Math. Systems Theory **28**, 469-486 (1995)

Mathematical Systems Theory

Linear-Time Snapshot Implementations in Unbalanced Systems*

A. Israeli,[1] A. Shaham,[2] and A. Shirazi[2]

[1] Department of Electrical Engineering, Technion, Haifa 32000, Israel

[2] Department of Computer Science, Technion, Haifa 32000, Israel

Abstract. An *atomic snapshot memory* object in shared memory systems enables a set of processes, called *scanners*, to obtain a consistent picture of the shared memory while other processes, called *updaters*, keep updating memory locations concurrently. In this paper we present two conversion methods of snapshot implementations. Using the first conversion method we obtain a new snapshot implementation in which the scan operation has linear time complexity and the time complexity of the update operation becomes the sum of the time complexities of the original implementation. Applying the second conversion method yields similar results, where in this case the time complexity of the update protocol becomes linear. Although our conversion methods use unbounded space, their space complexity can be bounded using known techniques.

One of the most intriguing open problems in distributed wait-free computing is the existence of a linear-time implementation of this object. Using our conversion methods and known constructions we obtain the following results:

- Consider a system of n processes, each an updater and a scanner. We present an implementation in which the time complexity of either the update or the scan operation is linear, while the time complexity of the second operation is $O(n \log n)$.
- We present an implementation with linear time complexity when the number of either updaters or scanners is $O(n/\log n)$, where n is the total number of processes.

---

* A preliminary version of this paper appeared in *Proceedings of the Seventh Workshop on Distributed Algorithms*, 1993. The first two authors were partially supported by NWO through NFI Project ALADDIN under Contract Number NF 62-376.

- We present an implementation with amortized linear time complexity when one of the protocols (either upate or scan) is executed significantly more often than the other protocol.

1. Introduction

Consider a system of n processes communicating through shared memory in which write and read operations are executed instantaneously. At any given time t, each memory cell holds a well-defined value which is the value that was most recently written to it (or its initial value if no such write action occurs before t. A *snapshot* at time t is the vector of values held by all memory cells at t. An *atomic shapshot memory* (for brevity, a *snapshot memory*) is an object that allows some processes to acquire a snapshot while other processes can update their memory cells *concurrently*. An *implementation* of a snapshot memory object consists of two protocols called the *updater* protocol and the *scanner* protocol. A process that wishes to update one of the memory cells executes the updater protocol; a process that wishes to acquire a snapshot executes the scanner protocol. Implementations of snapshot memories are key tools in designing concurrent protocols and have many applications. Among the applications are randomized consensus [As], concurrent timestamping [DS], approximate agreement [ALS], and wait-free implementation of data structures [AH].

Traditionally, a snapshot memory object is implemented by means of *locking*—a process that wishes to scan the memory first locks it so no other process can update a memory cell until the scan operation is completed. This approach is used in many database systems satisfactorily. In contrast to the locking approach, there is an increasing interest in *wait-free* implementations. In these implementations no process is required to wait for the actions of another process while executing its own scan or update protocol. Wait-free implementations have a strong practical motivation: in a multiprocessor environment, processors of different speeds frequently need to cooperate. In such cases it is inefficient to allow a process executing on a fast processor to wait for a process executing on a slow processor. Moreover, in a multiprocess environment, processes may be delayed for long periods due to swapping, I/O operations, page faults, etc. Once more, waiting for a delayed process decreases the throughput. In addition, wait-free implementations are resilient to process failures.

A wait-free implementation is evaluated by two complexity measures:

1. Time complexity—the maximal number of read and write actions during a single execution of an update or a scan operation..
2. Space complexity—the maximal size of the shared memory (not including the space in which the actual value is held) used by the implementation.

1.1. *Related Work*

The wait-free snapshot memory object was proposed independently by Afek *et al.* in [AAD] and by Anderson in [An1] and [An2]. In [An2] Anderson presented an exponential-time implementation for snapshot memory. An implementation with quadratic time complexity was presented in [AAD]. Another quadratic implementation was presented by Aspnes and Herlihy in [AH]. A linear implementation for a system with *one* scanner was presented by Kirousis *et al.* in [KST1] and [KST2].

A new approach was proposed by Attiya *et al.* in [AHR]. In this paper the authors introduced the notion of *lattice agreement* and showed that one can implement a snapshot memory using an implementation of lattice agreement without increasing the order of the time complexity. Then they presented a lattice-agreement implementation whose time complexity is linear using test and set registers for two processes. This last implementation induces a randomized implementation of a snapshot memory using read write registers. The expected time complexity of this implementation is $O(n \log^2 n)$, where n is the total number of processes. Another randomized implementation, with the same expected time complexity, was presented by Chandra and Dwork [CD].

Deterministic implementations of the lattice-agreement object (and hence for the snapshot memory object) were proposed independently by Israeli and Shirazi in [IS2] (time complexity $O(n^{1.5} \log n)$) and by Attiya and Rachman in [AR] (time complexity $O(n \log n)$).

The time complexity of all these implementations (except the single-scanner implementation) is superlinear. A linear-time implementation for a similar object, called *time-lapse* snapshot memory, was presented by Dwork *et al.* in [DHPW]. This object however is slightly weaker than snapshot memory, since time-lapse snapshots may be inconsistent. Hoepman and Tromp showed in [HT] that in order to implement a snapshot memory object it suffices to consider a system in which the value fields are single bits.

