April 1992 Report No. STAN-G-92-1423

Time-Lapse Snapshots

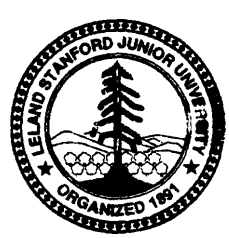
by

C. Dwork, M. Herlihy, S. Plotkin, 0. Waarts

Department of Computer Science

Stanford University

Stanford, California 94305



Time-Lapse Snapshots

Cynthia *Dwork*
IBM Almaden Research Center

Maurice *Herlihy*
DEC Cambridge Research Lab.

Serge ***A. Plotkin****
Stanford University

Orli Waarts**
Stanford University

Abstract. A snapshot scan algorithm takes an "instantaneous" picture of a region of shared memory that may be updated by concurrent processes. Many complex shared memory algorithms can be greatly simplified by structuring them around the snapshot scan abstraction. Unfortunately, the substantial decrease in conceptual complexity is quite often counterbalanced by an increase in computational complexity.

In this paper, we introduce the notion of a ***weak snapshot*** scan, a slightly weaker primitive that has a more efficient implementation. We propose the following methodology for using this abstraction: first, design and verify an algorithm using the more powerful snapshot scan, and second, replace the more powerful but less efficient snapshot with the weaker but more efficient snapshot, and show that the weaker abstraction nevertheless suffices to ensure the correctness of the enclosing algorithm.

We give two examples of algorithms whose performance can be enhanced while retaining a simple modular structure: bounded concurrent timestamping, and bounded randomized consensus. The resulting timestamping protocol is the fastest known bounded concurrent timestamping protocol. The resulting randomized consensus protocol matches the computational complexity of the best known protocol that uses only bounded values.

1 Introduction

Synchronization algorithms for shared-memory multiprocessors are notoriously difficult to understand and to prove correct. Recently, however, researchers have identified several powerful abstractions that greatly simplify the conceptual complexity of many shared-memory algorithms. One of the most powerful of these is ***atomic snapshot scan*** (in this paper we sometimes omit the word "scan"). Informally, this is a procedure that makes an "instantaneous" copy of memory that is being updated by concurrent processes. More precisely, the

• Research supported by NSF Research Initiation Award CCR-9008226, by U.S. Army Research Office Grant DAAL-03-91-G-0102, by ONR Contract N00014–88–K–0166, and by a grant from Mitsubishi Electric Laboratories.

• * Research supported by NSF grant CCR-8814921, U.S. Army Research Office Grant DAAL-03-91-G-0102, ONR contract N00014-88-K-0166, and IBM fellowship.

problem is defined as follows. A set of n asynchronous processes share an n-element array A, where P is the only process that writes $A[P]$[3]. An atomic snapshot is a read of all the elements in the array that appears to occur instantaneously. Formally, scans and updates are required to be *linearizable* [20], i.e. each operation appears to take effect instantaneously at some point between the operation's invocation and response.

Atomic snapshot scan algorithms have been constructed by Anderson [3] (bounded registers and exponential running time), Aspnes and Herlihy [6] (unbounded registers and $O(n^2)$ running time), and by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [2] (bounded registers and $O(n^2)$ running time). Chandy and Lamport [13] considered a closely related problem in the message-passing model.

Unfortunately, the substantial decrease in conceptual complexity provided by atomic snapshot scan is often counterbalanced by an increase in computational complexity. In this paper, we introduce the notion of a weak *snapshot scan, a* slightly weaker abstraction than the atomic snapshot scan. The advantage of using weak snapshot is that it can be implemented in O(n) time. Thus, the cost of our weak snapshot scan is asymptotically the same as the cost of a simple "collect" of the n values. Our primitive, however, is much more powerful. Moreover, the best known atomic snapshot requires atomic registers of size $O(nv)$, where v is the maximum number of bits in any element in the array A. In contrast, our weak snapshot requires registers of size $O(n + v)$ only.

Our results indiite that weak snapshot scan can sometiis alleviate the trade-off between conceptual and computational complexity. We focus on two well-studied problems: bounded concurrent timestamping and randomized consensus. In particular, we consider algorithms for these problems based on an atomic snapshot. In both cases, we show that one can simply replace the atomic snapshot scan with a weak snapshot scan, thus retaining the algorithms' structure while improving their performance.

The weak snapshot algorithm presented here was influenced by work of Kirousis, Spirakis, and Tsigas [23], who designed a linear-time atomic snapshot algorithm for a *single* scanner. In this special case our algorithm solves the original atomic snapshot problem as well.

One important application of snapshots *is* to *bounded concurrent timestamping,* in which processes repeatedly choose labels, or timestamps, reflectii the real-time order of events. More specifically, in a concurrent timestamping system processes can repeatedly perform two types of operations. The first is a *Label* operation in which a process assigns itself a new label; the second is a *Scan* operation, in which a process obtains a set of current labels, one per process, and determines a total order on these labels that is consistent with the real tie order of their corresponding Label operations.[4]

Israeli and Li [21] were the first to investigate bounded sequential timestamp systems, and Dolev and Shavit [17] were the first to explore *the concurrent* version of the problem. The Dolev-Shavit construction requires O(n)-sized registers and labels, O(n) time for a label operation, and $O(n^2 \log n)$ time for a scan. In their algorithm each processor is assigned a single multi-reader single-writer register of $O(n)$ bits. Extending the Dolev-Shavit solution in a non-trivial way, Israeli and Pinhasov [22] obtained a bounded concurrent timestamp system that is linear in time and label size, but uses registers of size $O(n^2)$. An alternative

[3] One can also define multi-writer algorithms in which any process can write to any location.

[4] Observe that the scan required by the timestamping is not necessarily identical to the atomic snapshot scan. Unfortunately, the two operations have the same name in the literature.

implementation of their algorithm uses single-reader-single-writer registers[5] of size $O(n)$, but requires $O(n^2)$ time to perform a *Scan.* Later, Dwork and Waarts [18] obtained a completely different linear-time solution, not based on any of the previous solutions, with a simpler proof of correctness. The drawback of their construction is that it requires registers and labels of size $O(n \log n)$.

Dolev and Shavit observed that the *conceptual* complexity of their concurrent timestamping algorithm can be reduced by using atomic snapshot scan. We show that, in addition, the algorithm's *computational* complexity can be reduced by simply replacing the snapshot scan with the weak snapshot, making no other changes to the original algorithm. The resulting bounded concurrent timestamping algorithm is linear in both time and the size of registers and labels, and is conceptually simpler than the Dolev-Shavit and Israeli-Pinhasov solutions.

Another important application of atomic snapshots is randomized *consensus:* each of n asynchronous processes starts with an input value taken from a two-element set, and runs until it chooses a *decision value* and halts. The protocol must *be consistent:* no two processes choose different decision values; *valid:* the decision value is some process' preference; and randomized wait-free: each process decides after a finite expected number of steps. The consensus problem lies at the heart of the more general problem of constructing highly concurrent data structures [19]. Consensus has no deterministic solution in asynchronous shared-memory [16]. Nevertheless, it *can be* solved *by randomized* protocols in which each process is guaranteed to decide after a finite *expected* number of steps. Randomized consensus protocols that use unbounded registers have been proposed by Chor, Israeli, and Li [14] (against a "weak" adversary), by Abrahamson [1] (exponential running time), by Aspnes and Herlihy [7] (the first polynomial algorithm), by Saks, Shavit, and Woll [27] (optimized for the case where processes run in lock step), and by Bracha and Rachman [11] (running time $O(n^2 \log n)$).

