events of I_x.*

Note that we do not care about the case where e is not contained in an interval since as we will see later, in this case it cannot be included in the view returned by a scan.

Proof. According to the algorithm a processor may perform e only while helping some interval I_y (lines 24, 25, 30 and 31), where $[y, fill]$ is the value read by the processor in line 10 of the algorithm. If $y \neq x$, then by Claim 5.1 $y < x$ and by Lemma 5.9(d), we have a contradiction. Therefore $x = y$ and by the Definition 5.8, e is a helping event of I_x. □

Corollary 5.11. *Given a completed interval I_x,* `curr_seq`, `views[`V`].sequence` *and all the fields of* `views[`V`].reg` *where V is I_x's view, are updated during I_x exactly once and by the helping events of I_x.*

Proof. According to Corollary 5.10 we may assume that only helping events of I_x update those fields during I_x. By Lemma 5.9(a), (b) and (c), `views[`V`].sequence` and all the fields of `views[`V`].reg` change from a `Null` to a non-`Null` value during I_x. By the algorithm (lines 24, 25, 28, and 29) the transitions of the values stored in `views[`V`].sequence` and all the fields of `views[`V`].reg` are of the form `Null` to non-`Null`. Since by Lemma 5.6, those fields could not be initialized during I_x, they will be updated exactly once. According to Lemma 5.9(c), `curr_seq` is updated at least once during I_x. Since by the algorithm (line 25) all changes on `curr_seq` are of the form x to $x+1$, it follows that `curr_seq` will be updated exactly once. □

Lemma 5.12. *Every completed interval I_x contains exactly one virtual_fill execution on I_x's view.*

Proof. Given a completed interval I_x, let V be I_x's view. By Corollary 5.11 `curr_seq`, `views[`V`].sequence` and all the fields of `views[`V`].reg`, are updated during I_x exactly once and by the helping events of I_x. According to Definition 5.9, we construct a *virtual_fill* execution in the following way: choose *init* to be the processor that while helping I_x updated `views[`V`].sequence`. Choose *init'* to be the processor that while helping I_x updated `curr_seq`. Clearly by Lemma 5.9(b) and (c), *init* and *init'* performed $LL^{init}(curr\_seq, seq)$ and $SC^{init'}(curr\_seq, seq+1)$ for some value *seq*, respectively. Now, for every $i = 0 \ldots n-1$, choose k_i to be the processor that while helping I_x updated `views[`V`].regs[`i`]`. By Lemma 5.9 all those processors read $seq+1$ in `views[`V`].sequence` while executing line 29 in the algorithm. For that reason and according to lines 29–31 in the algorithm, for every $i = 0, \ldots, n-1$, k_i has performed

$$R^{k_i}(memory[i].high, val_i) \rightarrow LL^{k_i}(views[V].regs[i], Null) \rightarrow SC^{k_i}(views[V].regs[i], val_i.value),$$

where $val_i.seq < seq + 1$ or

$$R^{k_i}(memory[i].high, high) \rightarrow R^{k_i}(memory[i].low, val_i) \rightarrow LL^{k_i}(views[V].regs[i], Null) \rightarrow SC^{k_i}(views[V].regs[i], val_i.value),$$

where $high.seq = seq + 1$. Therefore I_x contains at least one *virtual_fill* execution. Now, by Corollary 5.10, during I_x, any $SC^p(l, val)$ event where l is one of the `sequence` or `reg` fields of a view, must be one of the helping events of I_x. Consequently, by Lemma 5.7, any such event operates on V. By Corollary 5.11, I_x contains at least one *virtual_fill* execution. □

Definition 5.11. Given a completed interval I_x we denote the *virtual_fill* execution that occurred during I_x as VF_x.

Corollary 5.13. *Given a completed interval I_x, let V be I_x's view and let $v_0, v_1, \ldots, v_{n-1}$ be the results of VF_x (see Definition 5.9), then for every $0 \leqslant i \leqslant n-1$,*

$$Content(r, Done_x, views[V].regs[i]) = v_i.$$

Proof. Follows directly from the construction of VF_x in Lemmas 5.12, 5.9 and Corollary 5.10. □

As all the intervals during the execution are totally ordered by $\rightarrow$, so are the virtual filling executions.

Definition 5.12. Given a sequence of events S, we denote by $OP_S(var,val)$ the event (assuming that it exists and that it is unique) of the form $OP(var,val)$ that occurs in S.

For example, for a *virtual_fill* execution VF_x, SC_{VF_x} (`curr_seq`, $*$) represents the $SC(curr\_seq, *)$ event contained in VF_x. We now proceed to prove that our algorithm meets property P1 (see Section 2.1). First, we show that there exists a linearization order between the update sequences and the *virtual_fill* sequences. Informally, an update operation will be linearized before the first *virtual_fill* which "sees" this update.

Definition 5.13. The linearization order of *update* and *virtual_fill* sequences denoted by ($\Rightarrow$) abides by the following rules:
1. $VF_x \Rightarrow VF_{x'}$ iff $VF_x \rightarrow VF_{x'}$.
2. $U^i \Rightarrow VF_x$ iff

$$R_{U^i}(curr\_seq, *) \rightarrow SC_{VF_x}\ (curr\_seq, *)$$

and

$$W_{U^i}(memory[i].high, *) \rightarrow R_{VF_x}(memory[i].high, *).$$

3. $U^i \Rightarrow U^j$ iff

$$R_{U^i}(curr\_seq, *) \rightarrow R_{U^j}(curr\_seq, *)$$

and there is no VF_x, such that

$$U^j \rightarrow VF_x \text{ and } VF_x \rightarrow U^i.$$

Lemma 5.14. *The linearization order* ($\Rightarrow$) *between update and virtual_fill sequences is consistent with the real time order* $\rightarrow$.