Several new results appeared recently. Israeli and Shirazi showed in [IS3] that the time complexity of an optimal deterministic update protocol is $\Theta(\min\{u, s\})$, where u is the number of updaters and s is the number of scanners. In an interesting paper [ICMT] Inoue *et al.* presented a linear-time implementation for snapshot memories. However, this implementation assumes a model which differs from the standard model. Namely, it assumes that the registers can be written by all the processes (and not by a single process). In [BIS] Ben-dor *et al.* studied the space complexity required by snapshot memories and showed two lower bounds for special cases.

1.2. Our Results

A common conjecture states: *An implementation of a snapshot memory exists such that the time complexity of both* **update** *and* **scan** *protocols is linear.* In this paper we present two methods for converting an arbitrary implementation of snapshot memory into another implementation, such that one of its protocols has linear time complexity. Let $\mathcal{I}$ be any snapshot implementation. For a system of u updaters and s scanners, we denote the time complexities of the update and scan operations in $\mathcal{I}$ by $utc_{\mathcal{I}}(u, s)$ and $stc_{\mathcal{I}}(u, s)$, respectively. The first method converts $\mathcal{I}$ into an implementation $\mathcal{I}'$ for which $utc_{\mathcal{I}'}(u, s) = utc_{\mathcal{I}}(u, u) + stc_{\mathcal{I}}(u, u) + 1$, and

$stc_{\mathscr{I}'}(u, s) = u$. The second method converts $\mathscr{I}$ into an implementation $\mathscr{I}'$ for which $utc_{\mathscr{I}'}(u, s) = u$ and $stc_{\mathscr{I}'}(u, s) = u + utc_{\mathscr{I}}(s, s) + stc_{\mathscr{I}}(s, s)$.

In view of former work, two remarks are in order. First, both of our conversion methods store unbounded counters in the shared memory. Nevertheless, both of them can be bounded by applying the techniques of [DHW]. However, doing so will increase the time complexities of the resulting implementations: the time complexities of the scan protocol in the first conversion method and of the update protocol in the second conversion method will be linear in the total number of processes (instead of in the number of updaters only).

Our second remark addresses the *multiwriter snapshot memory* object. In this object the updating processes can change *all* the locations in the memory (only one location at a time). Anderson presented, in [An1], a general method that converts an implementation of snapshot memory into an implementation of multiwriter snapshot memory. Applying this method to our first conversion method will not change the order of the time complexities in the resulting implementation. Unfortunately, applying this method to our second conversion method is not better than applying it to the original implementation.

Since all known general implementations of snapshot memory satisfy that the time complexities of both protocols are superlinear and of the same order of magnitude, our conversion methods improve *all* known implementations. In particular, applying each of these methods to the implementation presented in [AR], we obtain two implementations whose time complexities are the best known. Consider a system of n processes, each of which is an updater and a scanner. In these implementations one operation (either update or scan) has linear time complexity, while the time complexity of the second operation is $O(n \log n)$. The time complexities of these two implementations are not comparable. Since we introduce conversion methods for an *arbitrary implementation*, these methods improve any future implementation in which the time complexities of the update and scan protocols are superlinear.

We remark that the linear time complexity achieved by our conversions is optimal (the second conversion method yields an implementation which is only asymptotically optimal): In the worst case, every scanner must read the registers of all the updaters. Otherwise, it is possible for a scanner not to notice an update operation completed before the scan operation begun. Also, as mentioned above, [IS3] showed that the time complexity of the update protocol is $\Omega(n)$. We stress that both the lower bounds and the implementations do not bound the size of the registers being used.

As the implementations of a snapshot memory presented in this paper are not linear for both protocols in a system of n updaters and n scanners, we turn to study the circumstances under which implementations with two linear protocols exist. We note that both protocols of the implementation presented by Afek *et al.* work in quadratic time in the number of updaters only. Therefore, their implementation works in linear time whenever the number of updaters is $O(\sqrt{n})$, where n is the total number of processes. A system is *unbalanced* if one of the following holds:

1. Let u and s be the number of updaters and scanners, respectively. Either u/s or s/u is $\Omega(\log n)$.
2. Let n_u and n_s be the number of times the update protocol and the scan protocol, respectively, are executed. Either n_u/n_s or n_s/n_u is $\Omega(\log n)$.

Using our conversion methods, we derive two snapshot implementations whose time complexities are linear for systems that are unbalanced under the first definition. Furthermore, the amortized time complexity of these implementations is linear in systems that are unbalanced under the second definition. Given an unbalanced system, we obtain the linear-time implementation by applying the appropriate conversion method to the implementation of [Ar] (for which both protocols work in time $O(n \log n)$).

Finally, we remark that our results have not been rendered obsolete by the recent linear-time implementation of [ICMT]. This is mainly because their implementation assumes, in contrast with the accepted model, that all the processes can write into all the registers. Moreover, when the number of updaters is significantly smaller than the number of scanners, their implementation will also be improved by our conversion methods. In addition, our second conversion method was used in [IS3] to determine the time complexity of an optimal update protocol.

The rest of this paper is organized as follows: The model of computation is presented in Section 2. The conversion methods are presented in Sections 3 and 4. Section 5 contains the main results of this paper. Concluding remarks are given in Section 6.

2. Model and Requirements

In this section we define the model of computation and the atomic snapshot memory implementation. Informally, *processes* are deterministic sequential threads of control that communicate through shared data structures called *objects*. Formally, we model processes and objects using a simplified form of the I/O automata of Lynch and Tuttle [LT]. Processes access the objects by executing *operations*. Operations are either *atomic* or *nonatomic*. An atomic operation is executed instantaneously. A nonatomic operation is built from several operations executed one after the other (for convenience, we neglect the internal computation). A process is described by its *protocol*—a nonatomic operation that can be executed several times (possibly an infinite number of times). Each atomic operation corresponds to a single state-transition. Each protocol has a distinguished atomic operation, called the *initial operation*, corresponding to the initial state.

