1992

A. Israeli, A. Shaham, A. Shirazi

Linear-time snapshot protocols for unbalanced systems

Computer Science/Department of Algorithmics and Architecture Report CS-R9236 September

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Linear-Time Snapshot Protocols for Unbalanced Systems

Amos Israeli
CWI
P.O. Box 4079, 1009 AB
Amsterdam, The Netherlands
AND
Dept. of Electrical Engineering
Technion — Israel

Amnon Shaham
CWI
P.O. Box 4079, 1009 AB
Amsterdam, The Netherlands
AND
Dept. of Computer Science
Technion — Israel

Asaf Shirazi
Dept. of Computer Science
Technion — Israel

Abstract. The *snapshot* problem for shared memory systems is to enable a set of processors called *scanners* to obtain a consistent picture of the shared memory while other processors called *updaters* keep updating memory locations concurrently. One of the most intriguing open-problems in wait-free distributed computing is the existence of a linear-time solution to this problem. In this paper we show that if the number of either scanners or updaters is smaller than the square root of the total number of processors then such a linear solution exists.

1991 Mathematics Subject Classification: 68M10, 68Q22, 68Q25.
CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.
Keywords and Phrases: Snapshot, Wait-Free, Shared Memory, Register, Scan, Update, View.
Note: This work is partially supported by NWO through NFI Project ALADDIN under Contract number NF 62-376.

1 Introduction

Consider a system of processors communicating through shared memory in which write and read to the shared memory 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. The *snapshot* problem is to allow some processors to acquire a snapshot while other processors are updating their memory cells *concurrently*. A solution to the snapshot problem consists of two

Report CS-R9236
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

programs called the *updater* protocol and the *scanner* protocol. A processor that wishes to update one of its memory cells executes the updater protocol; a processor that wishes to acquire a snapshot executes the scanner protocol. Solutions to the snapshot problem are key tools in designing concurrent protocols.

Traditionally this problem is solved by means of *locking* — a processor who wishes to scan the memory first locks it so no other processor can write until the scan operation is completed. This approach is used in many database systems satisfactorily. From a theoretical point of view however, there is an interest in *nonwaiting* protocols. In these protocols no processor is required to wait for actions of another processor while executing its own scan or update protocol. Besides the theoretical interest nonwaiting protocols have a desirable fault tolerant property: Since no processor waits for any other processor, a halted processor cannot stop the execution of any other processor.

One should distinguish between the *multi-writer* problem and the *single-writer* problem: In the multi-writer problem each memory cell can be written into by all updaters while in the single-writer problem each cell can be written to by a single processor. The complexity of a nonwaiting solution to each of the snapshot problems is measured by several criteria. The two most important criteria are:

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 auxiliary hardware (not including the space in which the actual value is held) used by the protocols.

The wait-free snapshot problem was proposed and solved independently by Afek et al in [AAD90] and by Anderson in [A90]. In [A90] Anderson presents exponential-time protocols for the multi-writer snapshot problem. A polynomial solution for both single-writer and multi-writer snapshot problems was presented in [AAD90]. Later works dealt with the single writer problem: another polynomial solution was presented by Aspnes and Herlihy in [AH90]. A solution for a system with *one* scanner was presented by Kirousis, Spirakis and Tsigas in [KST91]. Recently a new solution was proposed by Attiya, Herlihy and Rachman in [AHR92]. In this paper the authors introduce the notion of *Lattice-Agreement* and show that the snapshot problem and the lattice agreement problem are equivalent for wait-free computation. Then they introduce a lattice agreement protocol whose time-complexity is linear using test and set registers for two writers and two readers. This last protocol induces a randomized solution for the snapshot problem using read write registers. The expected time complexity of this solution is $O(n \log^2 n)$. The time-complexity of all these solutions except the single scanner solution of [KST91] is super-linear. A linear-time protocol for a similar problem called the *Time-Lapse* snapshot problem is presented by Dwork, Herlihy, Plotkin and Waarts in [DHPW92]. This problem however is slightly weaker than the snapshot problem, since the snapshots it returns are allowed to be inconsistent.