Proof. We must show that all the following hold:
1. If $VF_x \rightarrow VF_{x'}$ then $VF_x \Rightarrow VF_{x'}$.
2. If $U^i \rightarrow VF_x$ then $U^i \Rightarrow VF_x$.
3. If $VF_x \rightarrow U^i$ then $VF_x \Rightarrow U^i$.
4. If $U^i \rightarrow U^j$ then $U^i \Rightarrow U^j$.

Cases 1–3 follow immediately from Definition 5.13. To prove Case 4, assume by way of contradiction that for two updates U^i and U^j, $U^i \rightarrow U^j$ but $U^j \Rightarrow U^i$. By Definition 5.13, this may happen only if there is some VF_x such that $U^j \Rightarrow VF_x \Rightarrow U^i$. In that case, according to the same definition $U^j \Rightarrow VF_x$ implies that $R_{U^j}(\texttt{curr\_seq}, *) \rightarrow SC_{VF_x}$ $(\texttt{curr\_seq}, *)$. Since, by the hypothesis, $U^i \rightarrow U^j$, it follows that $R_{U^i}(\texttt{curr\_seq}, *) \rightarrow R_{U^j}(\texttt{curr\_seq}, *)$ and therefore $R_{U^i}(\texttt{curr\_seq}, *) \rightarrow SC_{VF_x}$ $(\texttt{curr\_seq}, *)$. Therefore, since $VF_x \Rightarrow U^i$, R_{VF_x} $(\texttt{memory}[i].\texttt{high}, *) \rightarrow W_{U^i}(\texttt{memory}[i].\texttt{high}, *)$. In that case, SC_{VF_x} $(\texttt{curr\_seq}, *) \rightarrow W_{U^i}(\texttt{memory}[i].\texttt{high}, *)$ and therefore SC_{VF_x} $(\texttt{curr\_seq}, *)$ $\rightarrow R_{U^j}(\texttt{curr\_seq}, *)$ and consequently $VF_x \Rightarrow U^j$ which is a contradiction. □

Lemma 5.15. *Assume that* p *is helping* I_x, *then while helping* I_x, p *executes line* 34 *only if* I_x *is a completed interval and only after* $Done_x$.

Proof. Assume by way of contradiction that p executes line 34 while helping I_x and either I_x is an interval that is not completed or p executes line 34 before $Done_x$. In both cases, p reached line 34 in `Help_fill` without updating $\texttt{A}[x \bmod n]$ to `Null`. By Lemma 5.7, this may happen only if while executing lines 32 and 33, p has either (1) read a value in $\texttt{A}[x \bmod n]$ that differs from I_x's view, or (2) read a value in `curr_index` that differs from $[x, \texttt{fill}]$, or (3) failed in the $SC^p(\texttt{A}[x \bmod n], \texttt{Null})$. By line 10 of the algorithm for a processor helping an interval to execute `Help_fill` with parameter `index` $[x, \texttt{fill}]$, it must have read in line 10 the value of $[x, \texttt{fill}]$ into `curr_index`. By Claim 5.1, p may read a value in `curr_index` that differs from $[x, \texttt{fill}]$ only after End_x, and consequently after $Done_x$. This precludes Case (2). By Lemma 5.4, during I_x $\texttt{A}[x \bmod n]$ contains one and only one value that differs from `Null`. Therefore, if p has read some value different from I_x's view, the value read by p must be `Null`, implying case (1) cannot hold. Therefore, p failed to update $\texttt{A}[x \bmod n]$ in the $SC^p(\texttt{A}[x \bmod n], \texttt{Null})$ operation. However, this may occur only if the content of $\texttt{A}[x \bmod n]$ was changed before $SC^p(\texttt{A}[x \bmod n], \texttt{Null})$. Since by Lemma 5.4, $\texttt{A}[x \bmod n]$ was changed to `Null`, we have a contradiction. □

Lemma 5.16. *The value returned by a scan operation is the result of a virtual_fill which occurs within the scan execution interval.*

Proof. Assume that processor p performed a *Scan*(view[$0 \ldots n-1$]) operation. By the algorithm, p returned either (1) the contents of one of its views (line 9 in the `Scan` procedure) or (2) a copy of a view it helped to fill (line 14 in the `Scan` procedure). We will show that in both cases the claim holds.

Case (1): Processor p must have executed

$$R^p(A[p], Null) \rightarrow W^p(A[p], V) \rightarrow R^p(A[p], Null),$$

where V is one of p's views. In that case, there is an event $e = SC(A[p], \texttt{Null})$, such that $W(A[p], V) \rightarrow e \rightarrow R^p(A[p], \texttt{Null})$. By Lemma 5.5, e occurred within one of p's intervals, let us say I_x, which is obviously a completed interval. In that case, since only p writes non-`Null` values into `A[`p`]`, by Lemma 5.4 at $Start_x$, $A[p] \neq \texttt{Null}$, and therefore $W(A[p], V) \rightarrow Start_x \rightarrow Done_x \rightarrow R(A[p], \texttt{Null})$ and V is I_x's view. Finally, by Lemma 5.9, at $Done_x$, `views[`V`]` contained the result of VF_x. Since p is the owner of the view and only p may change non-`Null` values, the view will remain unchanged between $Done_x$ and p's read, implying the claim.

Case (2): By the algorithm (lines 34 and 35 in procedure `Help_fill`) this situation occurred after p helped two intervals, say I_x and $I_{x'}$. By Lemma 5.15, both intervals are completed and p started to collect the filling result (line 35) after $Done_x$ and $Done_{x'}$, respectively. Since at $Done_x$ `A[`$x \bmod n$`]` $=$ `Null` and since p must have performed $R($`A[`$x \bmod n$`]` $\neq$ `Null`$)$ before starting to help $I_{x'}$ we may conclude that $x \neq x'$. Therefore, p helped two different intervals. It follows from the sequence of events in the construction of a *virtual_fill* in the proof of Lemma 5.12, that $VF_{x'}$ occurred before $Done_{x'}$ and therefore $VF_{x'}$ occurred within the scan interval. By Lemma 5.7, processor p operated on $I_{x'}$'s view with $I_{x'}$'s version value.