Executions are described under the interleaving model. A global state is described by a *configuration*—a vector containing the state of each process and the state of each object. The system's *initial configuration* contains the processes' and objects' initial states. The execution of an (atomic) operation is called an (*atomic*) *action*. An execution is a sequence of configurations and atomic actions $E = conf_0$, act_1, $conf_1, \dots, conf_{i-1}$, act_i, $conf_i, \dots$, where $conf_0$ is the system's initial configuration and, for every $i > 0$, $conf_i$ is obtained from $conf_{i-1}$ by executing the atomic action act_i ($conf_i$ is the *occurrence configuration of* act_i). We stress that no assumption is made on the relative speeds of the processes. Usually, atomic actions of different processes are interleaved. An execution is *sequential* if atomic actions of different nonatomic operations are not interleaved.

An object is specified by a set of legal sequential executions [HW]. The notion of *implementation* is discussed and formally defined in [H] and [AAD]. We omit the formal definitions and give the essence of implementing an object. An implementation is *wait-free* if, in all its executions, the number of atomic actions executed in each protocol is bounded, where the bound may depend on the number of processes in the system. We require our implementations to be wait-free. In addition, an implementation must satisfy the *linearizability* correctness condition [HW]. Informally, we want each nonatomic action to *appear as if* it was executed instantaneously. In addition, the order between nonatomic actions that are not concurrent should be preserved.

Formally, consider any execution, E, of an implementation in which the atomic actions of the processes are interleaved. The execution induces a *partial order* $<_E$ on nonatomic operations executed in E: Let a, b be two nonatomic operations executed in E. If a ended before b started, than $a <_E b$. An execution is *linearizable* if $<_E$ can be extended to a *complete* order $<_S$, where $<_S$ is the order induced by some sequential execution of the nonatomic operations in E and S is one of the legal sequential executions. Clearly, since $<_S$ extends $<_E$, the order between nonatomic operations that are not concurrent in E is preserved. An implementation is *linearizable* if all its executions are linearizable.

Let a be a nonatomic operation executed in E. We denote the start and end configurations of a in E by $start_E(a)$ and $end_E(a)$, respectively (if a does not complete in E, then $end_E(a) = \infty$). The *execution interval* of a in E includes all the configurations in the interval $[start(a), end(a)]$. In order to prove that E is linearizable, it is sufficient to assign a *linearization point* to each operation in E, a, such that the linearization point of a lies in the execution interval of a in E. The sequential execution obtained must be one of the legal sequential executions.

We consider two types of shared objects: *atomic registers*, henceforth *registers* and *atomic snapshot memory*. Processes access registers by executing *write* and *read operations*. A write operation stores a new value in the register and a read operation obtains the value stored in the register. In the initial configuration, each register holds its *initial value*. Each operation is executed instantaneously[1] and each read operation returns the value written by the most recent, preceding, write operation, or the initial value if no such write operation exists. The registers are *single writer multireader* registers. That is, each register is associated with one process, called its *owner*, which is the only process that can write to it. However, any process can read the register.

[1] It is possible that an atomic register is implemented from weaker registers [L1], [L2], [VA], [ILV], [Ab], [IS1]. However, in such cases the linearizability of the implementation allows us to assume that the operations appear as if they are executed instantaneously [HW].

An atomic snapshot memory, henceforth a *snapshot memory*, is defined with respect to a set of *cells* (this set is called a compound register in [An2]). The processes are divided into two groups: *updaters* and *scanners*. Updater i, denoted U_i, owns cell i and can change the value of its cell in an *update operations*. Scanner j, denoted S_j, obtains the value of *all* the cells in a *scan operation*. Though the two groups of processes are not necessarily distinct, it is convenient to assume that they are. This does not harm the generality of our results since a process which is both an updater and a scanner can be viewd as two processes. We denote the number of updaters and scanners by u and s, respectively. Also, n denotes the total number of processes $(n = u + s)$. A snapshot object must be implemented from registers. An implementation of a snapshot object consists of two protocols, one for updaters and one for scanners.

The time complexity of a snapshot implementation is a pair $(utc(u,s), stc(u,s))$, where $utc(u,s)$ and $stc(u,s)$ are the maximal number of atomic operations on registers performed in an update and scan operation, respectively. We stress that, as in all previous papers, the cost of both the write operation and the read operation is one, regardless of the length of the register which is accessed. As we present conversions of snapshot implementations, one of our objects is a snapshot memory. The original implementation and its two protocols are called the *elementary implementation* and *elementary protocols*, respectively. Since the elementary implementation is linearizable, we can assume that operations on this object occur instantaneously. However, since we measure the time complexity in the number of operations on registers, the time complexity incurred by these operations is the time complexity of the elementary protocol used. Throughout the rest of this paper, it is understood that the read, write, elementary scan, and elementary update operations are atomic, while the scan and update operations are not.

3. Implementations with Linear Scan Protocols

In this section we describe a method to convert an arbitrary implementation of snapshot memory to another implementation. The time complexity of the scan protocol in the new implementation is equal to the number of updaters. This is best possible since for any implementation an execution exists in which the scan protocol must read all the updaters' registers (otherwise, it is possible that the snapshot returned might not include all the update actions completed prior to the start of the scan operation). The underlying idea is that the updaters execute the scan operation for the scanners, using the elementary scan protocol. The result of each such elementary scan is an elementary snapshot and all elementary snapshots are ordered. A scanner reads an elementary snapshot from each updater and returns the latest one.