A common conjecture states: *There exists a solution to the snapshot problem such that the time complexity of both* **update** *and* **scan** *protocols is linear.* In this paper we show that if the number of either scanners or updaters is smaller then the square root of the total number of processors then such a linear solution exists. This is done by presenting two methods of converting arbitrary solutions for the snapshot problem to other solutions in which one of the protocols has linear-time complexity: The first method converts an arbitrary solution to a solution whose scan protocol works in linear-time. The second method converts an arbitrary solution to a solution whose update protocol works in linear-time. The linear-time protocols for unbalanced systems are obtained by applying both methods to the protocol of [AAD90]. Both methods rely on time-stamps to keep total order among all update actions of every individual updater, hence the space complexity of the resulting solutions is unbounded. For the case of a system with more updaters than scanners we have a similar method that yields bounded solutions. Since this method is more complex we chose to present the unbounded method.

The rest of this paper is organized as follows: In Section 2 we present the computational model and the snapshot problem. The conversion methods are presented in Sections 3 and 4. In Section 5 we show how to derive linear-time protocols for unbalanced systems using our methods.

2 Model and Requirements

A system consists of a set of processing entities called *processors*, a set of memory entities called *registers* and a set of *operations*. Each processor has a unique *id*. Each register has a set of *permitted values*. One of the register's permitted values is a distinguished value called the register's *initial* value. Each operation is associated with a processor that can *execute* the operation and with a register which is *accessed* by the operation. Operations are partitioned into input operations and output operations. An *input* operation gets an input parameter while an *output* operation returns an output parameter. Each operation is executed *instantaneously*. A *system execution* is a sequence of operation executions (an execution of an operation is called an *action*) which satisfies some consistency requirements. Each action is said to happen at its *occurrence time* which is an indivisible instant. The consistency requirements depend on the individual system; as an example consider the set of operations which includes *read* operations from the system's registers and *write* operations to the system's registers. The consistency requirement for this system is: Each read action accessing register r returns the input parameter to the latest write action that accessed r (or r's initial value if no such write action occurs).

A *compound* system is defined using another sysytem called the *elementary system.* Both elementary and compound systems have the same sets of processors and registers. The set of *compound* operations (operations of the compound system) is defined in terms of *programs* of the elementary system. The building blocks of programs are *instructions*; each instruction starts with a nonempty sequence of internal computations which is succeeded by at most one elementary operation. Each program has a distinguished instruction called the program's *initial instruction.* If the program implements an input action then it takes an input parameter; if it implements an output action then it returns an output parameter. For simplicity we assume that each processor P has a single program called P's program.

A *program execution* is a sequence of actions determined by the program and by the parameters returned by the output actions. Each processor has a *program counter* which at any time points to the next instruction the processor is about to execute. When the compound system is initialized all program counters point to the processors' respective initial instructions, and each register holds its initial value. An *execution of a compound system* is an execution of the elementary system in which each processor repeatedly executes its program. A *schedule* is a sequence of processor *ids*; every schedule induces a compound system execution in which processors execute elementary actions one after the other in the order dictated by the schedule. It is important to note that elementary actions of different processors, each of which executing its own program, might be interleaved. The set of compound system executions is the set of executions induced by all possible schedules.

Our goal is to use the compound system as an elementary system for implementing yet another, more complex, compound system. Roughly speaking this goal requires that executions of the compound system can be viewed as elementary executions. In particular it is required that each compound action can be looked at as if it happens *instantaneously.* A *serialization* of a compound execution is an assignment of *serialization times* for each compound action. Serialization times are specified using the elementary actions, that is, an action might be serialized either at the occurrence time of some elementary action or in between the occurrence times of two consecutive elementary actions. Once an execution is serialized one can view every compound action as if it happens instantaneously at its serialization time and check whether the execution preserves the consistency requirements of the compound system. A serialization *scheme* for a compound system S is a function from the set of all executions of S that matches a serialization for every execution. A compound system is an *implementation* of an elementary system if it has a serialization scheme under which every execution satisfies the elementary system's consistency requirements. It should be noted that the elementary system implemented by the compound system is *different* from the elementary system by which the compound system was defined.

A *snapshot* system is a system with two operations called *update* and *scan.* Update operations are executed by w processors called *updaters*. Each updater U_i has a register called R_i, an update operation executed by U_i stores a value in R_i. A *snapshot* at time t, is the vector of values stored in $R_1 \dots R_w$ at t. A *scan* operation is an output operation that returns the snapshot at its occurrence time. A program is *wait-free* if all its executions consist of a bounded number of elementary actions, where the bound may depend on the number of processors in the system. The *execution interval* of a compound action is the time interval that starts with the occurrence time of the action's first elementary action and ends with the occurrence time of its last elementary action. In this paper we present implementations of snapshot systems. To avoid trivial implementation we require that each compound operation is implemented by a wait-free program and that the serialization time of each action falls within its execution interval.