Assume by way of contradiction, that the result of $VF_{x'}$ differs from the vector collected by p. According to Lemma 5.15 and Corollary 5.13, this may happen only if one of the locations in `views[`V`].regs` was updated after $Done_x$ and before p read it. In that case, by Corollary 5.10 there is some interval I_x'', where $x'' \bmod n = x' \bmod n$ and I_x' and I_x'' have the same view. Now, notice that processor $o = x' \bmod n$ first increases the version field before reinitializing and reusing a view, and p executed $R^p($`views[`V`].version`$, ver_x)$ after copying `views[`V`].regs`. It follows that by the algorithm p should have returned a `FAIL` value and could not have returned the content of `result` – a contradiction. □

Let the linearization order of a scan operation with respect to update operations be that of the *virtual_fill* execution whose result is returned by the scan. By Lemmas 5.14 and 5.16 it follows that:

Lemma 5.17. *The coordinated-collect multi-scanner algorithm meets property* (P1).

We now proceed to prove that our algorithm meets property (P2). Given an execution r let r' be the same sequence of events as r, preceded the by the update sequences

$$U^0(Empty), U^2(Empty) \ldots U^{n-1}(Empty).$$

These operations do not change the shared memory and do not affect the local state of the processors, and r' is a possible execution of the algorithm. Thus if r' is linearizable so is r. We henceforth assume executions have the form r', with "ghost" sequences at their beginning. This assumption simplifies the proof of property (P2) since it ensures that every scan operation has at least one update operation execution ordered before it.

Lemma 5.18. *Let VF_x be a virtual filling operation execution with result view$[0 \ldots n-1]$. For every $i = 0, \ldots, n-1$, if view$[i] = x$, the last update operation on register i, $U^i(val)$ which is linearized before VF_x, satisfies $val = x$.*

Proof. Consider the last $U^i(val)$ operation linearized before VF_x and assume by way of contradiction that $val \neq view[i]$. According to the definition of the linearization order $\Rightarrow$ (see Definition 5.13)

$$R_{U^i}(curr\_seq, seq) \rightarrow SC_{VF_x}(curr\_seq, new)$$

and

$$W_{U^i}(memory[i].high, [val, seq]) \rightarrow R_{VF_x}(memory[i].high, [val', seq']).$$

Now, if $[val', seq'] = [val, seq]$, by Definition 5.9 VF_x contains an event $SC(views[V].$ `regs[`i`],val`) and consequently $view[i] = val$, a contradiction. Therefore `memory[`i`].high` must have been updated before R_{VF_x} $(memory[i].high, [val', seq'])$. Consider U'^i the update operation that wrote $[val', seq']$ into `memory[`i`].high`. Since we assumed that U^i is the last update operation linearized before VF_x, it follows that $VF_x \Rightarrow U'^i$. According to Definition 5.13, either U'^i read $curr\_seq$ before it was updated by VF_x, or U'^i wrote to `memory[`i`].high` after VF_x read it. Since we assume that U'^i wrote to `memory[`i`].high` before it was read by VF_x, U'^i must contain $R_{U'_i}(curr\_seq, new)$ and consequently $seq' = new$. In that case, by Definition 5.9, VF_x contains:

$$R_{VF_x}(memory[i].high, val', seq') \rightarrow R_{VF_x}(memory[i].low, [val'', [seq''])$$
$$\rightarrow SC(view[V].regs[i], val'')$$

implying that the value collected for entry i during VF_x was the value stored in i's low register. We will show now that $[seq''', val'''] = [val, seq]$.

By the code of the `Update` procedure, U'^i contains

$$R_{U'_i}(curr\_seq, new) \rightarrow R_{U'_i}(memory[i].high, [val, seq])$$
$$\rightarrow W_{U'_i}(memory[i].low, [val, seq])$$
$$\rightarrow W_{U'_i}(memory[i].high, [val', new]).$$

Since

$$W_{U_i'}(memory[i].high, [val', new]) \rightarrow R_{VF_x}(memory[i].high, [val', seq']),$$

it follows that

$$W_{U_i'}(memory[i].low, [val, seq]) \rightarrow R_{VF_x}(memory[i].high, [val', seq']) \rightarrow R_{VF_x}(memory[i].low, [val'', seq'']).$$

By the `Update` algorithm, any update operation U_i' that occurs after U'^i and that read `curr_seq` before the end of VF_x will not modify `memory[i].low`, and therefore the claim holds. □

From Lemmas 5.16 and 5.18 we have:

Lemma 5.19. *The coordinated-collect multi-scanner algorithm meets property* (P2).

We now proceed to prove that the *coordinated-collect* is *wait_free*. We first make a distinction between the various situations that may cause a processor to exit the *Help_fill* procedure without being able to return a scan result. We say that a *non-participating failure* occurred whenever a processor does not enter the `Help_fill` procedure after reading `curr_index` (lines 15–18 in `Scan` procedure), or after it returns from `Help_fill` with a `Fail` status before participating in the collect process (line 23 in `Help_fill`). Otherwise, when a processor returns a `Fail` value from `Help_fill`, we say that a *participating failure* has occurred. The reader can easily convince herself that every non-participating failure takes O(1) machine instructions, while every participating failure takes O(n) machine instructions.

Lemma 5.20. *During a scan operation, the number of non-participating failures a processor may suffer from is at most* O(n) *and the number of participating failures it may suffer from is at most* O(1).