Protocols that use bounded registers have been proposed by Attiia, Dolev, and Shavit [8] (running time $O(n^3)$), by Aspnes [5] (running time $O(n^2(p^2 + n))$, where p *is the* number of active processors), and by Bracha and Rachman [10] (running time $O(n(p^2 + n))$. *The* bottleneck in Aspnes' algorithm is atomic snapshot. Replacing thii atomic snapshot with our more efficient weak snapshot improves the running time by $Q(n)$ (from $O(n^2(p^2 + n))$ to $O(n(p^2 + n))$), and yields a protocol that matches the fastest known randomized consensus algorithm that uses only bounded registers, due to Bracha and Rachman [10]. Both our consensus algorithm and the one in [10] are based on Aspnes' algorithm. The main difference is that the solution of Bracha and Rachman is specific to consensus, whereas our algorithm is an immediate application of the primitive developed in this paper.

The remainder of thii paper is organized as follows. Section 2 gives our model of computation and defines the weak snapshot primitive. Some properties of weak snapshots appear in Section 3. The remaining sections describe the weak snapshot algorithm and its applications.

2 Model and Deflnitions

A *concurrent system* consists of a collection of n asynchronous processes that communicate through an initialized shared memory. Each memory location, called a *register, can be* written by one "owner" process and read by any process. Reads and writes to shared registers are assumed to *be atomic,* that is, they can be viewed as occurring at a single instant of time. In order to be consistent with the literature on the discussed problems, our time and space

[5] All other results mentioned are in terms of multi-reader-single-writer registers.

complexity measures are expressed in terms of read and write operations on single-writer multi-reader registers of size $O(n)$. Polynomial-tii algorithms for implementing large single-writer/multi-reader atomic registers from small, weaker, registers are well known [12, 24, 25, 26].

An algorithm is wait-free if there is an *a* priori upper bound on the number of steps a process might take when running the algorithm, regardless of how its steps are interleaved with those of other processes. All algorithms discussed in this paper are wait-free.

An atomic snapshot *memory* supports two kinds of abstract operations: Update modifies a location in the shared array, and Scan instantaneously reads (makes a copy of) the entire array. Let U_i^k (S_i^k) denote the *k*th Update *(Scan)* of process i, and v_i^k the value written by i during U_i^k. The superscripts are omitted where it can not cause confusion. An operation A precedes operation B, written as "$A \longrightarrow B$", if B starts after A finishes. Operations unrelated by precedence are concurrent. Processes are sequential: each process starts a new operation only when its previous operation has finished, hence its operations are totally ordered by precedence.

Correctness of an atomic snapshot memory is defined as follows. There exists a total order "$\Longrightarrow$" on operations such that:

- If $A \longrightarrow B$ then $A \Longrightarrow B$.
- If *Scan* S_p returns $\bar{v} = \langle v_1, \ldots, \text{v,}\rangle$, then v_q is the value written by the latest *Update* U_q ordered before S_p by $\Longrightarrow$.

The order "$\Longrightarrow$" is called the linearization order [20]. Intuitively, the first condition says that the linearization order respects the "real-time" precedence order, and the second says that each correct concurrent computation is equivalent to some sequential computation where the scan returns the last value written by each process.

We define a weak snapshot as follows: we impose the same two conditions, but we allow "$\Longrightarrow$" to be a partial *order*[6] rather than a total order. We call this order a *partial* linearization order. If $A \Longrightarrow B$ *we say* that *B observes A.*

This weaker notion of correctness allows two scans S and S' to disagree on the order of two *Updates* U and U'; but only if all four operations are concurrent with one another. Scanning processes must agree about *Updates* that happened "in the past " but may disagree about concurrent updates. Thus, in a system with only one scanner, atomic snapshots and weak snapshots are equivalent. Similarly, the two types of snapshots are equivalent if no two updates occur concurrently.

3 Properties of Weak Snapshots

The reader can easily verify that weak snapshots satisfy the following axioms:

- Regularity: For any value v_a^i returned by S_b^j, U_a^i begins before S_b^j terminates, and there is no U_a^k such that $U_a^i \longrightarrow U_a^k \longrightarrow S_b^j$.
- **Monotonicity of Scans**: If S_a^i and S_b^j are two scans satisfying $S_a^i \longrightarrow S_b^j$, (a and b could be the same process), and if S_a^i observes update U_c^k (formally, $U_c^k \Longrightarrow S_a^i$), then S_b^j *observes* U_c^k.

[6] In this paper, aii partial orders are irreflexive.

- **Monotonicity of *Updates***: If U_a^i and U_b^j are two Update operations (possibly by the same process), such that $U_a^i \longrightarrow U_b^j$, and if S_c^k is a ***Scan*** operation, possibly concurrent with both U_a^i and U_b^j, such that S_c^k observes U_b^j ($U_b^j \Longrightarrow S_c^k$), then S_c^k observes U_a^i.

Roughly speaking, weak snapshots satisfy all the properties of atomic snapshots except for the consistency property which states: If ***Scans*** S_a^i, S_b^j return $\bar{v} = \langle v_1, \ldots \text{v,}\rangle$ and 6' = $\langle v_1' \ldots v_n' \rangle$, respectively, then either $U_k \not\longrightarrow U_k'$ for **every** $k = 1, \ldots, n$, or vice versa.

Define the ***span*** of a value v_p^i to be the interval from the start of U_p^i to the end of U_p^{i+1}. Clearly, values written by successive ***Updates*** have overlapping spans. The following lemma formalizes the intuition that a weak snapshot scan returns a possibly existing state of the system.

Lemma 1. *If a weak snapshot scan returns a set of values 6, then their spans have a non-empty intersection.*

Proof: Let v_p^i and v_q^j be in $\bar{v}$ such that the span of v_p^i is the latest to start and the span of v_q^j is the first to end. Then, it is enough to show that the spans of v_p^i and v_q^j intersect. Suppose not. Then $U_q^j \longrightarrow U_p^i$. By the definition of span, $U_q^{j+1} \longrightarrow U_p^i$, and hence $U_q^{j+1} \Longrightarrow U_p^i$, which violates the requirement that each ***Scan*** return the latest value written by the latest ***Update*** ordered before it by $\Longrightarrow$. ■

Let a ***Scan*** S of a weak snapshot start at time t_s, end at time t_e, and return a set of values $\bar{v}$. Lemma 1 implies that there is a point t in which the spans of all these values intersect. There may be more than one such point; however, the Regularity property of weak snapshots and Lemma 1 imply that there is at least one such point t such that $t_s \leq t \leq t_e$. This is because the first clause in the definition of regularity implies that the span of v_a^i begins before t_e, while the second clause implies that the span of v_a^i ends after t_s. We will refer to the latest such point t by t_{scan} of S.

4 Weak Snapshot

Intuitively, in order to be able to impose partial order on the scans and updates, we need to ensure that a scan that did not return value v_a^j of processor a because v_a^j is too new, will not return a value v_b^i that was written by processor b ***after b saw*** v_a^j. ***By the*** properties of weak snapshot, if the scan returns v_b^i, then it must be ordered after b's update in the partial order. Since this update has to be ordered after a's update, we have that a's update has to be ordered before the scan. This contradicts the assumption that the scan saw neither v_a^j nor any later **update** by a.