3.1. Description

The update and scan protocols are presented in Figure 1. The elementary update and scan protocols are denoted by *escan* and *eupdate*, respectively. Each updater, U_i, keeps an internal variable, $count_i$, which is initialized to zero and incremented by 1

Update$_i$(value)	
$count_i \leftarrow count_i + 1$	
eupdate(*value,count$_i$*)	eu^a_a
$s[1 \ldots u] \leftarrow$ **escan**	es^a_i
$r_i \leftarrow$ **write** $(s[1 \ldots u])$	w^a_i
Scan$_j$	
for $k \leftarrow 1$ **to** u **read** (r_k)	$r^b_j[1] \ldots r^b_j[u]$
return dominating elementary snapshot	

Fig. 1. The protocols for a few updaters.

every time U_i executes an update operation. The new update protocol for U_i consists of an elementary update operation of the record (*value, count*). Next, an elementary scan operation is executed. The elementary snapshot obtained by this escan action is written into an additional register called r_i which can be read by all the scanners. The ath update operation of U_i is denoted by U^a_i, the value it gets as input is denoted by val^a_i; its elementary update, elementary scan, and write operations are denoted by eu^a_i, es^a_i, and w^a_i, respectively. The elementary snapshot returned by es^a_i is denoted by $esnap^a_i$.

The new scan protocol works as follows: First, each r_i is read to obtain an elementary snapshot from each updater U_i and then the dominating elementary snapshot is returned. The domination order is naturally defined on the *count* fields in the following way: Let *esnap* and *esnap'* be two elementary snapshots. *esnap* dominates *esnap'* if for every i, $1 \leq i \leq u$, the count field in the ith entry of *esnap* is not less than the count field in the ith entry of *esnap'*. Note that since for every updater, $U_i, count_i$ is nondecreasing, and since the elementary implementation is linearizable, it follows that all elementary snapshots which are written into registers are fully ordered by domination. The bth scan operation executed by S_j is denoted by S^b_j; its subactions are denoted by $r^b_j[1], \ldots, r^b_j[u]$ and the snapshot it returns is denoted by $snap^b_j$. The complexity of the update protocol is equal to the sum of the complexities of the elementary protocols. The complexity of the scan protocol is equal to u, the number of updaters.

3.2. Linearization Scheme

In the linearization scheme for the protocols, we allow an action to be linearized within its execution interval or just after it. That is, we allow an action to be linearized after the last configuration in its execution interval, but before the next configuration in the execution. Clearly, this preserves the order between noncurrent operations.[2] For the rest of this section we consider an arbitrary execution, E, of the converted implementation and prove that the execution is linearizable. For brevity, we omit the reference to the execution throughout the remainder of this section. In particular, $<$ and $\leq$ should ble read as $<_E$ and $\leq_E$, respectively. For any nonatomic action a, the linearization point of a, denoted $lin(a)$, is related to some specific configuration. This configuration is called the *linearization configuration* of a and is denoted by $lin\_conf(a)$. Sometimes, more than one action is serialized by the

[2] Alternatively, we could have added a dummy operation at the end of the protocol

same configuration. In some of the arguments below, it is simpler to consider the linearization configuration. The starting and ending configurations of a are denoted by $start(a)$ and $end(a)$, respectively. For an atomic action $a, occ(a)$ denotes the occurrence configuration of a.

The occurrence configurations of the atomic actions are used to define the linearization for the new implementation. We first define the linearization for update actions.

Definition 1. The linearization configuration for update action U_i^a is defined to be the occurrence configuration of the earliest write, w_k^c, whose corresponding escan action, es_k^c, occurred after eu_i^a:

$$lin\_conf(U_i^a) = \min_{j,\, b}\{occ(w_j^b) | occ(es_j^b) > occ(eu_i^a)\}.$$

We say that U_i^a is linearized by w_k^c. The linearization point of U_i^a is just after $lin\_conf(U_i^a)$. If several update actions are linearized by the same write action, they are further linearized in the order of their own eupdate actions, in this way no two update actions are linearized at the same point.

Definition 2. Let U_l^d be the latest update action whose value is included in $snap_j^b$. The linearization configuration of scan action S_j^b is defined to be the maximum between its starting configuration and the linearization configuration of U_l^d:

$$lin\_conf(S_j^b) = \max\{start(S_j^b),\ \max_{k,\, c}\{lin\_conf(U_k^c) | val_k^c \in snap_j^b\}\}.$$

In the first case the linearization point is the same as the linearization configuration. In the second case we say that S_j^b is linearized by U_l^d, and the linearization point of S_j^b is just after $lin(U_l^d)$, where ties are broken arbitrarily.

3.3. *Correctness Proof*

Lemma 1. *Every action is linearized within its execution interval.*

Proof. We start with update actions. Let U_i^a be an arbitrary update action. Definition 1 immediately implies:

- $lin\_conf(U_i^a) > occ(eu_i^a)$.
- $lin\_conf(U_i^a) \leq occ(w_i^a) = end(U_i^a)$.

The proof follows.