Proof. Assume that some processor p starts to execute `Scan`. We first show that after O(1) non-participating failures $A[p] \neq$ `Null`. If at the beginning of the execution $A[p] \neq$ `Null`, then we are done. Thus, assume that at the beginning of the execution $A[p] =$ `Null`. By the algorithm (line 4 in Fig. 5) p will not be able to post a new view in $A[p]$ only if $curr\_index.value \bmod n = i$ and $curr\_index.state =$ `fill` holds. However, by property **a** in Lemma 5.4, after one non-participating failure, this condition will not hold anymore at least until $A[p] \neq$ `Null`. Proving the first part of the claim is easy. Every time that a non-participating failure occurs, `curr_index`is advanced, and so a processor is ensured that after at most $O(n)$ non-participating failures `curr_index`points to its entry in `A`. Therefore, by Lemma 5.4, its entry in `A` must contain `Null` before the next advance of `curr_index`. Consequently, the scanner will exit and return its own view. Assume now, that a participating failure occurred during

processor p's scan operation. Processor p executed:

$$R^p(curr\_index,[x,fill]) \to R^p(\mathtt{A}[x \bmod n],V) \to R^p(views[V].version,ver)$$
$$\to R^p(curr\_index,[x,fill])$$
$$\to R^p(views[V].version,views[V].version \neq ver)$$

for some values x, ver and some view V. By the algorithm, only processor $o = x \bmod n$ may update `views[V].version`, and that only during the call to `init` function at line 7. Since every processor proposes its two views alternately (line 6 in the algorithm), in order to use view V twice o must have executed (lines 2–8):

$$R^o(A[o],Null) \to W^o(views[V].version,ver) \to W^o(A]o],V) \to R^o(A]o],Null)$$
$$\to R^o(curr\_index,curr\_index.value \bmod n \neq o$$
$$\vee curr\_index.state \neq fill)$$
$$\to W^o(views[V'].version,*) \to W^o(A[o],V') \to R^o(A[o],Null)$$
$$\to W^o(views[V].version,ver+1),$$

where V and V' are processor o's views. Since p must have executed $R^p(curr\_index, [x,fill]) \to R^p(\mathtt{A}[x \bmod n],V) \to R^p(curr\_index,[x,fill])$ we may deduce that $R^p(\mathtt{A}[x \bmod n],V) \to End_x$. Therefore, the interval between $R^p(\mathtt{A}[x \bmod n],V)$ and

$$R^p(views[V].version,views[V].version \neq ver_p)$$

overlaps at least a part of I_x and I_{x+n}. But since between two *virtual_fills* on views from the same `announce` entry (i.e. same processor) `curr_index` must have been advanced at least once over the entire `announce` array, p's view should have been filled during this interval. □

The following theorem is a direct corollary from Lemma 5.20.

Theorem 5.21. *In the coordinated-collect algorithm an Update operation takes* O(1) *machine instructions and a Scan takes* O(n) *machine instructions.*

A similar but simpler proof can establish the correctness of the single-scanner algorithm. On an intuitive level, since there is only a single scanner, by definition every scan operation contains a trivial *virtual_fill* operation, and all *virtual_fill* operations are serialized. Correctness will follow from Lemmas 5.17 and 5.19 by viewing every *Store-conditional* as a trivial write operation. Wait-freedom will follow trivially from the code in Fig. 1. We leave this to the interested reader.

Corollary 5.22. *The single-scanner algorithm of Fig. 1 is a wait-free implementation of a single-scanner atomic snapshot object.*

6. Performance evaluation of snapshot algorithms

We compared a collection of snapshot algorithms on a 64-processor simulated Alewife cache-coherent distributed-memory machine [5] developed by Agrawal et al. using the Proteus simulator developed by Brewer et al. [17][4]. Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not support complete simulation of the hardware. Instead, operations which are local (i.e. do not interact with the parallel environment) are executed uninterruptedly on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time. Proteus makes sure a thread can only see global events within the scope of its local time. Since actual machine instructions are counted for local operations, the quality of the code used to implement algorithms under Proteus can play an important part in determining the running time of the entire application. Though the simulator allows the user to determine the relative weight of local operations, we used the simulator's default costs which are derived from the Alewife machine.

In our simulations each processor had a cache with 2048 lines of 8 bytes and a memory access cost of 4 cycles. The cost of switching or wiring in the Alewife architecture was 1 cycle/packet. The current version of Proteus does not support *Load-linked*/*Store-conditional* instructions. Instead we used a slightly modified version that supports a 64-bit *Compare-and-Swap* operation where 32 bits serve as a time stamp. Naturally, this operation is less efficient than the theoretically accepted *Load-linked*/*Store-conditional* [25] (which we could have built directly into Proteus), since a failing *Compare-and-Swap* will cost a memory access while a failing *Store-conditional* will not. However, we believe the 64-bit *Compare-and-Swap* is closer to the real world than the theoretical *Load-linked*/*Store-conditional* since existing implementations of *Load-linked*/*Store-conditional* as on Alpha [19] or PowerPC [28] do not allow shared memory locations to be accessed between the *Load-linked* and the *Store-conditional* operations. On existing machines, the 64 bit compare-and-swap may be implemented by using a 64 bit *Load-linked*/*Store-conditional* as on the Alpha [33].

For each scan and update implementation we measured:

Throughput: The total number of completed operations by all the processors in the system running for 10^6 cycles.

[4] Version 3.00, dated February 18, 1993.

```
SCANNER:
                while (current_time < MAX_TIME) {
                        repeat random(scan_wait) times
                                /* do nothing */ ;
                        scan();
                }

UPDATER:
                while (current_time < MAX_TIME) {
                        repeat random(update_wait) times
                                /* do nothing */ ;
                        update(val);
                }
```

Fig. 7. Benchmarks.

Latency: The average amount of time between the start and the end of an operation for all the processors in the system.

We present the results of evaluating the algorithms in executions where each scanner/updater processor executes scan/update operations repeatedly (Fig. 7). Between any two operations, a processor waits for an amount of time chosen uniformly at random in the interval 0 to `scan_wait` for a scanner, and 0 to `update_wait` for an updater. We used the following synthetic benchmarks:

Checkpoint: The system has only one processor which executes scan operations (scanner) and the other processors execute update operations (updaters). This benchmark models the behavior of a "checkpoint" mechanism for collecting consistent backups of a multiprocessor system or for concurrent debugging. The results we present were tested with `MAX_TIME` equal to 10^6 cycles, and `scan_wait` and `update_wait` equal to 10^3 cycles.