3 Solutions with Linear Scan Protocols

In this section we describe a method to convert an arbitrary solution to the snapshot problem to another solution with a *linear-time* scan protocol. The requirements from the protocols of the original solution ensure that we can use them as elementary operations in the snapshot system we present. The original solution and its two protocols are called the *elementary* solution and *elementary* protocols, respectively. The underlying idea is that the updaters execute the scan for the scanners, using the elementary scan protocol. The result of each such elementary scan is a snapshot and all snapshots are ordered temporally by time-stamps. The scanner collects a 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 es(can) and eu(pdate) respectively. Each updater, U_i, keeps an internal variable $count_i$ which is initialized to zero and incremented by 1 every time P_i executes an update operation. The *data* field of the elementary protocol is replaced with a record (*data, count*) where $count_i$ is written in the count field in every execution of the elementary update protocol. The new update protocol for U_i consists of an elementary update operation which is followed by an elementary scan operation. The snapshot obtained by this elementary scan is written atomically into an additional $(1, r)$ register called r_i which can be read by all the scanners. The ath update action of U_i is denoted by U_i^a; the *value* it gets as input is denoted by u_i^a; Its elementary update, elementary scan and the atomic write are denoted by eu_i^a, es_i^a and ew_i^a respectively. The sum associated with a snapshot is the sum of its *count* fields. The snapshot returned by es_i^a is denoted by ev_i^a, its sum is denoted by $sum(ev_i^a)$. The new scan protocol works as follows: first each r_i is read to obtain a snapshot from each updater U_i and then the snapshot whose *sum* is largest is returned. The bth scan operation executed by S_j is denoted by S_j^b; its subactions are denoted by $r_j^b[1], \dots r_j^b[w]$ and the snapshot it returns is denoted by s_j^b. 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 w, that is *linear* in the number of writers.

3.2 Serialization Scheme

In the serialization scheme for the protocol we slightly modify the serialization requirement and allow an action to be serialized a short time before its starting time or after its ending time, where short time is defined relative to the interval between any two consecutive elementary actions. (In this paper actions are serialized only after the end of actions). Alternatively we could have added two dummy elementary operations to each program, one before the first "real" elementary operation and the other after the last one. For any action a, the serialization time of a is denoted by $t(a)$, while its starting and ending times are denoted by $start(a)$ and $end(a)$, respectively. Let a and b be two elementary actions, we say that a *sees* b if a is serialized after b.

The occurrence times of the elementary actions are used to define the serialization for the new solution. In this serialization update operations are serialized independently of scans, while scan

$Update_i(value)$
  $count_i := count_i + 1$
  $eu(value, count_i)$ $\qquad eu_i^a$
  $s[1 \ldots n] := es$ $\qquad es_i^a$
  $r_i :=$ **write** $(s[1..n])$ $\qquad ew_i^a$

$Scan_i$
  for $j := 1$ **to** w **read** (r_j) $\qquad r_k^c[1] \ldots r_k^c[w]$
  return snapshot with maximum sum

FIGURE 1. The Protocols for Few Updaters

operations take the serialization of updates into account, therefore we first define the serialization for updates:

DEFINITION 1: The serialization time for update action U_i^a is defined to be just after the serialization time of the earliest *ewrite* ew_k^c whose corresponding *escan* es_k^c sees eu_i^a.

$$t(U_i^a) = \min_{j,b} \{t(ew_j^b) : t(es_j^b) > t(eu_i^a)\} + \varepsilon$$

The constant ε is defined to be small relative to the time interval between any consecutive elementary operations.

In this case we say that U_i^a is *serialized by* ew_k^c. If several update operations are serialized by the same *ewrite* operation, they are further serialized in the order of their own *eupdates*, in this way no two updates are serialized at the same time. Nevertheless, we will not address the differences in their serialization times and regard all these update actions as if they are serialized at the same time.

DEFINITION 2: The serialization time of a scan operation S_j^b is defined to be just after the maximum between its starting time and the serialization time of the latest update action whose value is included in s_j^b.

$$t(S_j^b) = \max\ \{start(S_j^b),\ \max_{k,c} \{t(U_k^c) : u_k^c \in s_j^b\}\} + \varepsilon$$

In case the serialization time of S_j^b is determined by update operation U_ℓ^d we say that S_j^b is serialized by U_ℓ^d. By this definition a scan S_j^b that returns snapshot ev_ℓ^d may be serialized *before* ew_ℓ^d.