We now prove the lemma for scan actions: Let S_j^b be an arbitrary scan action. If S_j^b is linearized at its beginning, then the proof is immediate. Therefore, we assume that S_j^b is linearized after its beginning by update action U_k^c. Hence, $val_k^c \in snap_j^b$ and

$$lin\_conf(S_j^b) = lin\_conf(U_k^c).$$

Let $esnap_l^d$ be the elementary snapshot returned by S_j^b (possibly $U_k^c = U_l^d$). Since $val_k^c \in esnap_l^d$, Definition 1 implies

$$lin\_conf(U_k^c) \le occ(w_l^d).$$

Since S_j^b returns $esnap_l^d$, we obtain that

$$occ(w_l^d) < occ(r_j^b[l]) \le end(S_j^b).$$

Combining these inequalities we get that $lin\_conf(S_j^b) < end(S_j^b)$. □

The linearization configuration of an update action might be up to $2{\cdot}u$ configurations away from the occurrence configuration of its eupdate action. However, the following lemma shows that the complete order between the linearized update actions is identical to the complete order between their eupdate actions. This fact greatly simplifies the proof of Lemma 3.

Lemma 2. *Let U_i^a and U_j^b be two arbitrary update actions. If $occ(eu_i^a) < occ(eu_j^b)$, then $lin(U_i^a) < lin(U_j^b)$.*

Proof. We assume that $occ(eu_i^a) < occ(eu_j^b)$ and prove that $lin(U_i^a) < lin(U_j^b)$. First, we claim that $lin\_conf(U_i^a) \le lin\_conf(U_j^b)$. To prove the claim, assume U_j^b is linearized by w_k^c. Clearly, $occ(eu_j^b) < occ(es_k^c)$. Since $occ(eu_i^a) < occ(eu_j^b)$, we have that $occ(eu_i^a) < occ(es_k^c)$. By Definition 1 we have that $lin\_conf(U_i^a) \le occ(w_k^c)$. Hence, the claim.

Next, we consider two cases. If $lin\_conf(U_i^a) \ne lin\_conf(U_j^b)$, the above claim implies $lin\_conf(U_i^a) < lin\_conf(U_j^b)$ and we are done. On the other hand, if U_i^a and U_j^b are both linearized by w_k^c, by Definition 1 they are further linearized in the order of their eupdate actions. The lemma follows. □

By Lemma 2, every elementary snapshot is also the valid snapshot just after the configuration in which the last update included in it is linearized. To complete the correctness proof, we show that the elementary snapshot returned by any scan action is the valid snapshot at the linearization point of the scan action.

Lemma 3. *Snapshot $snap_i^a$ is the valid snapshot at $lin(S_i^a)$.*

Proof. Let S_i^a be an arbitrary scan action and let U_j^b be the last update whose value is included in $snap_i^a$. From the previous lemma it is clear that $snap_i^a$ is a snapshot just after $lin(U_j^b)$. Hence, if S_i^a is linearized by U_j^b we are done.

Next, we look at the case that S_i^a is linearized at its beginning. Definition 2 ensures that a scan action does not return any value whose update action is linearized after that scan action. Let U_j^b be the last update action of updater j that is linearized before $start(S_i^a)$. In order to prove the lemma we have to show that val_j^b is included in at least one elementary snapshot read by S_i^a. Since the dominating elementary snapshot is returned, it is clear that the most recent value of each updater is returned. We assume that U_j^b is linearized by U_k^c (possibly

$U_j^b = U_k^c$). Therefore, $val_j^b \in esnap_k^c$. Since U_j^b is linearized before $start(S_i^a)$, it holds that $occ(w_k^c) < start(S_i^a)$. The register of updater k is read in $S_i^a[k]$. From the above observations, it follows that val_j^b is included in the elementary snapshot obtained. □

4. Implementations with Linear Update Protocols

In this section we describe a method to convert an arbitrary elementary implementation of snapshot memory to another implementation with a *linear-time* update protocol. The underlying idea is a modification of a single scanner protocol of Kirousis *et al.* in [KST2]. Similar to the previous section, each updater prepares information for the scanners. However, the information is less precise—instead of preparing an elementary snapshot, the updater prepares a *view*: It reads the registers of all the updaters and for every updater it chooses the latest value it sees. The updater completes its protocol by writing its view. A scanner also begins by preparing a view, composed from the views held by updaters. Clearly, the views held by different scanners are only *partially* ordered. Therefore, the views may not serve as snapshots. In order to achieve a *complete* order, each *scanner* eupdates its view and then performs an escan operation to obtain the views held by the other scanners. Choosing the latest value seen for each updater yields a snapshot.

4.1. Description

The Protocols appear in Figures 2 and 3. Once more, updater U_i keeps an internal variable, $count_i$, which is initialized to 0 and incremented at the beginning of every update action. The register of U_i, called $view_i$, consists of an array of u entries. Each entry is a pair of the form (*value, count*), and the kth entry of $view_i$ always holds a (*value, count*) pair of U_k. The updater protocol is to read the views of all other updaters, and for each updater to choose the pair with the highest count. All these pairs are stored in a local view variable called *lview* (the count fields of *lview* are initialized to −1). After that, U_i assigns its new (*value, count*) pair to *lview*[i] and

```
begin
  count_i ← count_i + 1
  for j ← 1 to u
    temp ← read(view_j)                    r_i^a[j]
    for k ← 1 to u
      if temp[k] . count > lview[k]  count then lview[k]←temp[k]
    endfor
  endfor
  lview[i] ← (count_i, value)
  view_i ← write(lview)                    w_i^a
end
```

Fig. 2. Protocol for U_i.

```
begin
  for k ← 1 to u
    temp ← read(view_k)                r_j^b[k]
    for l ← 1 to u
      if temp[l] count > lview[l] . count then lview[l] ← temp[l]
    endfor
  endfor
  eupdate(lview)                       eu_j^b
  scan ← escan                         es_j^b
  for k ← 1 to s
    for l ← 1 to u
      if scan[k][l] . count > lview[l] . count then lview[l] ← scan[k][l]
    endfor
  endfor
  Return (lview)
end
```

Fig. 3. Protocol for S_j.

then *lview* is written into $view_i$. We denote the *a*th update action of updater i by U_i^a. The atomic actions executed during U_i^a are denoted by $r_i^a[1] \dots r_i^a[u], w_i^a$. The value and the view written during U_i^a are denoted by val_i^a and $view_i^a$, respectively. The complexity of the update protocol is u, the number of updaters.