Concurrent data structure: The system has half of the processors execute scans and the other execute updates. Though this is a somewhat arbitrary choice, we feel it is representative of possible use of snapshots for concurrent-data-structure design, where multiple processors update or request an atomic view of the state of the shared object. The presented results are algorithms tested with `MAX_TIME` equal to 10^6 cycles, and `scan_wait` and `update_wait` equal to 10^3 cycles. We ran several other sets of tests, among them ones with `update_wait` equal to 100 cycles in order to simulate a "heavy load" of updaters but choose not to present them since we noted no significant differences in the relative performance of the tested algorithms.

6.1. The algorithms

In the checkpoint benchmark, we tested the single-scanner algorithm (denoted *Single* in the graphs), as described in Section 3. In the concurrent data structure benchmark we tested the *coordinated-collect* multi-scanner algorithm (denoted as *coordinated*).

We compared our algorithms, with the following previously known snapshot algorithms:

A+: The unbounded sequence number version of algorithm of the Afek et al. [2], which has $O(n^2)$ scan and update complexity, and uses $O(n)$-valued registers. Each register in this version consists of *data*, *seq*, and n-valued *view* components. This algorithm has a bounded space version but we use the unbounded sequence number one for simplicity and to improve performance. Since we could not implement $O(n)$ values registers, and since this algorithm's flow control is not dependent on the contents of the n-valued *view* component of each register, we did not implement that component of the register.

AR: The unbounded space version of the sophisticated Attiya and Rachman algorithm [13], which has $O(n \log n)$ scan and update complexity and uses registers holding $O(n^2)$ values simultaneously, was simulated. Each register, which participates in the *lattice agreement* procedure, contains n vectors, where each vector represents a view of $O(n)$ values. The algorithm repeatedly creates a single new view vector using fields from two given view vectors. Since the vector elements values are not used by the algorithm, we used a bit for each value instead of the complete n-valued vector.

CD: The unbounded space version of the efficient Chandra and Dwork [18] algorithm, which has a scan complexity of $O(n)$ and update complexity of $O(n + C(n))$, where $C(n)$ is the consensus complexity. We used a *Load-linked/Store-conditional* primitive to implement the consensus primitive achieving scan and update complexity of $O(n)$. However, this algorithm uses registers holding $O(n)$ value simultaneously and atomic writes to these multiple locations. For each scanner the algorithm has new and old versions of an $O(n)$ value *view* register and an $O(1)$ value *time-stamp*. The control flow of the algorithm is dependent only on the values of the new and old *time-stamp* components, therefore we included only these components in our implementation's registers, without making the algorithm pay for the added $O(n)$ values that must be stored in other registers.

Lock-free: The simple algorithm in which a scanner repeatedly tries to perform a successful *double collect*, during which no change to memory occurred, and an updater which writes to its register in a straightforward manner. See details in [2].

Block-update: The scanner uses a multi-valued semaphore to "block" any updaters from performing a write to any of the registers, while it collects their values. The updaters use random backoff to control contention while "waiting" for the semaphore to be cleared.

Our implementations of AR and CD use unbounded space (unbounded in the strong sense, i.e an unbounded number of new register locations), and therefore their performance is guaranteed to be better than any of appropriate bounded implementation of the algorithms. Our benchmarks also make the realistic assumption that the implementation of registers containing $\Omega(k)$ values requires at least k local steps for each read operation. We avoid making this assumption for write operations, since a write could involve a change to only few of the n locations written. In summary, an n-valued read costs n local operations while an n-valued write costs only $O(1)$ local operations. As

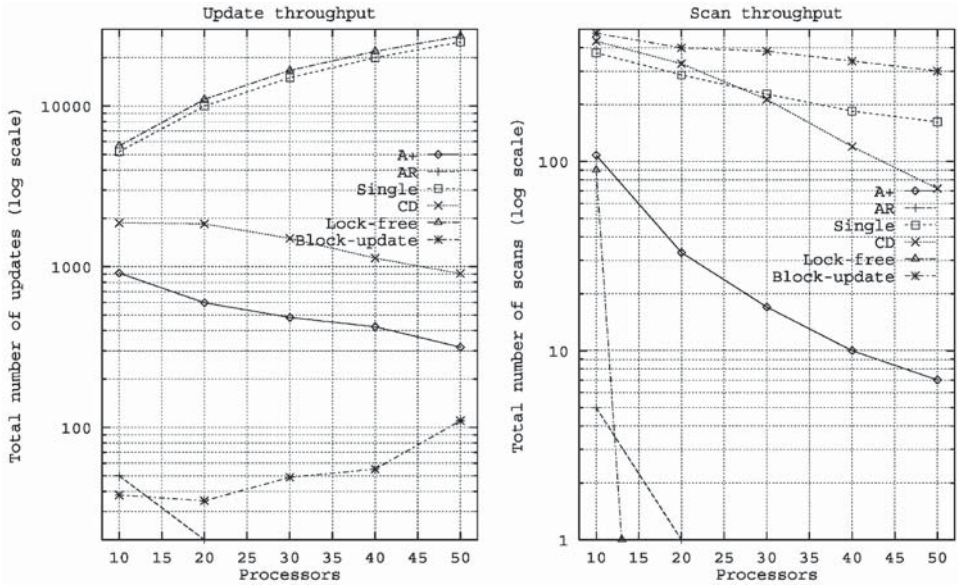

Fig. 8. Checkpoint benchmark throughput (Log scale).

will be seen in Section 6.4, we also performed tests under the unrealistic assumption of availability of an O(1) cost atomic O(n)-valued read, with no significant changes in our conclusions.

6.2. Checkpoint benchmark results

As expected, the checkpoint benchmark results (Fig. 8) show that the *block-update* and *lock-free* algorithms are at extreme ends with respect to their scan and update throughput. [5]

The *block-update* algorithm has the highest scan throughput. Most likely the main reason for this is that its scan operations are performed without any 'interference' from the updaters (interference is in terms of interconnect contention and cache misses that are known phenomena when running algorithms that access shared memory modules [22]). It has very low update throughput, since the updates can be executed only between scan operations. Nevertheless, there is a performance increase due to having more concurrent update attempts.