3.3 Correctness Proof

LEMMA 1: Let es_i^a and es_j^b be two *escan* actions. If $sum(ev_i^a) \geq sum(ev_j^b)$, then every eupdate eu_k^c satisfying $t(eu_k^c) < t(es_j^b)$ also satisfies $t(eu_k^c) < t(es_i^a)$.

Proof: If $sum(ev_i^a) > sum(ev_j^b)$ then the fact that the value of every *count* field is monotonically increasing and the atomicity of the elementary protocols imply that $t(es_i^a) > t(es_j^b)$ and the lemma follows immediately. If $sum(ev_i^a) = sum(ev_j^b)$ then it is possible that $t(es_i^a) < t(es_j^b)$ but it is not hard to see that no eupdate action occurs in between and the lemma follows once more. □

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

LEMMA 2: Every compound action is serialized within its execution interval.

Proof: We first prove the lemma for update action: Let U_i^a be an arbitrary update. By Definition 1 we have : $t(eu_i^a) < t(U_i^a) \leq t(ew_i^a) + \varepsilon = end(U_i^a) + \varepsilon$. The lemma follows.

We now prove the lemma for scan actions: Let S_j^b be an arbitrary scan action, the lemma holds trivially if S_j^b is serialized at its starting time. Assume that S_j^b is serialized by update operation U_k^c and let ev_ℓ^d be the elementary snapshot returned by S_j^b. Since S_j^b returns ev_ℓ^d we get that $t(ew_\ell^d) < t(r_j^b[\ell]) \leq end(S_j^b)$. Since $u_k^c \in s_j^b$ it holds that eu_k^c is seen by es_ℓ^d. By Definition 1 we have : $t(U_k^c) \leq t(ew_\ell^d) + \varepsilon$. Since S_j^b is serialized by U_k^c it holds that $t(S_j^b) = t(U_k^c) + \varepsilon$. Combining these inequalities we get $t(S_j^b) = t(U_k^c) + \varepsilon \leq t(ew_\ell^d) + 2\varepsilon < end(S_j^b) + 2\varepsilon$. The lemma follows. □

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

LEMMA 3: If S_i^a is serialized at t then s_i^a is the snapshot at t.

Proof: It should be noted that since update actions are not serialized at the time of their corresponding eupdate actions the fact that s_i^a is a snapshot does not follow immediately from the correctness of the elementary solution. Definition 2 ensures that a scan does not return any value whose update is serialized after that scan; to prove the lemma we have to show that all values whose update is serialized before a scan are considered by that scan, since in this case the ordering of the update actions ensures that the most recent value of each updater is returned. Let S_i^a be an arbitrary scan action, let U_m^e be the update that is serialized last among the updates included in s_i^a, and let ev_k^c be the elementary snapshot returned by S_i^a. We consider two cases according to the way S_i^a is serialized:

Case 1: Action S_i^a is serialized at $start(S_i^a)$. In this case we prove that the value of every update that is serialized before the beginning of S_i^a (or a later value of the same updater) is included in ev_k^c. Assume by way of contradiction that U_j^b is an update serialized before $start(S_i^a)$ and its value (or a later value) is not included in ev_k^c and let ew_ℓ^d be the *ewrite* action by which U_j^b is serialized. It follows that $t(ew_\ell^d) < start(S_i^a)$ hence S_i^a collects ev_ℓ^d (or a later elementary snapshot executed by U_ℓ). Since S_i^a collects both ev_ℓ^d and ev_k^c and chooses ev_k^c we conclude that $sum(ev_\ell^d) \leq sum(ev_k^c)$. From Lemma 1 we have that eu_j^b (or a later value of U_j) is included in ev_k^c, a contradiction.