If each value returned by the ***Scan*** is the value written by the latest update that terminated before a specific point in the ***Scan,*** the above situation does not occur. Thii observation, due to Kirousis, Spirakis, and Tsigas **[23],** motivates our solution. Roughly speaking, in our solution, at the start of a scan, the scanner produces a new number, called ***color,*** for each other process. When a process wants to perform an update, it reads the colors produced for it (one color by each scanner) and tags its new value with these colors. Thii enables the scanner to distinguish older values from newer ones.

The next subsection describes a solution that uses an unbounded number of colors. Later we will show how to simulate this solution using only a bounded number of colors. The

simulation uses a simplification of the Traceable Use abstraction defined by Dwork and Waarts in [18].

4.1 Unbounded Weak Snapshot

We follow the convention that shared registers appear in upper-case and private variables in lower-case. In order to simplify the presentation, we assume that ail the private variables are persistent. If a variable is subscripted, the first subscript indicates the unique process that writes it, and the second, if present, indicates the process that uses it. Each process b has variables VALUE_b, which stores b's current value, PCOLOR_b, QCOLOR_b, each of which stores an n-element array of colors, and VASIDE_{bc}, for each $c \neq b$. We frequently refer to $\text{PCOLOR}_b[c]$ as PCOLOR_{bc} (analogously for $\text{QCOLOR}_b[c]$). In this section, we assume that all these variables are stored in a single register. Section 4.4 describes how to eliminate this assumption. The code for the Update and Scan operations appears in Figures 1 and 2, respectively; the code for the Produce operation, called by Scan, appears in Figure 3. At the start of a scan, the scanner b produces a new color for each updater c and stores it in PCOLOR_{bc}. It then reads VALUE_c, VASIDE_{cb}, and QCOLOR_{cb} atomically. If QCOLOR_{cb} is equal to the color produced by b for c (and stored in PCOLOR_{bc}), b takes VASIDE_{cb} as the value for c, Otherwise b takes VALUE_c.

The updater b first reads PCOLOR_{cb} and then writes its new VALUE_b atomically with $\text{QCOLOR}_{bc} := \text{PCOLOR}_{cb}$ for all c. At the same time it updates VASIDE_{bc} for all c that the updater detects have started to execute a concurrent Scan.

The intuition behind the use of the VASIDE variable can be best described if we will consider an example where we have a "fast" updater b and a "slow" scanner c, where c executes a single ***Scan while*** b executes many Updates. In thii case, the updater will update VALUE_b each time, but will update VASIDE_{bc} only once, when it will detect that c is scanning concurrently. Intuitively, VASIDE_{bc} allows the scanner c to return a value for process b that was written by b during an update started no later than the end of the color producing step of the current scan. Therefore, such value can depend only on values that are not more recent than the values returned by the ***Scan.***

---

1. For all $c \neq b$, read $qcolor_b[c] := \text{PCOLOR}_{cb}$
2. For all $c \neq b$, if $qcolor_b[c] \neq \text{QCOLOR}_{bc}$
   then $vaside_b[c] := \text{VALUE}_b$
3. Atomically write:
   $\text{VALUE}_b :=$ new value
   For all $c \neq b$, $\text{VASIDE}_{bc} := vaside_b[c]$
   For all $c \neq b$, $\text{QCOLOR}_{bc} := qcolor_b$ [c]

Fig. 1. Update Operation for Process b.

We superscript the values of variables to indicate the execution of Update or Scan in which they are written. For example PCOLOR^l_{bc} is the value of PCOLOR_{bc} written during Scan S^l_b. Next, we construct an explicit partial linearization order $\Longrightarrow$ as follows. Define $U^j_q \Rightarrow S^i_p$ to hold if S^i_p takes the value originally written by U^j_q. (Note that S^i_p may read thii value from

1. **Call** Produce
2. For **all** c $\neq b$ atomically read:
   $value_b[c]$:= $VALUE_c$
   $qcolor_b[c]$:= $QCOLOR_{cb}$
   $vaside_b[c]$:= $VASIDE_{cb}$
3. For all $c \neq b$
   if $qcolor_b[c] \neq pcolor_b[c]$
      then $data_b[c]$:= $value_b[c]$
      else $data_b[c]$:= $vaside_b[c]$
4. Return($data_b[1]$,..., $VALUE_b$, . . . , $data_b[n]$)

Fig. 2. Scan Operation for Process b.

1. For **all** c $\neq b$ $pcolor_b[c]$:= $PCOLOR_{bc}$ + 1
2. Atomically write **for all** $_{c \neq b} PCOLOR_{bc}$:= $pcolor_b[c]$

Fig. 3. ***Produce Operation*** for Process b.

$VASIDE^k_{qp}$, where $k > j$). ***Define*** $\Longrightarrow$ to be the transitive closure of $\longrightarrow \cup \Rightarrow$. It follows from the following two lemmas that the ***Scan*** and ***Update*** procedures yield a weak snapshot memory.

Lemma 2. *The relation* $\Longrightarrow$ *is a partial order*

Proof: It suffices to check that $\Longrightarrow$ is acyclic. Suppose there exists a cycle $A_0, \ldots, A_k$, where adjacent operations are related by $\longrightarrow$ or $\Rightarrow$, and the cycle length is minimal. Because $\longrightarrow$ is acyclic and transitiie, some of these operations must be related only by $\Rightarrow$. Since the cycle is minimal, and $\longrightarrow$ is transitive, there are no adjacent $\longrightarrow$ edges and therefore each consecutive pair A_i and A_{i+1}, if $A_i \Rightarrow A_{i+1}$ then $A_i \not\longrightarrow A_{i+1}$. Moreover, since $\Rightarrow$ goes only from ***Update*** to ***Scan*** operations, there are no adjacent $\Rightarrow$ edges, and therefore the edges of the cycle must alternate between $\Rightarrow$ and $\longrightarrow$. It follows that k is odd. Without loss of generality, assume $A_0 \Rightarrow A_1$.

We argue by induction that, for $\ell \geq 0$, we have that A_0 starts before $A_{2\ell+2}$ starts. Throughout the proof **all** subscripts are taken **modulo** k + 1. For the base case ($\ell = 0$), observe that A_0 and A_1 are concurrent by construction (otherwise we would have had both $A_0 \Rightarrow A_1$ and $A_0 \longrightarrow A_1$), and hence A_0 starts before A_1 **finishes**. Since by construction $A_1 \longrightarrow A_2$ (alternating edges property), we have that A_1 **finishes** before A_2 starts, and the base case follows.

Assume the result for ℓ. We have $A_{2\ell+2} \Rightarrow A_{2\ell+3}$ (alternating edges), and hence $A_{2\ell+2} \not\longrightarrow A_{2\ell+3}$, i.e. $A_{2\ell+2}$ and $A_{2\ell+3}$ are concurrent. This implies that $A_{2\ell+2}$ starts before $A_{2\ell+3}$ finishes. By the inductive hypothesis, A_0 starts before $A_{2\ell+2}$ starts and hence A_0 starts before $A_{2\ell+3}$ finishes. To finish the argument, note that $A_{2\ell+3} \longrightarrow A_{2\ell+4}$ (alternating edges), which implies that A_0 starts before $A_{2\ell+4}$ starts, completing the induction.