The *lock-free* algorithm has very poor scan throughput because of repeated double collect failures that increase with the number of updaters. On the other hand, its update throughput scales linearly with the number of updaters. This is clearly due to the small number of operations necessary to complete an update.

In the A+ algorithm, the throughput of the scan and update operations degrade similarly as the number of processors grows. Failures of its double collects increase

[5] The scan and update latency results are not presented since they are almost exactly inverse to the scan and update throughput.

with the number of updaters, increasing both its scan and update latency, (recall that its update procedure includes a scan). The AR algorithm, though asymptotically superior to A+, performs substantially worse than the A+ algorithm because of the constant overhead involved in each operation which is independent of the actual number of processors and also due to the additional cost of reading $O(n^2)$-valued registers. The AR algorithm does not manage to complete a single scan when the number of participating processes is greater than 20. Its update throughput scales poorly for the same reasons as in A+.

The CD algorithm has low update throughput which degrades moderately as the number of updaters grows, due to the $O(n)$ local work executed in each update operation and the additional cost of reading $O(n)$ valued registers. Its scan has good throughput for small numbers, but scales poorly since the increase in the number of updaters adds significantly to the network contention.

The single-scanner algorithm's update throughput is nearly the same as that of the *lock-free*'s due to the small number (four) of operations constituting an update. It also has a surprisingly high scan throughput, close to that of the non-interfering scan of the *block-update* algorithm, since its scan collects the updated values in a straightforward read sequence with no additional costs.

In conclusion, the key to an effective algorithm under the checkpoint benchmark is simplicity. Minimized coordination and dependency on information gathered and passed from one processor to another eliminates unnecessary computation and results in small constants and good performance.

6.3. Concurrent data structure benchmark results

The results of the concurrent data structure benchmark, appear in Fig. 9. For most of the tested algorithms these results have much in common with those of the checkpoint benchmark. We will therefore concentrate on the major differences.

The *block-update* algorithm does not seem to succeed in completing an update for concurrency levels of more than 10 processors due to the increased number of scanners which disable the updaters' progress. The scan throughput of the *lock-free* algorithm degrades rapidly due to the increased failure of the double collects as the number of updaters increases. The CD algorithm starts with a very good scan throughput and some scaling but as the updaters' 'interference' grows it degrades substantially.

The *coordinated-collect* algorithm maintains consistently high scan throughput and linear scaling of scan latency (the size of the view collected increases linearly with the number of processes). Unlike in the case of the checkpoint benchmark, there is an improved throughput since in many cases the returned result of several scanners is the same view.

6.4. Some final notes

We repeated our exact experiments without the added realistic cost of a multi-valued read, in an attempt to find out the effect of assuming the existence of a powerful

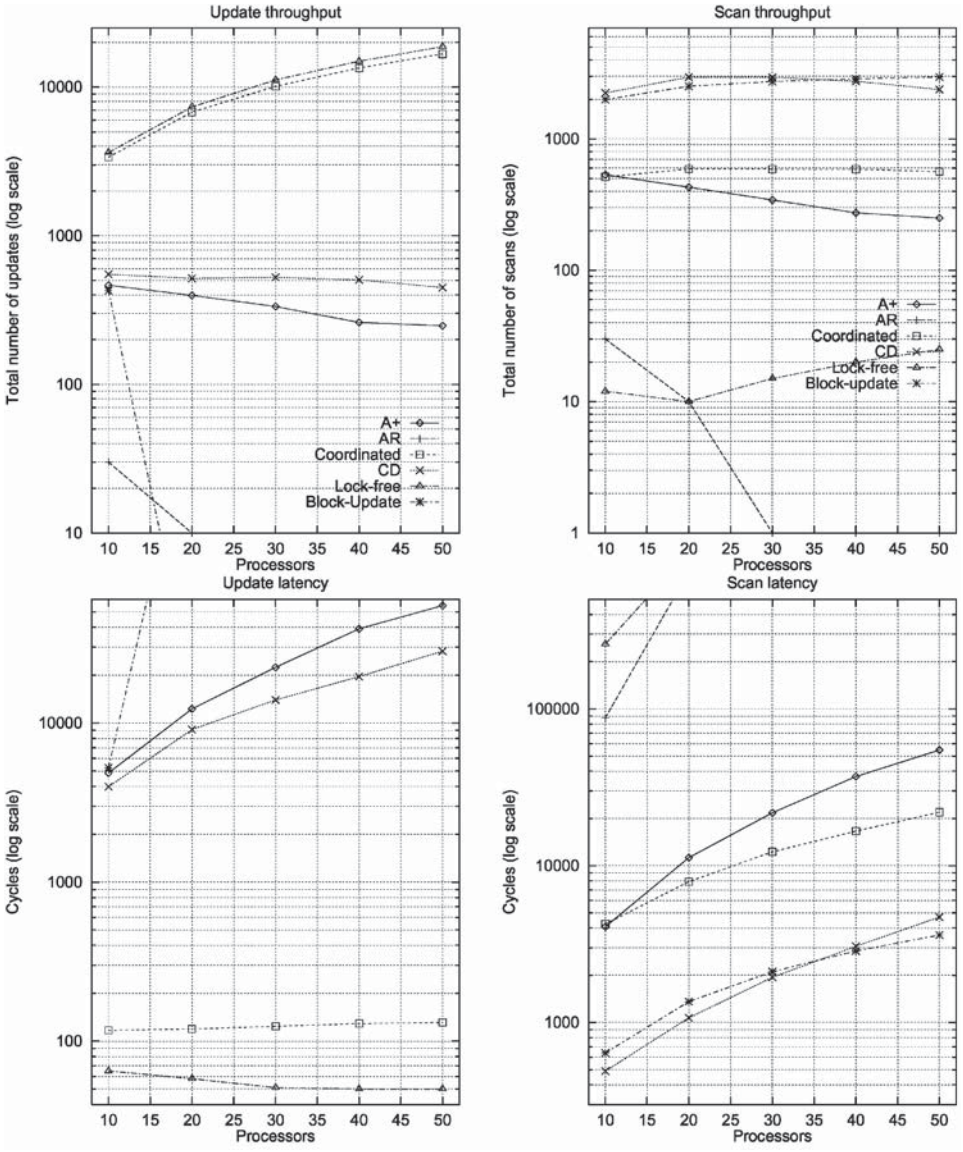

Fig. 9. Concurrent data structure results (Log scale).