Case 2: Action S_i^a is serialized by U_m^e. In this case $t(S_i^a) = t(ew_p^f) + 2\varepsilon$, where U_m^e is serialized by ew_p^f. We prove that the elementary update of every update that is serialized before ew_p^f is serialized before es_k^c. As a result, we get that every update serialized before S_i^a is considered by es_k^c. Assume by way of contradiction that U_j^b is an update that is serialized before ew_p^f but its elementary update is serialized after es_k^c. Let ew_ℓ^d, $(\ell, d) \neq (p, f)$, be the *ewrite* action by which U_j^b is serialized. Since eu_j^b is serialized before es_ℓ^d and after es_k^c, it holds that $t(es_\ell^d) > t(es_k^c)$. Therefore es_ℓ^d is serialized after eu_m^e . Since U_j^b is serialized by ew_ℓ^d before S_i^a, it follows that $t(ew_\ell^d) < t(ew_p^f)$. Therefore, U_m^e is serialized by ew_ℓ^d, a contradiction. Note that the above proof does not include updates serialized by ew_p^f. Indeed, it is possible for some of the updates serialized by ew_p^f not to be included in es_k^c. However, this case can be easily handled by serializing the scan between such two updates since updates serialized by the same atomic write are further serialized by their elementary update times.

□

4 SOLUTIONS WITH LINEAR UPDATE PROTOCOLS

In this section we describe a method to convert an arbitrary solution (to which we once more refer as the *elementary* solution) to the snapshot problem to another solution with a *linear-time* update protocol. The underlying idea is to modify the single scanner protocol of [KST91]. The scanners use the elementary solution to reach a lattice agreement (see [AHR92]).

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

FIGURE 2. Protocol for U_i

4.1 Description

The Protocols appear in Figures 2 and 3. Once more every value is associated with a number called its *count*. Updater U_i keeps an internal variable $count_i$ which is initialized to 0 and is incremented at the beginning of every update action. The register of U_i consists of an array of w entries called $view_i$ in which each entry is a pair of the form $(value, count)$. The k-th 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*. After that U_i assigns its new $(value, count)$ pair to $lview[i]$ and then *lview* is atomically written into $view_i$. The elementary actions executed during U_i^a are denoted by $r_i^a[1]$ ... $r_i^a[w]$, w_i^a. The value and the view written during U_i^a are denoted by u_i^a and v_i^a, respectively. The complexity of the update protocol is w, that is, *linear* in the number of writers.

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 executes an *eupdate* operation in which its local view is written to $view_j$ (assuming S_j^b is executed) followed by an *escan* operation on the views of all scanners. Following the elementary scan operation the local view is once more updated as before. At this point $lview_j$ holds a snapshot which is returned. The elementary actions executed during S_j^b are denoted by $r_j^b[1]$ ... $r_j^b[w]$, 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 v_j^b and s_j^b respectively. The *count* field of U_k in v_j^b is denoted by $v_j^b[k].count$. 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 Serialization Scheme

As in the previous solution we use the occurrence times of the elementary actions to define the serialization for the new solution. We start with defining serialization time for scan operations which are serialized independently of updates:

DEFINITION 3: The serialization time of S_i^a is defined as follows: Let t_1 be the occurrence time of es_i^a and let t_2 denote the occurrence time of the first elementary write action w_j^b, $j \leq w$, for which b is larger then the count of $s_i^a[j]$. Action S_i^a is serialized at $min(t_1, t_2 - \varepsilon)$. where ε is defined once more to be small relative to the time interval between any consecutive elementary operations. In case $t_2 < t_1$ we say that S_i^a is serialized by w_j^b. Note that no two scans of the same processor can be serialized by the same *ewrite* action. We say that view v_i *dominates* view v_j if for all k, $1 \leq k \leq w$,

```
begin
  for k := 1 to w
    temp := read (view_k)                    r_j^b[k]
    for m := 1 to w
      if temp[m].count > lview[m].count then lview[m] := temp[m]
    endfor
  endfor
  eu(lview)                                  ew_j^b
  scan := es                                 es_j^b
  for k := 1 to r
    for m := 1 to w
      if scan[k][m].count > lview[m].count then lview[m] := scan[k][m]
    endfor
  endfor
  Return (lview)
end
```

FIGURE 3. Protocol for S_j

$v_i[k].count \geq v_j[k].count$. In Lemma 6 below we show that if two views are not equal then one of them dominates the other. Thus if two scan actions are serialized by the same *ewrite* action then they are further serialized by domination order of the views they return as snapshots where the action whose snapshot is dominated is serialized first. If the snapshots are equal then the operations are serialized an ascending order of their *ids*.