Recall that the cycle has even length, and that this length is at least 2. Thus, A_0 starts before A_{k+1} starts, but since all subscripts are **modulo** k + 1 this says that A_0 starts before itself, which is a contradiction. ■

Lemma 3. ***For each process,*** our ***weak scan*** returns ***the*** value written ***by the*** latest update ***ordered*** before that ***scan by*** $\Longrightarrow$.

Proof: Recall that v_q^k denotes the value originally written by U_q^k. Let U_q^j be the last update by process q to be ordered before S_p^i by $\Longrightarrow$. The proof of the lemma relies on the following claim.

Claim 4. U_q^j terminates **before** S_p^i ***reads*** VALUE, . ***Moreover,*** U_q^j ***does not read*** PCOLOR_{pq}^i.

Proof: By definition of $\Longrightarrow$ there must be a sequence $A_0, \ldots, A_k$ where adjacent operations are related by $\longrightarrow$ or $\Rightarrow$ and where $A_0 = U_q^j$ and $A_k = S_p^i$. The proof proceeds by induction on the length k of a **minimal** such sequence.

For the base case, k = 1, observe that either $U_q^j \longrightarrow S_p^i$ or $U_q^j \Rightarrow S_p^i$. **In the first case** the claim trivially follows from the regularity of a read. In the second case, since S_p^i returns v_q^j we have that U_q^j must terminate before S_p^i reads VALUE, . To complete the proof of this case, we will show by contradiction that U_q^j does not read PCOLOR_{pq}^i. Suppose otherwise. It follows from Step 3 of the ***Scan*** operation that S_p^i could not have taken v_q^j from VALUE, because $qcolor_q^j[p] = pcolor_p^i[q]$. So S_p^i must have taken v_q^j from VASIDE_{qp}. Thii implies that there is some $U_q^{j'}$, j' > j that wrote v_q^j into VASIDE_{qp}, and that terminated before S_p^i performs Step 2. We show that there is no such $U_q^{j'}$. **Since** U_q^j reads PCOLOR_{pq}^i, the monotonicity of a read implies that so does any later $U_q^{j'}$ that terminates before S_p^i**'s** READ in Step 2, and hence it follows from the code of the Update operation that any such later $U_q^{j'}$ ***sees*** $qcolor_q[p] = \text{QCOLOR}_{qp}$, and hence does not write v_q^j into VASIDE_{qp}.

Assume the claim for k and suppose the minimal sequence from U_q^j to S_p^i is of length k + 1. Then one of the following must hold:

1. $U_q^j \longrightarrow U_z^l \Longrightarrow S_p^i$.
2. $U_q^j \Rightarrow S_g^m \longrightarrow U_z^l \Longrightarrow S_p^i$
3. $U_q^j \Rightarrow S_g^m \longrightarrow S_p^i$

(We don't consider the case: $U_q^j \longrightarrow S_g^m \Longrightarrow S_p^i$ because then either $U_q^j \longrightarrow S_g^m \longrightarrow U_z^l \Longrightarrow S_p^i$ which leads to Case **(1)**, or $U_q^j \longrightarrow S_g^m \longrightarrow S_p^i$ which leads to the base case of $U_q^j \longrightarrow S_p^i$.)

For Case **(1),** by the inductive hypothesis U_z^l does not read PCOLOR_{pq}^i and it terminates before S_p^i**'s** READ in Step 2. Clearly, the regularity of a read implies that U_z^l does not start after the ***Produce*** operation of S_p^i completed. Consequently, U_q^j completed before thii ***Produce*** is completed, and the claim follows. In Case (2) we have that U_q^j must have completed before S_g^m has completed and hence before U_z^l has started, and the claim follows as in Case (1). For case (3) we have that U_q^j completed before S_p^i started and the claim **trivially** follows. ■

Now observe that since U_q^j is the last update of q ordered before S_p^i by $\Longrightarrow$, S_p^i could not have returned $v_q^{j'}$ for some j' > j. Therefore, to complete the proof it is enough to show that

S_p^i does not return $v_q^{j'}$ for j' < j. However, from Claim 4 it follows that U_q^j has completed before S_p^i performs its read in Step 2. So before the time S_p^i performs Step 2, v_q^j is written into VALUE, . The only way that S_p^i could still take some $v_q^{j'}$ for j' < j is if it takes it from VASIDE_{qp}. This can happen only if the color that that S_p^i reads, $\text{QCOLOR}_{qp}^{j'''} = \text{PCOLOR}_{pq}^i$. By Claim 4, U_q^j does not read PCOLOR_{pq}^i and hence we have that $\text{QCOLOR}_{qp}^j \neq \text{PCOLOR}_{pq}^i$. This implies that there exists j < j''' ≤ j" such that $\text{QCOLOR}_{qp}^{j'''} \neq \text{QCOLOR}_{qp}^{j'''-1}$. But this implies that VASIDE_{qp} is updated with $v_q^{j'''-1}$ by update $U_q^{j'''}$, contradicting the assumption that S_p^i takes $v_q^{j'}$ from VASIDE_{qp}, for some j' < j. ■

4.2 Review of the *Traceable* Use Abstraction

We use a simplified version of the Traceable Use Abstraction of Dwork and Waarts [18] in order to convert the unbounded weak snapshot described in the previous section into a bounded one. We start by reviewing the abstraction. Recall that in the unbounded solution, when process b produces a new color for process c, this new color was never produced by b for c beforehand. This feature implies that when b sees VALUE_c tagged by this new color it knows that this VALUE_c is too recent (was written after the scan began), and will not return it as the result of its scan. However, the same property will follow also if when b produces a new color for c, it will simply choose a color that is guaranteed not to tag c's value unless b produces it for c again. To do this b must be able to detect which of the colors it produced for c may still tag c's values. This requirement can be easily satisfied by incorporating a simplified version of the Traceable *Use* abstraction.

In general, the goal of the Traceable Use abstraction *is* to enable the *colors* to *be traceable,* in that at any time it should be possible for a processor to determine which of its colors might tag any current or future values, where by "future value" we mean a value that has been prepared but not yet written. Although we allow a color that is marked as "in use" not to be used at all, we require that the number of such colors will be bounded.

The simplified version of the *Traceable* Use abstraction has three types of wait-free operations: ***Consume, Reveal and Garbage Collection.***

- ***Consume:*** Allows the calling processor to obtain the current color produced for it by another processor. It takes two parameters: the name c of the processor from which the color is being consumed, and the name of the color (that is, the shared variable holding the color). It returns the value of the consumed color.
- ***Reveal:*** Allows a processor to update a vector containing its colors. It takes two parameters: the name of the vector and a new value for the vector.
- Garbage ***Collection:*** Allows a processor to detect all of its colors that are currently "in use". It takes a list of shared variables in which the garbage collector's colors reside. It returns a list of colors that may currently be in use.

It is important to distinguish between shared variables of an algorithm that uses the *Traceable* Use abstraction and auxiliary shared variables needed for the implementation of the abstraction. We call the first type of variables principal shared variables, and the second type ***auxiliary.*** Only principal shared variables are passed to the Garbage ***Collection*** procedure. For example, in the weak snapshot system the only principal shared variables are PCOLOR, , and QCOLOR_{pq} foranypand q.