The register of each scanner also holds a view; these registers are accessed by the elementary update and scan protocols. The scanner protocol consists of two parts: In the first part the scanner reads the updaters' registers and computes a local view from the views of all updaters. In the second part the scanner eupdates its local view and then escans the views of all scanners. Following the elementary scan operation, the local view is computed from the views obtained in the snapshot action. At this point, $lview_j$ holds a snapshot which is returned. We denote the *b*th scan action of scanner j by S_j^b. The atomic actions executed during S_j^b are denoted by $r_j^b[1] \cdots r_j^b[u], eu_j^b, es_j^b$. The view eupdated in action eu_j^b and the snapshot returned by S_j^b are denoted by $view_j^b$ and $snap_j^b$, respectively. Recall that we neglect the internal computation, and hence the last atomic action of S_j^b is es_j^b. The complexity of the scan protocol is equal to the number of updaters plus the sum of the complexities of the elementary protocols.

4.2. Linearization Scheme

As in the previous section, we consider an arbitrary execution, E, of the converted implementation and prove that the execution is linearizable. For brevity, we omit the reference to the execution throughout the remainder of this section. We use the occurrence configurations of the atomic actions to define the linearization for the new implementation. We start with linearizing scan actions.

Definition 3. The linearization configuration of scan action S_i^a is defined to be the minimum between $end(S_i^a)$ and the occurrence configuration of the first write action, w_k^c, for which c is larger than the count of $snap_i^a[k]$:

$$lin\_conf(S_i^a) = \min\{end(S_i^a), \min_{j,b}\{occ(w_j^b) | b > snap_i^a[j].count\}\}.$$

In the first case the linearization point is the same as the linearization configuration. In the second case we say that S_i^a is linearized by w_k^c, and the linearization point is just before the linearization configuration. If two scan actions are linearized by the same write action, then they are further linearized by the domination order of the views they return as snapshots. If the snapshots are equal, then the scan actions are linearized arbitrarily.

The validity of the above definition is guaranteed by Lemma 5 which shows that all views returned as snapshots are ordered by domination and the fact that two scan actions of the same scanner cannot be linearized by the same write action.

Definition 4. The linearization configuration of update action U_j^b is defined to be the minimum between $end(U_j^b)$ and the linearization configuration of the first scan action S_l^b which returns val_j^b:

$$lin\_conf(U_j^b) = \min\{end(U_j^b), \min_{k,c}\{lin\_conf(S_k^c) | val_j^b \in snap_k^c\}\}.$$

In the first case the linearization point is the same as the linearization configuration. In the second case we say that U_j^b is linearized by S_l^d, and the linearization point is just before $lin(S_l^d)$, where ties are broken arbitrarily.

4.3. Correctness Proof

Lemma 4. *Every view written by an updater dominates all previous views written by it; the same holds for every view that is eupdated by a scanner.*

Proof. We prove the lemma by induction on the configurations. The lemma trivially holds for the first configuration. Assume the next action is w_i^a (by an updater) or eu_i^a (by a scanner). For any updater, U_j, let U_k be the updater from which the (*value, count*) pair of U_j in $view_i^{a-1}$ was taken. By the induction hypothesis, the count of U_j read in $r_i^a[k]$ is not smaller than the count of U_j read in $r_i^{a-1}[k]$, hence the lemma. □

We can now prove in a similar manner that all the views returned as snapshots are ordered by domination.

Lemma 5. *If $snap_i^a$ and $snap_j^b$ are views returned as snapshots (where i is not* *necessarily different from j), then either* $view_i^a$ *dominates* $view_j^b$ *or* $view_j^b$ *dominates* $view_j^a$.

The following lemma shows that every action is linearized within its execution interval:

Lemma 6. *Every action is linearized within its execution interval.*

Proof. By Definitions 3 and 4, all actions are linearized no later than their last atomic action. We have to show that they are not linearized before their first read action. We start with scan actions. Let S_i^a be an arbitrary scan action. If S_i^a is linearized at its ending configuration, we are done. Assume S_i^a is linearized by w_j^b. In this case $snap_i^a[j].count < b$ and hence $start(S_i^a) \leq occ(r_i^a[j]) < occ(w_j^b)$ (otherwise in $r_i^a[j], S_i$ reads val_j^b).