DEFINITION 4: The serialization time of update action U_j^b is defined as follows: Let t_1 be the occurrence time of w_j^b and let t_2 denote the serialization time of the first scan action S_k^c which returns u_j^b. Action U_j^b is serialized at $min(t_1, t_2 - \varepsilon)$. In case $t_2 < t_1$ we say that U_j^b is serialized by S_k^c. If two updates are serialized by the same *escan* then they are further serialized by an ascending order of their *ids*.

4.3 Correctness Proof

Two values u_j^b and u_k^c are *inconsistent* if there exists no snapshot consisting of both values, that is U_j^{b+1} ends before U_k^c starts. A view v is *consistent* if it contains no inconsistent values. First we show that each view returned by the scanners is consistent.

LEMMA 4: Every view v returned by a scanner is consistent.

Proof: Assume by way of contradiction that v is inconsistent and let u_j^b and u_k^c be the inconsistent pair of values, where U_j^{b+1} ends before U_k^c starts. By the definition of inconsistent pair we get that $t(w_j^{b+1}) < t(r_k^c[j]) < t(w_k^c)$. Therefore u_j^{b+1} (or a later value of U_j) appears in v_k^c (the view written at the end of U_k^c). Hence every view which includes u_k^c includes u_j^{b+1} or a later value of U_j, a contradiction. □

LEMMA 5: Every view written by updater U_i dominates all previous views written by U_i; the same holds for ever view that is *eupdated* by scanner S_j.

Proof: We prove the lemma for updaters only. The proof for scanners is similar. Assume by way of contradiction that the lemma does not hold. Let w_i^a be the *first ewrite* action, according to the total order on the elementary execution, that contradicts the lemma. It follows that there exists $j \neq i$ such that $v_i^a[j].count < v_i^{a-1}[j].count$. Let U_k be the updater from which the $(value, count)$ pair of U_j in v_i^{a-1} was taken. The count of U_j read in $r_i^a[k]$ is smaller then the count of U_j read in $r_i^{a-1}[k]$. Since $t(r_i^{a-1}[k]) < t(r_i^a[k])$ this is a contradiction to the minimality of w_i^a. □

Two views v_p and v_q are *contradictory* if u_m^d and u_k^c are values in v_p while u_k^{c+r} and u_m^{d-s}, $r, s > 0$, are values in v_q. Two contradictory views cannot be snapshots under the same serialization.

LEMMA 6: If v_i^a and v_j^b are views returned as snapshots (where i is not necessarily different from j) then they are not contradictory.

Proof: Assume by way of contradiction that u_m^d and u_k^c are values in v_i^a while u_m^{d+r} and u_k^{c-s}, $r, s > 0$, are values in v_j^b. Without loss of generality assume that $t(es_i^a) < t(es_j^b)$. Hence Lemma 5 implies that at least one of the indices of U_k returned by es_j^b (there are r such indices) is at least c, a contradiction. □

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

LEMMA 7: Every action is serialized within its execution interval.

Proof: By Definition 3 and Definition 4 every scan and update are serialized no later then their last elementary action. We have to show that they are not serialized before their first elementary read action. We start with scan actions. Let S_i^a be an arbitrary scan action, if S_i^a is serialized at es_i^a then we are done. Assume S_i^a is serialized by w_j^b. In this case $s_i^a[j].count < b$, hence $start(S_i^a) < t(w_j^b)$, otherwise in $r_i^a[j]$, S_i reads u_j^b.

We continue with update actions: Assume by way of contradiction that U_i^a is serialized before $r_i^a[1]$. In this case there is a scan action S_j^b which returns u_i^a, and S_j^b is serialized before $r_i^a[1]$. The value u_i^a is written for the first time in action w_i^a, since it is included in s_j^b we can conclude that $t(w_i^a) < t(es_j^b)$ hence S_j^b is not serialized at es_j^b. Let w_k^c be the elementary write action by which S_j^b is serialized, by Definition 3 w_k^c occurs before $r_i^a[1]$. Therefore u_k^c belongs to v_i^a. Since u_i^a appears in the view of S_j^b, u_k^c should be in the view of S_j^b as well, a contradiction to the definition of U_k^c. □

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

LEMMA 8: If S_i^a is serialized at t then s_i^a is the snapshot at t.