For $1 \leq i \leq n$, let $R_i^k(C_i^k)$ denote the *k*th *Reveal* (Consume) operation performed by processor i. X_i^k denotes the vector written by i during R_i^k. Let b consume a color X_c from c. Then X_c is said to be ***in use*** from the end of the ***Consume*** operation until the beginning of b's next ***Consume*** from c. In addition, all colors revealed by i appearing in any principal shared variables are also said to be in use. We require the following properties:

- Regularity: For any color X_p^a consumed by C_i^k, R_p^a begins before C_i^k terminates, and there is no R_p^b such that $R_p^a \longrightarrow R_p^b \longrightarrow C_i^k$.
- **Monotonicity:** Let C_i^k, $C_j^{k'}$ (where i and j may be equal) be a pair of ***Consume*** operations returning the colors X_p^a, X_p^b. If $C_i^k \longrightarrow C_j^{k'}$ then $a \leq b$.
- **Detectability:** If a color v revealed by processor b ***was*** not seen by b during ***Garbage Collection,*** then v will not be in use unless b ***reveals*** it again.
- **Boundedness:** The ratio between the maximum number of colors detected by b during ***Garbage Collection as*** possibly being in use and the maximum number of colors that can actually be in use concurrently is bounded by a constant factor.

The regularity and monotonicity properties of the *Traceable* Use guarantee the regularity and monotonicity properties of the bounded weak snapshot system. Detectability guarantees that a processor will be able to safely recycle its colors. Boundedness guarantees that by taking the local pools to be sufficiently large, the producer will always find colors to recycle.

The implementation of ***the Traceable Use*** abstraction described in [18] assumes following restrictions:

- **Conservation:** If a color v_c consumed by C_i^k from c is still used by i when it performs a new ***Consume*** from c, $C_i^{k'}$, $k' > k$, then at the start of $C_i^{k'}$ this color is in one of i's principal shared variables.
- **Limited Mobility:** A color consumed by b and stored in a principal shared variable X_b cannot be moved to a different principal shared variable Y_b (i.e., removed from X_b and placed in Y_b).

We show that the simplified ***Traceable Use*** under these restrictions suffices for our weak snapshot algorithm.

4.3 Bounded Weak Snapshots

For simplicity of exposition, we first present an algorithm that uses registers of size $O(nv)$, where v is the maximum number of bits in any process' value. In Subsection 4.4 we show how to modify thii algorithm so that registers of size $O(n+v)$ will suffice. In order to convert the unbounded solution to a bounded one, we replace the ***Produce*** operation shown in Figure 3 by the Produce operation shown in Figure 4. The meaning of the notation in Step 1.3 of the new Produce operation is that all n colors $pcolor_b[i]$, $1 \leq i \leq n$, are written atomically to PCOLOR_{bi}.

Also, Line 1 of the ***Update*** operation shown in Figure 1 is replaced by the following:

1. For all $c \neq b$, $qcolor_b[c] := Consume(c, \text{PCOLOR}_c b)$.

1.a. For all $1 \leq i \leq n$ $X[i]$: = *Garbage Collection*($PCOLOR_{bi}$, $QCOLOR_{ib}$)
1.b. For all c $\neq b$, choose $pcolor_b[c] \notin X[c]$
1.c. *Reveal* ($PCOLOR_b$, $pcolor_b$)

Fig. 4. *Produce* Operation for Process *b*.

Next we show that the bounded construction is indeed a weak snapshot algorithm. Observe that the proof of Lemma 2 applies directly for the bounded weak snapshot. The proof of Claim 4 holds because of the Regularity and Monontonicity properties of Traceable *Use.* In the proof of Lemma 3 all statements are true up to the statement "By Claim 4, U_q^j does not read $PCOLOR_{pq}^i$, we have that $QCOLOR_{qp}^j \neq PCOLOR_{pq}^i$". This is not necessarily correct because we recycle the colors. Clearly, if $QCOLOR_{qp}^j \neq PCOLOR_{pq}^i$, the same argument holds. Consider the case where $QCOLOR_{qp}^j = PCOLOR_{pq}^i$. By Claim 4, U_q^j did not read $PCOLOR_{pq}^i$. By the Detectability property of Traceable Use, this implies that in the end of the *Garbage Collection* step executed by S_p^i, QCOLOR, , contains a color $QCOLOR_{qp}^{j_1} \neq PCOLOR_{pq}^i$, where $j_1 > j$. The rest of the proof follows analogously, with j_1 replacing j.

The complexity of Traceable Use given *in* [18] *is O(n)* per each *Consume* or Reveal, and $O(n^2)$ per each Garbage *Collection.* However, in our particular case a trivial modification of the implementation in [18] reduces the cost of Garbage *Collection* to *O(k),* where *k* is the number of variables passed as parameters to the Garbage *Collection* procedure. Also, it is easy to see that we can get by with a constant number of colors for each pair of processes.

4.4 Reducing the Register Size

The weak snapshot described above uses registers of size $O(nv)$ where v is the maximum number of bits in any value $VALUE_b$. This is due to the fact that an updater *b* may set aside a different value for each scanner c in a variable $VASIDE_{bc}$, and all these values are kept in a single register. To reduce the size of the registers, each updater *b,* will store $VASIDE_{bc}$ in a separate register for each c. Only after this has been accomplished, *b* atomically updates $VALUE_b$ and, for all c $\neq b$, $VCOLOR_{bc}$.

The modifications to the code are straightforward. Lines 2 and 3 of the code for the Scan (Figure 2) are replaced by Lines 2' and 3' below.

2'. For all c $\neq b$ atomically read:
    $value_b[c] := VALUE_c$
    $qcolor_b[c] := QCOLOR_{cb}$
3'. For all c $\neq b$
    If $qcolor_b[c] \neq pcolor_b[c]$
        then $data_b[c] := value_b[c]$
        else read $data_b[c] := VASIDE_{cb}$

Lines 2 and 3 of the code for the *Update* operation (Figure 1) are replaced by the following Lines 2' and 3'.

2'. For aii c $\neq b$, if $qcolor_b[c] \neq \text{QCOLOR}_{bc}$ then $vaside_b[c] := \text{VALUE}_b$
$\text{VASIDE}_{bc} := vaside_b[c]$
3'. **Atomically** write:
$\text{VALUE}_b :=$ new value
For all c $\neq b$, $\text{QCOLOR}_{bc} := qcolor_b[c]$

Observe that the **only** difference between the modified and the original algorithm is that the shared variables VASIDE_{qp} and VALUE_q are not read atomically together by ***Scan*** and not written atomically together by the Update.

The **only** way we can get an execution of the modified algorithm that does not correspond to an execution of the original algorithm is when the ***Scan*** reads VALUE_q^k and $\text{VASIDE}_{qp}^{k'}$ and returns the latter, where $\text{VASIDE}_{qp}^k \neq \text{VASIDE}_{qp}^{k'}$. We now show that this can not happen.

Since VASIDE_{qp} is written before VALUE,, we have that $k' > k$. Since *the scan* S_p^i returns the **value** it read from VASIDE_{qp}, we have $\text{PCOLOR}_{pq}^i = \text{QCOLOR}_{qp}^k$. By the Detectability property, U_q^k consumes **color** PCOLOR_{pq}^i. By **Monotonicity,** for **all** $k \leq k_1 \leq k'$, $U_q^{k_1}$ ***consumes same color.*** Hence none of $U_q^{k_1}$ changed the **value** in VASIDE_{qp}, i.e. $\text{VASIDE}_{qp}^k = \text{VASIDE}_{qp}^{k'}$.