multi-read operation on our experimental results. In our tests the CD algorithm [18] showed no significant improvement, most likely because the algorithm makes intensive use of atomically writing n values but little use of the expensive atomic n-value read. As can be seen when comparing the performance results for the checkpoint benchmark in Figs. 8 and 10, the A+ and the AR algorithms have higher scan and update throughput and less degradation when the n operation overhead is eliminated. However, this

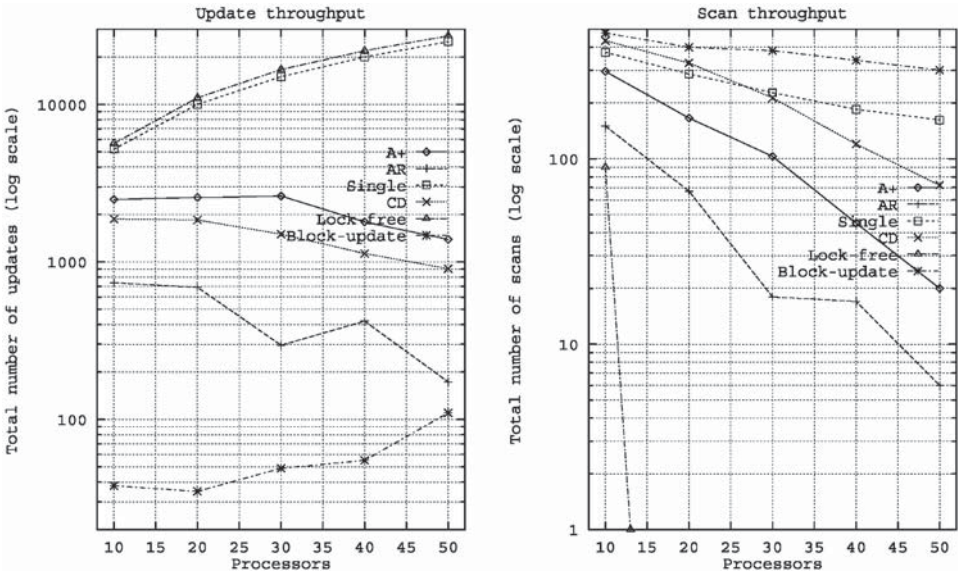

Fig. 10. Checkpoint throughput results – no additional cost (Log scale).

improvement does not significantly affect their performance with respect to the single-scanner algorithm, which remains substantially better in terms of scan and update throughput. For the concurrent data structure benchmark in Figs. 9 and 11, the A+ and AR algorithms show improved performance. The scan throughput of the A+ algorithm became better than the *coordinated* algorithm for small numbers of processors because of the structural overhead of the *coordinated* algorithm is relatively high, yet degrades as the number of processors grow. A similar improvement in the cost of updates is not sufficient to overcome the update throughput of the *coordinated* algorithm.

In conclusion, the cost of register operations is not a major performance factor in snapshot algorithms. The dominating factor with respect to algorithmic performance is the amount of cooperation among processors in collecting returned views, and the overhead associated with doing so.

7. Conclusions

This paper hopes to start researchers on the road to creating snapshot algorithms that will have practical appeal. Though the asymptotic complexity of our algorithms is optimal, there are quite a few directions in which their actual performance can be enhanced.

Enhancements would involve eliminating some of the constant overheads and making the algorithm's complexity more closely dependent on the actual number of scanners and updaters accessing it at a given time. Finally, the current trend towards running

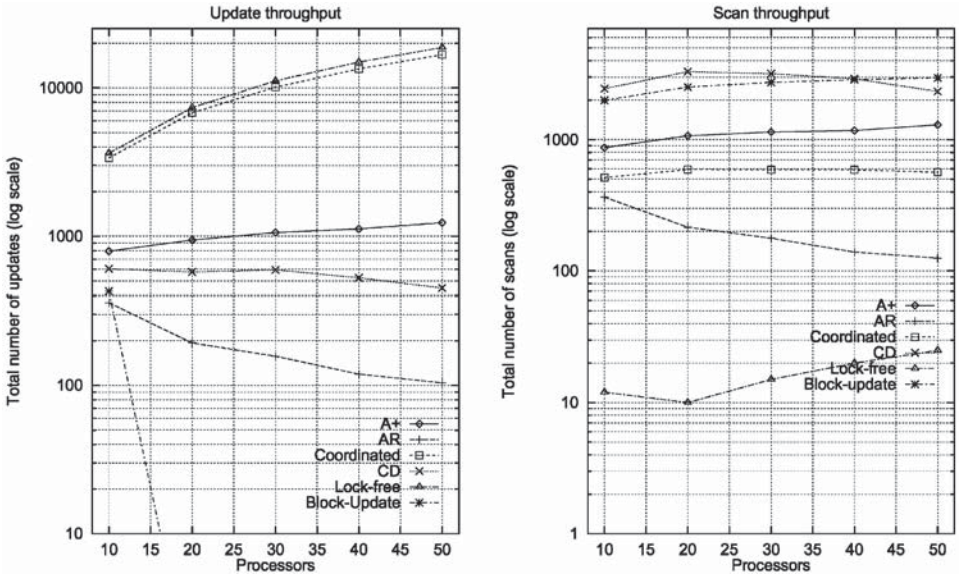

Fig. 11. Concurrent data structure results – no additional cost (Log scale).

multiprocessors applications in message passing architectures (farms of workstations) raises the interesting question of an efficient wait-free message passing implementation of an atomic snapshot object.