We continue with update actions: Assume by way of contradiction that U_i^a is linearized before $r_i^a[1]$. In this case there is a scan action S_j^b which returns val_i^a, and S_j^b is linearized before $r_i^a[1]$. Recall that val_i^a is written for the first time in action w_i^a. Since it is included in $snap_j^b$ we can conclude that $occ(w_i^a) < occ(es_j^b)$. Hence S_j^b is not linearized at es_j^b, but sooner by a write action. Let w_k^c be the write action by which S_j^b is linearized before $start(U_i^a)$. It follows that $val_k^c \notin snap_j^b$ and $occ(w_k^c) < start(U_i^a)$. However, $occ(w_k^c) < start(U_i^a)$ implies that $val_k^c \in view_i^a$. Since val_i^a appears in $snap_j^b, val_k^c$ should be in $snap_j^b$ as well, a contradiction. □

To complete the correctness proof, we show that the scan protocol returns snapshots:

Lemma 7. *Snapshot* $snap_i^a$ *is the valid snapshot at* $lin(S_i^a)$.

Proof. We show that if val_k^c belongs to $snap_i^a$, then $lin(U_k^c) < lin(S_i^a) < lin(U_k^{c+1})$.

By Definition 4, $lin(U_k^c) < lin(S_i^a)$. Also, by Definition 3, we get that $lin(S_i^a) < occ(w_k^{c+1})$. Hence, if U_k^{c+1} is linearized at w_k^{c+1} we are done, so we assume that $lin\_conf(U_k^{c+1}) < occ(w_k^{c+1})$. In this case Definition 4 implies that U_k^{c+1} is linearized by some scan action S_j^b where $val_k^{c+1} \in snap_j^b$. Since $val_k^c \in snap_i^a$, Lemma 5 implies that $snap_j^b$ dominates $snap_i^a$. Since $val_k^{c+1} \in snap_j^b$, it is clear that $occ(w_k^{c+1}) < occ(es_j^b)$. Hence, S_j^b is not linearized at es_j^b (since U_k^{c+1} is linearized by S_j^b *before* $occ(w_k^{c+1})$), but by some write action w_m^d where $snap_j^m[m].count < d$. Since $snap_j^b$ dominates $snap_i^a$, we get that $snap_i^a[m].count < d$ as well. Recall that $lin\_conf(U_k^{c+1}) = occ(w_m^d)$, and that we want to show $lin(S_i^a) < lin(U_k^{c+1})$. Hence, if $lin\_conf(S_i^a) < occ(w_m^d)$, we are done. Otherwise, by Definition 3, $lin\_conf(S_i^a) = occ(w_m^d)$. Since $view_i^a$ is dominated by $view_j^b$, it follows that $lin(S_i^a) < lin(S_j^b)$. Since U_k^{c+1} is linearized by S_j^b we get $lin(S_i^a) < lin(U_k^{c+1}) < lin(S_j^b)$, and the proof follows. □

5. Main Results

In this section we use the conversion methods presented in the previous sections in order to establish the main claims of the paper.

Theorem 8. *Let I be an implementation of snapshot memory for which the time complexities of the update and scan protocols are $f(u, s)$ and $g(u, s)$, respectively. There is an implementation of snapshot memory for which the time complexities of the update and scan protocols are $f(u, u) + g(u, u) + 1$ and u, respectively. In addition, a second implementation of snapshot memory exists for which the time complexities of the update and scan protocols are u and $u + f(s, s) + g(s, s)$, respectively.*

Proof. The first part of the theorem follows by applying the first conversion method to the given implementation I. The second part of the theorem follows by applying the second conversion method to the given implementation I. □

Recall that the implementation presented in [AR] satisfies that the time complexity of both update and scan protocols is $O(n \log n)$, where n is the total number of processes in the system. Applying the above theorem to the implementation presented in [AR] as the elementary implementation, we obtain the following two corollaries:

Corollary 9. *Consider a system of n updaters and n scanners. There is an implementation of snapshot memory for which the time complexities of the update and scan protocols are $O(n \log n)$ and n, respectively. In addition, a second implementation of snapshot memory exists for which the time complexities of the update and scan protocols are n and $O(n \log n)$, respectively.*

Corollary 10. *Consider a system of u updaters and s scanners and let n be the total number of processes. If the number of either scanners or updaters is $O(n/\log n)$, then an implementation of snapshot memory exists for which the time complexities of the update and scan protocols are $O(n)$.*

In some applications the number of times that one of the operations is executed is significantly greater than the number of times that the second operation is executed. For instance, it is possible for a process to examine the state of the system (meaning perform the snapshot operation) many times before it updates its register. On the other hand, it may be the case that a process performs many update operations based on a single observation. This is possible in applications that resemble the consensus protocol of [BR], in which the entire memory is read only after many writes to the register. For this kind of unbalanced system, it is possible to amortize the total cost due to operations on the snapshot object and obtain a linear amortized cost. This is stated formally in the next corollary:

Corollary 11. *Let n_u and n_s be the number of times the update protocol and the scan protocol, respectively, are executed. If either n_u/n_s or n_s/n_u is $\Omega(\log n)$, then* *an implementation of snapshot memory exists for which the amortized time complexities of the update and scan protocols are $O(n)$.*

6. Concluding Remarks

We presented two conversion methods for snapshot implementations. The first method converts an arbitrary implementation to an implementation whose scan operation time complexity is linear; while the time complexity of the update operation becomes the sum of the time complexities of the two given protocols. The second method converts an arbitrary implementation to an implementation whose update operation time complexity is linear, while the time complexity of the scan operation becomes the sum of the time complexities of the two given protocols. These conversion methods improve all known implementations of snapshot memory, and will improve any future implementation in which the time complexities of both protocols are superlinear.

It is interesting to see whether a way to bound our conversion methods without increasing the time complexity exists. In addition, one of the most intriguing open problems in wait-free computation is whether an implementation exists that is optimal for *both* protocols. In other words, does a linear implementation of snapshot memory exist?