5 Applications

In this section, we explore two applications of the weak snapshot: bounded concurrent **times**tamping and randomized consensus. **First** we take the bounded concurrent timestamping **protocol** of Doiev and Shavit **[17],** and show that the **labels** can be stored in an abstract weak snapshot object, where each access to the **labels** is through either the weak snapshot ***Update*** or the weak snapshot ***Scan*** operation. The resulting protocol has running time, label size, and register size **all** O(n).

We then take the elegant randomized consensus protocol of Aspnes **[5],** and show that replacing atomic snapshot with weak snapshot leads to an algorithm with $O(n(p^2 + n))$ expected number of operations. This is an improvement of $\Omega(n)$ over the **original** algorithm.

5.1 Efficient Bounded Concurrent Timestamping

In a concurrent timestamping system, processes repeatedly choose labels, or timestamps, **reflecting** the **real-time** order of events. More precisely, there are two kinds of operations: Label generates a new timestamp for the **calling** process, and ***Scan*** returns an indexed set of labels $\vec{\ell} = \langle \ell_1, \ldots, \ell_n \rangle$ and an **irreflexive total** order $\prec$ on the labels.

For $1 \leq i \leq$ n, let L_i^k (S_i^k) denote the ***k*th *Label (Scan)*** operation performed by processor i (processor i need not keep track of k, thii is simply a notational device allowing us to describe long-lived runs of the timestamping system). **Analogously,** ℓ_i^k denotes the label **obtained** by i during L_i^k. Correctness is defined by the following axioms:

- **Ordering:** There exists an **irreflexive total** order $\Longrightarrow$ on the set of **all** Label operations, such that:
  - **Precedence:** For any pair of Label operations L_p^a and L_q^b (where p and q may be equal), if $L_p^a \longrightarrow L_q^b$, then $L_p^a \Longrightarrow L_q^b$.

- **Consistency:** For any ***Scan*** operation S_i^k returning $(\bar{\ell}, \prec)$, $\ell_p^a \prec \ell_q^b$ if and only if $L_p^a \Longrightarrow L_q^b$.

- **Regularity:** For any **label** ℓ_p^a **in** $\bar{\ell}$ returned by S_i^k, L_p^a ***begins*** before S_i^k terminates, and there is no L_p^b such that $L_p^a \longrightarrow L_p^b \longrightarrow S_i^k$.

- **Monotonicity:** ⊛𝕞♦ $S_i^k, S_j^{k'}$ (where i and j may be equal) be a pair of Scan operations returning the vectors $\bar{\ell}, \bar{\ell}'$ respectively which contain labels ℓ_p^a, ℓ_p^b respectively. If $S_i^k \longrightarrow S_j^{k'}$ then $a \leq b$.

Dolev and Shavit describe a ***bounded*** concurrent timestamping system that uses atomic multi-reader registers of size O(n) and whose ***Scan***[7] ***and Label*** operations take time $O(n^2 \log n)$ and O(n) respectively. They also mention that the labels can be stored in an abstract atomic snapshot object, where each access to the labels is through either atomic snapshot **Update** or ***Scan*** operation. More specifically, they would replace the ***Collect*** performed during the Label operation by an atomic snapshot ***Scan,*** would replace the simple writing of the new label with an atomic snapshot **Update,** and would replace their entire original ***Scan with an*** atomic snapshot Scan.

However, as they note, this transformation has drawbacks. The size of the atomic registers in all known implementations of atomic snapshot **memory** is 0(nv), where v is the size of the local value of each processor, and hence the size of the atomic registers in the resulting timestamping system is $O(n^2)$ (because here v is a label, and their label is of size n). Second, since both Update and ***Scan*** operations of the snapshot take $O(n^2)$ steps, then while the running time of the ***Scan*** **operation** in the resulting timestamping system improves, the **running** time of the ***Label*** operation increases to $O(n^2)$.

We show that one can replace the atomic snapshot abstract object in the Dolev-Shavit timestamping system by the weak snapshot object. Note that this leads to a solution without the above-mentioned drawbacks. More precisely, we get a timestamping system with linear running time, register size and label size.

Next we prove that the resulting system is indeed a bounded concurrent timestamping system.

Theorem 5. Our modification ***of the Dolev-Shavit*** algorithm yields ***a*** bounded concurrent timestamping system.

Proof: Regularity and **Monotonicity** follow directly from the analogous properties of the weak snapshot (more specifically, they follow from the Regularity and the Monotonicity of Scans properties of weak snapshot). To complete the proof we need to show the Ordering property.

Consider an execution of our algorithm and focus on the sequence of labelling operations. In our algorithm, when a process performs a Label operation it collects the labels of the other processes using a weak snapshot ***Scan, while in the*** original algorithm of Dolev and Shavit these labels are obtained using a simple Collect. However, for **every** execution of our algorithm, there exists an execution of the Dolev-Shavit algorithm that produces the same sequence of the **labelling** operations. This is due to the fact that the set of labels read by a weak snapshot scan can be also read by a collect executed in the same time interval, and because the result of a **labelling** operation in the Dolev-Shavit algorithm depends only on the set of labels collected during this operation. Thii implies that there exists an irreflexive total

[7] Note that this Scan is different from our weak snapshot Scan.

order on the labelling operations in the execution of our algorithm that is consistent with the precedence relation on the labelling operations. (The ordering is the one guaranteed by the proof of the Dolev-Shavit algorithm on the corresponding execution of their algorithm.)

Given an execution, the total order on the labelling operations defined by Dolev and Shavit is as follows: if one labelling operation reads the label produced by another labelling operation, then the first operation is ordered after the second. To get the total order, take the transitive closure of this partial order and extend it to a total order by considering the values of the labels.

The next step is to show that the order produced by a *Scan* operation of our algorithm is consistent with this total order. A *Scan* operation of our algorithm is just a weak snapshot *Scan.* Consider a weak snapshot *Scan* that returns a set of labels $\bar{\ell}$. To compute the order between these labels, our algorithm makes direct use of the appropriate procedure in the Dolev-Shavit algorithm. Therefore, it remains to show that the order on these labels produced by this procedure is consistent with the total order defined above.

Define a modified execution of our algorithm where we stop each process after it completes the labelling operation that generates its label in $\bar{\ell}$. Observe that the Monotonicii of Scans property of weak snapshots implies that none of the labelling operations in the original execution that generated labels in $\bar{\ell}$ can observe labels that were not written in the modified execution. Consider a *Scan* of the original Dolev-Shavit algorithm that is executed at the end of this modified execution. The Scan of Dolev-Shavit reads the same labels as in-l. The ordering of the labels computed by this *Scan* is consistent with the ordering on the labelling operations in the modified execution, and hence the ordering of the labels produced by our algorithm is also consistent with the total order on the labels defined by the modified execution.