Acknowledgements

We wish to thank Yehuda Afek, Hagit Attiya, and the anonymous referees for the multitude of comments and observations they made throughout the writing of this paper.

References

[1] K. Abrahamson, On achieving consensus using a shared memory, Proc. 7th ACM Symp. on Principles of Distributed Computing, 1988, pp. 291–302.
[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, M. Shavit, Atomic snapshots of shared memory, Journal of the ACM 40 (4) (1993) 873–890.
[3] Y. Afek, D. Dauber, D. Touitou, Wait-free made fast, Proc. 27th ACM Symp. on Theory of Computing, May 1995, pp. 538–547.
[4] Y. Afek, E. Weisberger, The instancy of snapshots and commuting objects, Tel-Aviv University, June 1993.
[5] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, D. Nussbaum, The MIT Alewife machine: a large-scale distributed-memory multiprocessor. Scalable shared memory multiprocessors, Kluwer Academic Publishers, Dordrecht, 1991. Also as MIT Tech. Report MIT/LCS/TM-454, June 1991.
[6] J.H. Anderson, Multiple-writer composite registers, Distributed Comput. 7 (4) (1994) 175–195.
[7] J.H. Anderson, A.K. Singh, M.G. Gouda, The elusive atomic register revisited, Proc. 6th ACM Symp. on Principles of Distributed Computing, August 1987, pp. 206–221.

[8] J. Aspnes, M. Herlihy, Wait-free data structures in the asynchronous PRAM model, Proc. 2nd Ann. Symp. on Parallel Algorithms and Architectures, July 1990, pp. 340–349.
[9] J. Aspnes, M. Herlihy, N. Shavit, Counting networks, J. ACM 41 (5) (1994) 1020–1048.
[10] H. Attiya, D. Dolev, N. Shavit, Bounded polynomial randomized consensus, Proc. 8th ACM Symp. on Principles of Distributed Computing, August 1989, pp. 281–293.
[11] H. Attiya, M. Herlihy, O. Rachman, Atomic snapshots using lattice agreement, Distributed Comput. 8 (3) (1995) 121–132.
[12] H. Attiya, N. Lynch, N. Shavit, Are wait-free algorithms fast?, J. ACM 41 (4) (1994) 725–763.
[13] H. Attiya, O. Rachman, Atomic snapshots in O($n\log n$) operations, SIAM J. Comput. 27 (2) (1998) 319–340.
[14] E. Borowsky, E. Gafni, Immediate atomic snapshots and fast renaming, Proc. 12th ACM Symp. on Principles of Distributed Computing, pp. 1993, 41–51.
[15] E. Borowsky, E. Gafni, Generalized FLP impossibility result for t-tesilient asynchronous computations, Proc. 25th ACM Symp. on the Theory of Computing, 1993, pp. 91–100.
[16] E.A. Brewer, C.N. Dellarocas, Proteus User Documentation, Version 4.0, March 1992.
[17] E.A. Brewer, C.N. Dellarocas, A. Colbrook, W.E. Weihl, Proteus: a high performance parallel architecture simulator, MIT Tech. Report /MIT/LCS/TR-561, September 1992.
[18] T.D. Chandra, C. Dwork, Using consensus to solve atomic snapshots, Manuscript, 1993.
[19] Digital Equipment Corporation, ALPHA system reference manual.
[20] D. Dolev, N. Shavit, Bounded concurrent time-stamping, SIAM J. Comp. 26 (2) (1997) 418–455.
[21] C. Dwork, M. Herlihy, S.A. Plotkin, O. Waarts, Time lapse snapshots, in: D. Dolev, Z. Galil, M. Rodeh (Eds.), Proc. Israel Symp. on the Theory of Computing and Systems, Haifa, Israel, May 1992, pp. 154–170.
[22] D. Gawlick, Processing 'hot spots' in high performance systems, Proc. COMPCON '85, 1985.
[23] R. Gawlick, Concurrent time-stamping made simple, Tech. Report, M.Sc. Thesis, Laboratory for Computer Science, MIT/LCS/TR-556, 1992.
[24] M. Herlihy, Wait free synchronization, ACM Trans. Programming Languages Systems 13 (1) (1991) 124–149.
[25] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Trans. Programming Languages Systems 15 (5) (1993) 745–770.
[26] M. Herlihy, N. Shavit, The asynchronous computability theorem for t-resilient tasks, Proc. 25th ACM Symp. on Theory of Computing, 1993, pp. 111–120.
[27] M. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Programming Languages Systems 12 (3) (1990) 463–492.
[28] IBM Corporation, POWER PC reference manual.
[29] A. Israeli, M. Li, Bounded time stamps. Distributed Computing (6) (1993) 205–209.
[30] L.M. Kirousis, P. Spirakis, P. Tsigas, Reading many variables in one atomic operation: Solutions with linear or sublinear complexity, IEEE Trans. Parallel Distributed Systems 5 (1994) 688–696.
[31] L. Lamport, On interprocess communication, Parts I and II, Distributed Comput. 1 (1986) 77–101.
[32] MIPS Computer Company, The MIPS RISC Architecture.
[33] M. Moir, Practical implementations of non-blocking synchronization primitives, Proc. 16 Symp. on the Principals of Distributed Computing, Santa Barbara, 1997, pp. 219–228.
[34] M. Saks, F. Zaharoglou, Optimal space distributed move-to-front lists, Proc. 10 Symp. on the Principals of Distributed Computing, Montreal, 1991, pp. 65–73.
[35] N. Shavit, D. Touitou, Elimination trees and the constructions of pools and stacks, Theory Comput. Systems 30 (6) (1997) 645–670.
[36] N. Shavit, A. Zemach, Diffracting trees ACM Trans. Comput. Systems, TOCS 14 (4) (1996) 385–428.