Acknowledgments

We are grateful to Jaap Hoepman for his careful reading of an early version of this paper. The Center for Mathematics and Computer Science in Amsterdam, CWI, was a very warm host for two of the authors who gratefully acknowledge it.

References

[Ab] U. Abraham. On Interprocess Communication and the Implementation of a Multi-Writer Atomic Register, Preprint.

[An1] J. Anderson, Composite Registers, *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990, pp. 15–29.

[An2] J. Anderson, Composite Registers, *Distributed Computing*, Vol. 6, No. 3, 1993, pp. 141–154.

[As] J. Aspnes, Time- and Space-Efficient Randomized Consensus, *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990, pp. 15–29.

[AAD] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M Merritt, and N. Shavit, Atomic Snapshots of Shared Memory, *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990, pp. 1–13.

[AH] J. Aspnes and M. Herlihy, Wait-Free Data Structures in the Asynchronous PRAM Model, *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, 1990, pp. 340–349.

[AHR] H. Attiya, M. Herlihy, and O. RAchman, Efficient Atomic Snapshots Using Lattice Agreement, *Proceedings of the 6th International Workshop on Distributed Algorithms* *and Graphs*, Haifa, November 1992 (A. Segall and S. Zaks, eds.), pp. 35–53, Lecture Notes on Computer Science, Vol. 647, Springer-Verlag, Berlin, 1992.

[ALS] H. Attiya, N. A. Lynch, and N. Shavit, Are Wait-Free Algorithms Fast? *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, 1990, pp. 55–64.

[AR] H. Attiya and O. Rachman, Atomic Snapshots in $O(n \log n)$ Operations, *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993.

[BIS] A. Ben-dor, A. Israeli, and A. Shirazi, On The Space Complexity of Snapshot Memories, Preprint.

[BR] G. Bracha and O. Rachman, Randomized Consensus in Expected $O(n^2 \log n)$ Operations, *Proceedings of the Fifth Workshop on Distributed Algorithms*, 1991.

[CD] T. D. Chandra and C. Dwork, Personal communications.

[DHPW] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts, Time-Lapse Snapshots, *Proceedings of the 1st Israeli Symposium on Theory of Computing and Systems*, Haifa, May 1992 (D. Dolev, Z. Galil and M. Rodeh, eds.), pp. 154–170, Lecture Notes in Computer Science, Vol. 601, Springer-Verlag, Berlin, 1992.

[DHW] C. Dwork, H. Herlihy, and O. Waarts, Bounded Round Numbers, *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993.

[DS] D. Dolev and N. Shavit, Bounded Concurrent Time-Stamp Systems Are Constructible!, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 1989, pp. 454–465.

[H] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization, *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, 1988, pp. 276–290.

[HT] J. H. Hoepman and J. Tromp, Binary Snapshots, *Proceedings of the 7th International Workshop on Distributed Algorithms and Graphs*, Lausanne, September 1993, pp. 18–25, Lecture Notes in Computer Science, Vol. 725, Springer-Verlag, Berlin, 1993.

[HW] M. P. Herlihy and J. M. Wing, Linearizability: A Correctness Condition for Concurrent Objects, *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, July 1990, pp. 463–492.

[ICMT] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura, Linear-Time Snapshot Using Multi-Writer Multi-Reader Registers, *Proceedings of the 8th International Workshop on Distributed Algorithms and Graphs*, 1994.

[ILV] A. Israeli, M. Li, and P. Vitanyi, Simple Multireader Registers Using Time-Stamp Schemes, Report No. CS-R8758 Center for Mathematics and Computer Science, Amsterdam, November 1987.

[IS1] A. Israeli and A. Shaham, Optimal Multi-Writer Multi-Reader Atomic Registers, *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, 1992, pp. 71–82.

[IS2] A. Israeli and A. Shirazi, Efficient Snapshot Protocol Using 2-Lattice Agreement, Preprint.

[IS3] A. Israeli and A. Shirazi, The Complexity of Updating Snapshot Memories, Information *Processing Letters*, to appear. Also in *Proceedings of the 2nd Annual European Symposium on Algorithms*, 1994, pp. 171–182, Lecture Notes in Computer Science, Vol. 855, Springer-Verlag, Berlin, 1994.

[KST1] L. M. Kirousis, P. Spirakis, and P. Tsigas, Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity, *Proceedings of the 5th International Workshop on Distributed Algorithms and Graphs*, Delphi, October 1991 (S. Toueg, P. Spirakis, and L. Kirousis, eds.), pp. 229–241, Lecture Notes in Computer Science, Vol. 579, Springer-Verlag, Berlin, 1992.

[KST2] L. M. Kirousis, P. Spirakis, and P. Tsigas, Simple Atomic Snapshots, A Linear Complexity Solution with Unbounded Time-Stamps, *Proceedings of Advances in Computing and Information—ICCI*, 1991, pp. 582–587, Lecture Notes in Computer Science, Vol. 497, Springer-Verlag, Berlin, 1991.

[L1] L. Lamport, On Interprocess Communications. Part I: Basic Formalism, *Distributed Computing*, Vol. 1, No. 2 1986, pp. 77–85.

[L2] L. Lamport, On Interprocess Communication. Part II: Algorithms, *Distributed Computing*, Vol. 1, No. 2 1986, pp. 86–101.

[LT] N. Lynch and M. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms, *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1988, pp. 137–151.

[VA] P. Vitanyi and B. Awerbuch, Atomic Shared Register Access by Asynchronous Hardware, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, pp. 233–243.

Received March 1994 and in final form January 1995.