We claim that the total order on *Label* operations obtained for the original execution (from which we obtained the modified one) is consistent with the total order obtained by the modified execution. In other words, we have the original (infinite) execution and a modified (truncated) execution. Consider labelling operations that appear only in both executions and the two total orders defined on these operations. Note that the only way these two total orders could be inconsistent is if there exists a labelling operation in the original execution that generated a label in $\bar{\ell}$ and that read a label (during its weak snapshot *Scan)* that was not written in the modified execution. However, since the labels in $\bar{\ell}$ are read by a weak snapshot *Scan, the* "Monotonicity of Scans" property of weak snapshots implies that thii is impossible. ■

5.2 Efficient Randomized Consensus

In a randomized consensus protocol, each of n asynchronous processes starts with a *preference* taken from a two-element set (typically $\{0,1\}$), and runs until it chooses a decision value and halts. The protocol is correct if it is *consistent:* no two processes choose different decision values; *valid:* the decision value is some process's preference; and randomized *wait-free:* each process decides after a finite expected number of steps. When computing a protocol's expected number of steps, we assume that scheduling decisions are made by an adversary with unlimited resources and complete knowledge of the processes' protocols, their internal states, and the state of the shared memory. The adversary cannot, however, predict future coin flips.

Our technical arguments require some familiarity with the randomized consensus protocol of Aspnes [5]. This protocol makes two uses of atomic snapshot, both of which can be replaced by our weak snapshot. The protocol is centered around a *robust weak shared* coin

protocol, which is a kind of collective coin flip: all participating processes agree on the outcome of the coin flip, and an adversary scheduler has only a slight influence on the outcome. The n processes collectively undertake a one-dimensional random walk centered at the origin with absorbing barriers at $\pm$ (K + n), for some $K > 0$. The shared coin is implemented by a shared counter. Each process alternates between reading the counter's position and updating it. Eventually the counter reaches one of the absorbing barriers, determining the decision value. While the counter is near the middle of the region, each process flips an unbiased local coin to determine the direction in which to move the counter. If a process observes that the counter is within n of one of the barriers, however, the process moves the counter deterministically toward that barrier. The code for the robust shared coin appears in Figure 5.

```
FUNCTION SharedCoin
repeat
1. c := read(counter)
2. if c ≤ (K + n) then decide 0
3. elseif c ≥ (K + n) then decide 1
4. elseif c ≤ −K then decrement(counter)
5. elseif c ≥ K then increment(counter)
6. else
7.       if LocalCoin=0 then decrement(counter)
8.       else increment(counter)
```

Fig. 5. Robust Weak Shared Coin Protocol (Aspnes[5])

Aspnes implements the shared counter as an n-element array of atomic single-writer multi-reader registers, one per process. To increment or decrement the counter, a process updates its own field. To read the counter, it atomically scans all the fields. Careful use of modular arithmetic ensures that all values remain bounded. The expected running time of thii protocol, expressed in primitive reads and writes, is $0(n^2(p^2 + n))$, where p is the number of processes that actually participate in the protocol.

The shared counter at the heart of this protocol is linearizable [20]: There exists a total order "$\Longrightarrow$" on operations such that:

- If $A \longrightarrow B$ then $A \Longrightarrow B$.
- Each Read operation returns the sum of all increments and decrements ordered before it by $\Longrightarrow$.

We replace the linearizable counter with a different data abstraction: by analogy with the definition of weak snapshot, a *weak* counter imposes the same two restrictions, but allows $\Longrightarrow$ to be a partial order instead of a total order. Informally, concurrent Read operations may disagree about concurrent increment and decrement operations, but no others. We can construct a weak counter implementation from Aspnes's linearizable counter implementation simply by replacing the atomic snapshot scan with a weak snapshot scan. We now argue that the consensus protocol remains correct if we replace the linearizable counter with a more efficient weak counter.

The proof of the modified consensus protocol depends on the following lemma which is analogous to a similar lemma in [5]. Recall from Section 3 that for each *Scan* operation returning a vector $\bar{v}$ of values, there is an associated time t_{scan}, the latest time between the start and end times of the *Scan* at which the spans of the values in $\bar{v}$ intersect.

Let $R_p^i (I_q^j, D_q^j)$ denote p's (q's) i^{th} (j^{th}) read (increment, decrement) operation.

Lemma 6. *If* R_p^i returns value $v \geq K + n$, then all reads *whose tscan is not* smaller than *the tscan of* R_p^i will return *values* $\geq K + 1$. (The symmetric claim *holds when* $v \leq -(K + n)$.)

Proof: Suppose not. Pick an earliest (with respect to t_{scan}) R_q^j that violates the hypothesis. Denote the t_{scan} of R_p^i by t_p. Since $t$scan of R_q^j is $t_q \geq t_p$, it follows from the definition of t_{scan} that R_q^j must observe all updates that were completed before t_p, i.e. all these updates are ordered before it by $\Longrightarrow$.

Observe that for each processor z, any update U_z^k that completes not before t_p, except the first such update, must follow a read R_z^k that *started* not before t_p and hence the t_{scan} of this read (t_z) is not smaller than t_p. Note that by definition of "observed" relation, any R_z^k observed by R_q^j completes before t_q and hence $t_p \leq t_z < t_q$. Since R_q^j is the first to violate the claim, we have that any such R_z^k returns a value $\geq K + 1$.

Any counter modiition that follows such a read (R_z^k) must be an increment (see Step 5). Since processes alternate between reads and updates, any update seen by R_p^i *and* not seen by R_q^j finishes after t_p, and hence any subsequent update of the same processor that is observed by R_q^j must be an increment. Any update observed by R_q^j but not by R_p^i must finish after t_p. Similar to the argument above, any subsequent update of the same processor that is observed by R_q^j must be an increment.

The claim follows since any update by processor p that is not observed by R_p^i but is observed by R_q^j must follow a read that started after t_p, and hence must be an increment. ■

The protocol also uses an atomic snapshot to make the protocol's running time depend on p, the number of active processes. For thii purpose, in addition to the shared counter used for the random walk, the protocol also keeps two additional counters, called active counters (implemented in the same way as the "random walk" counter), to keep track of the number of active processes that start with initial values 0 and 1. Each process increments one active counter before it modifies the random walk counter for the first time. (More specifically, if the processor starts with initial value 0, it increments the first active counter, and otherwise it increments the second.) All three counters (that is, the shared coin counter and the two active counters) are read in a single atomic snapshot scan.

The proof of the expected running time of the protocol hinges on the following lemma, which holds even if we replace the atomic snapshot scan by a weak snapshot scan. Define the true position of the random walk at any instant to be the value the random walk counter would assume if all operations in progress were run to completion without starting any new operations.

Lemma 7. *Let* τ be *the true position* of *the* random *walk at* t_{scan} of R_p. *If* R_p returns *values* c, a_0, and a_1 *for* the random *walk* counter and *the two active counters, then* $c - (a_0 + a_1 - 1) \leq \tau \leq c + (a_0 + a_1 - 1)$.

Proof: A process q affects the random walk's true position only if it has started to increment or decrement the random walk counter by time t_{scan}. Any q that has started to modify the random walk counter by the t_{scan} of R_p has already finished incrementing the appropriate active counter before that time, so R_p observes that increment. It follows that R_p fails to observe at most $(a_0 + a_1 - 1)$ increments or decrements active at its t_{scan}, and the result follows. ■

6 Conclusions

We have defined the weak snapshot scan primitive and constructed an efficient implementation of it. We have given two examples of algorithms designed using the strong primitive of atomic snapshot scan for whiih it was possible to simply replace the expensive atomic snapshot with the much less expensive weak snapshot scan. Indeed, it seems that in many cases atomic snapshot scan can be simply replaced by weak snapshot scan. Our construction relied on the *Traceable* Use abstraction of Dwork and Waarts [18]. Alternatively, we could have used the weaker primitives of Tromp [28] or of Kirousis, Spirakis, and Tsigas [23].