Proof: We show that if u_k^c belongs to s_i^a then $t(U_k^c) < t(S_i^a) < t(U_k^{c+1})$. Clearly $t(U_k^c) < t(S_i^a)$; assume by way of contradiction that $t(U_k^{c+1}) < t(S_i^a)$. By Definition 3 we get that $t(S_i^a) < t(w_k^{c+1})$, thus $t(U_k^{c+1}) < t(w_k^{c+1})$. Definition 4 implies that U_k^{c+1} is serialized by some scan action S_j^b where $s_j^b[k].count = c+1$. Since $s_i^a[k].count = c$ Lemma 6 implies that s_j^b dominates s_i^a. Since S_j^b returns the value of u_k^{c+1} it is clear that $t(w_k^{c+1}) < t(es_j^b)$. Hence S_j^b is not serialized at es_j^b but by some elementary write action w_m^d where the count of $s_j^b[m]$ is smaller then d. Since s_j^b dominates s_i^a we get that the count of $s_i^a[m]$ is also smaller then d. Therefore S_i^a should be serialized also by U_m^d. Since $u_k^{c+1} \in s_i^a$ we get that v_i^a is dominated by v_j^b, which implies that $t(S_i^a) < t(S_j^b)$. Since U_k^{c+1} is serialized by S_j^b we get $t(S_i^a) < t(U_k^{c+1}) < t(S_j^b)$, contradiction. □

5 LINEAR SOLUTIONS

A snapshot system is *unbalanced* if the number of either updaters or scanners is not greater then the square root of the total number of processors. We are now ready to prove the existence of linear-time protocols for unbalanced snapshot systems:

THEOREM 9: There exist linear time snapshot solutions for every unbalanced system.

Proof: Using the solution of Afek et al in [AAD90] as the elementary solution we obtain linear solutions for unbalanced systems: Recall that the complexity of both protocols in this solution is $O(n^2)$ where $n = w = r$. If $w \leq \sqrt{r}$ then the first solution yields an update protocol whose complexity is the sum of the complexities of the basic protocols, that is $O(w^2) = O(n)$ while the complexity of the scan protocol is w. If $r \leq \sqrt{w}$ then the second solution yields an update protocol whose complexity is w, while the complexity of the scan protocol is $O(r^2) = O(n)$. □

Using our first method we improve the protocol of [AAD90] with no dependence on the ratio between updaters and scanners[1] as follows: It is not hard to see that the real requirement from the updater protocol is that a scan is executed between every two update actions (and not necessarily before the update). Therefore, our first method can be used as follows: The update protocol begins with writing the $(value, count)$ pair, continues with the scan protocol of [AAD90], and ends by writing its results; the scan protocol is now replaced by ours. The complexities of the protocols obtained from applying our methods to the solution of [AAD90] are not comparable. The first method yields a solution whose complexity is $O(w^2)$ for an update operation and w for a scan operation. The second method yields a solution whose complexity is w for an update operation and $w + O(r^2)$ for a scan operation.

ACKNOWLEDGEMENTS

We are grateful to Jap Henk Hoepman for his carful reading of the manuscript.

REFERENCES

[A90] J. Anderson, Composite Registers, *Proceedings of the 9th Annual ACM Sympossium on Principles of Distributed Computing*, 1990, pp. 15-29.

[AAD90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic Snapshots of Shared Memory, *Proceedings of the 9th Annual ACM Sympossium on Principles of Distributed Computing*, 1990, pp. 1-13.

[AH90] J. Aspned and M. Herlihy, Wait-free Data Structures in the Asynchronous PRAAM Model, *Proceedings of the 2nd Annual Symposium on Paarallel Algorithms and Architectures*, 1990, pp. 340-349.

[AHR92] H. Attiya, M. Herlihy amd O. Rachman, "Efficient Atomic Snapshots Using Lattice Agreement," preprint.

[DHPW92] C. Dwork M. Herlihy, S. Plotkin, O. Waarts, Time-Lapse Snapshots, *Proceedings of the 1st Israeli Symposium on Theory of Computing and Systems*, Haifa Israel May 1992, (D. Dolev Z. Galil and M. Rodeh, eds.) pp. 154-170,Lecture Notes in Computer Science #601, Springer-Verlag, 1992.

[KST91] 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 Workshopn on Distributed Algorithms and Graphs*, Delphy, Greece, October 1991, (S. Toueg, P. Spirakis and L. Kirousis, eds.) pp. 229-241, Lecture Notes in Computer Science #579, Springer-Verlag, 1992.

[1] If the number of updaters is $O(n)$ then the improvement of the updaters protocol is only by a constant.