In a similar spirit to the weak snapshot, one can define a weak concurrent timestamping system, which, roughly speaking, satisfies the properties of the standard timestamping system except that the ordering $\Longrightarrow$ on *Label* operations and the $\prec$ orders on labels are partial rather than total. Such a timestamping system is interesting for two reasons: it is conceptually simple and it can replace standard timestamping in at least one situation: Abrahamson's randomized consensus algorithm [1].

In conclusion, we can generalize our approach as follows. Consider a concurrent object with the following sequential specification. [8]

- **Mutator** operations modify the object's state, but do not return any values. Mutator operations executed by different processes commute: applying them in either order leaves the object in the same state.
- **Observer** operations return some function of the object's state, but do not modify the object.

A concurrent implementation of such an object is linearizable if the precedence order on operations can be extended to a total order $\Longrightarrow$ such that the value returned by each observer is the result of applying all the mutator operations ordered before it by $\Longrightarrow$. This kind of object has a straightforward wait-free linearizable implementation using atomic snapshot scan ([6]). A weakly *linearizable* implementation is one that permits $\Longrightarrow$ to be a partial order instead of a total order. This paper's contribution is to observe (1) that weakly linearizable objects can be implemented more efficiently than any algorithm known for their fully linearizable counterparts, and (2) there are certain important applications where one can replace linearizable objects with weakly linearizable objects, preserving the application's modular structure while enhancing performance.

Acknowledgements

We would like to thank Jim Aspnes, Hagit Attiya, and Nir Shavit for helpful discussions.

[8] This definition is similar to Anderson's notion of a pseudo read-modify-write (PMRW) operation [4]. Anderson, however, requires that all mutators commute, not just those applied by different processes.

References

1. K. Abrahamson, On Achieving Consensus Using a Shared Memory, *Proc.* 7ACM Symposium on Principles of Distributed Computing, pp.291-302, 1988, .
2. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic Snapshots of Shared Memory, *Proc.* 9ACMSymposfum on *Principles* of Distributed Computing, pp. l-13,1990.
3. J. Anderson, Composite Registers, *Proc.* 9 ACM Symposium *on* Principles of Distributed Computing, pp. 15-30, August 1990.
4. J. Anderson, and B. Groselj, Beyond Atomic Registers: Bounded Wait-free Implementations of Non-trivial Objects, *Proc.* 5th *International* Workshop *on* Distributed Algorithms, Delphi, Greece, October 1991.
5. J. Aspnes, Tie- and Space-Efficient Randomized Consensus, to appear in the *Journal of Algorfthms.* An earlier version appears *in Proc. 9ACMSymposfum on Principles of Dfstrfbuted Computing,* pp. 325-331, 1990.
6. J. Aspnes and M.P. Herliiy Wait-Free Data Structures in the Asynchronous PRAM Model, *Proc. 2nd Annual Symposium on Parallel Algorithms and Architectures,* July 1990, pages 340-349, Crete, Greece.
7. J. Aspnes and M.P. Herlihy, Fast Randomized Consensus using Shared Memory, *Journal of Algorithms,* 11(3):441-461, 1990.
8. H. Attiya, D. Dolev, and N. Shavit, Bounded Polynomial Randomized Consensus, *Proc. 8ACM Symposium on Principles of Distributed Computing,* pp. 281-294, 1989.
9. H. Attiya, N. Lynch, and N. Shavit, Are Wait-Free *Algorithms* Fast?, *Proc. 9 IEEE* Symposium *on* Foundations of Computer Science, pp. 363-375, 1990. Expanded version: Technical Memo MIT/LCS/TM-423, Laboratory for Computer Science, MIT, February 1990.
10. G. Bracha and 0. Rachrnan, Approximated Counters and Randomized Consensus, Technical Report Technion 662, 1990.
11. G. Bracha and 0. Rachrnan, Randomized Consensus in Expected $O(n^2$ log n), *Proc. 5th International Workshop on Distributed* Algorithms, Greece, 1991.
12. J.E. Burns and G.L. Peterson, Constructing Multi-reader Atomic Values from Non-atomic Values, Proceedings of *the* Sfxth *ACM* Symposium *on* Principles *of Dfstrfbuted Computing,* pages 222–231, 1987.
13. K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, Acm Trans. on Computer Systems *3:1,1985*, pp. 63-75.
14. B. Chor, A. Israeli, and M. Li, On processor coordination using asynchronous hardware, *Proc. 6th ACM Symposium on Principles of Distributed Computing, pages 86-97,* 1987.
15. E. W. Dijkstra, Solution of a problem in concurrent programming control, *Communications of the* ACM *8:165*, 1965.
16. D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus, *Journal of the ACM 34:1*, pp. 77-97, Januay, 1987.
17. D. Dolev and N. Shavit, Bounded Concurrent Tie-Stamp Systems are Constructible!, *Proc. 21 ACM Symposium on Theory* of Computfng, pp. 454-465, 1989.
18. C. Dwork and 0. Waarts, Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Tiiestamp Systems are Comprehensible!, IBM Research Report RJ 8425, October 1991. *Also,* to appear *in Proc. 24 ACM Symposium on Theory of Computing,* 1992.
19. M.P. Herliiy. Wait-free Synchronization, *ACM Transactions on Programming Languages and Systems,* 13(1):124–149, January 1991.
20. M.P. Herliiy and J.M. Wig, Linearizabiity: A Correctness Condition for Concurrent Objects, *ACM* Transactfons *on Programming Languages and Systems,* 12(3):463–492, July 1990.
21. A. Israeli *and* M. *Li, Bounded Tie Stamps, Proc. 28 IEEE Symposium on Foundations of Computer* Science, 1987.

22. A. Israeli and M. Pinhasov, A Concurrent Tie-Stamp Scheme which is Linear in Tie and Space, *manuscript*, 1991.
23. L. M. Kirousis, P. Spirakis and P. Tsigas, Reading Many Variables in One Atomic Operation Solutions With Linear or Subliiear Complexity, *Proc.* 5th International Workshop *on Distrfbuted Algorithms,* 1991.
24. L. Lamport, Concurrent readiig and writing. Communications of the ACM, 20(11):806–811, November 1977.
25. L. Lamport, The Mutual Exclusion Problem, Part I: A Theory of Interprocess Communication, *J.* ACM *33*(2), pp. 313-326, 1986.
26. G. Peterson, Concurrent Readiig While Writing, ACM Transactions *on* Programming Lan*guages* and *Systems* 5(1), pp. 46-55, 1983.
27. M. Saks, N. Shavit, and H. Woll, Optimal Tie Randomized Consensus - Making Resilient Algorithms Fast in Practice, Symposium on Discrete Algorithms, pp. 351-362, 1990.
28. J. Tromp, How to Construct an Atomic Variable, *Proc. 3rd International Workshop on Distrfbuted Algorithms,* LNCS *392*, 1989.

This article was processed using the LaTeX macro package with LLNCS style