Time-Lapse Snapshots

Cynthia Dwork
IBM Almaden Research Center
dwork@almaden.ibm.com

Maurice Herlihy
DEC Cambridge Research Lab.
herlihy@crl.dec.com

Serge Plotkin*
Stanford University
plotkin@theory.stanford.edu

Orli Waarts†
Stanford University
orli@cs.stanford.edu

December 12, 1994

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.

*Research supported by NSF Research Initiation Award CCR-900-8226, 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.

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 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 $A[P]$[1]. 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* [24], *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 [4] (bounded registers and exponential running time), Aspnes and Herlihy [7] (unbounded registers and $O(n^2)$ running time), and by Afek, Attiya, Dolev, Gafni, Merritt, and Shavit [3] (bounded registers and $O(n^2)$ running time). Here running time is measured by the number of accesses to shared memory. Chandy and Lamport [15] 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* (we also call it *time-lapse snapshot*), 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 indicate that weak snapshot scan can sometimes 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 [27], 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, reflecting 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 **labeling** 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 time order of their corresponding **labeling** operations.[2]

Israeli and Li [25] were the first to investigate bounded *sequential* timestamp systems, and Dolev and Shavit [19] 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 **labeling** operation, and $O(n^2 \log n)$ time for a **scan**. In their algorithm each process is assigned a single multi-reader single-writer register of $O(n)$ bits.

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

[2] 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.

Extending the Dolev-Shavit solution in a non-trivial way, Israeli and Pinhasov [26] obtained a bounded concurrent timestamp system that is linear in time and label size, but uses registers of size $O(n^2)$. An alternative implementation of their algorithm uses *single*-reader-single-writer registers[3] of size $O(n)$, but requires $O(n^2)$ time to perform a **scan**. Independent of, and slightly later, Dwork and Waarts [20] obtained a very 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 scan, 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. We do not need to prove directly that our timestamping is correct; rather, we show that assuming the Dolev-Shavit algorithm is a bounded concurrent timestamping system, so is ours.

Independently of our work, Gawlick, Lynch, and Shavit [22] actually structured the Dolev-Shavit algorithm around the atomic snapshot abstraction. The resulting algorithm is indeed significantly more simple and modular than that of Dolev and Shavit. Also, as opposed to the proof we present for our timestamping, they showed directly, introducing new proof tecnhiques, that their algorithm is a bounded concurrent timestamping system. However, using snapshots slowed down the original Dolev-Shavit algorithm yielding an algorithm that requires $O(n^2)$ accesses to registers of size $O(n^2)$ for both **labeling** and **scan** operations, given that the fastest known snapshot algorithm is the one in [3].

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 [23]. Consensus has no deterministic solution in asynchronous shared-memory [18]. 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 [16] (against a "weak" adversary), by Abrahamson [2] (exponential running time), by Aspnes and Herlihy [8] (the first polynomial algorithm), by Saks, Shavit, and Woll [32] (optimized for the case where processes run in lock step), and by Bracha and Rachman [12] (running time $O(n^2 \log n)$). After the time of our work, Aspnes and Waarts [9] presented a randomized consensus protocol that uses unbounded registers and in which a process performs $O(n \log^2 n)$ expected operations.

Protocols that use bounded registers have been proposed by Attiya, Dolev, and Shavit [10] (running time $O(n^3)$), by Aspnes [6] (running time $O(n^2(p^2 + n))$, where p is the number of active processors), and by Bracha and Rachman [13] (running time $O(n(p^2 + n))$). The bottleneck in Aspnes' algorithm is atomic snapshot. Replacing this atomic snapshot with our more efficient weak snapshot improves the running time by $\Omega(n)$ (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 [13]. Both our consensus algorithm and the one in [13] 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 this paper is organized as follows. Section 2 gives our model of computation and

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

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 Definitions

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 complexity measures are expressed in terms of read and write operations on single-writer multi-reader registers of size $O(n)$. Polynomial-time algorithms for implementing large single-writer/multi-reader atomic registers from small, weaker, registers are well known [14, 28, 29, 31].

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 kth **update** (**scan**) of process i, and v_i^k the value written by i during U_i^k. The superscripts are omitted where it cannot 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 $\bar{v} = \langle v_1, \ldots, v_n \rangle$, 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 [24]. 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* memory as follows: we impose the same two conditions, but we allow "$\Longrightarrow$" to be a *partial order*[4] 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 **scan**s S and S' to disagree on the order of two **update**s U and U', but only if all four operations are concurrent with one another. Scanning processes must agree about **update**s that happened "in the past" but may disagree about concurrent **update**s. 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 **update**s occur concurrently.

Finally, for technical reasons we require that if an **update** is observed by a **scan** then this **update** terminates before the **scan** does. This property is not necessary when applying the weak snapshot scan abstraction to randomized consensus; it is useful, however, when applying it to the timestamping of [19]

[4] In this paper, all partial orders are irreflexive.

since it allows us to reduce the correctness of the resulting system to the correctness of the original algorithm of [19].

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 **scan**s: If S_a^i and S_b^j are two **scan**s 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.
- Monotonicity of **update**s: 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 **scan**s S_a^i, S_b^j return $\bar{v} = \langle v_1, \ldots v_n \rangle$ and $\bar{v}' = \langle v_1' \ldots v_n' \rangle$, 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 **update**s have overlapping spans. The following lemma formalizes the intuition that a weak snapshot **scan** returns a possibly existing state of the system.

Lemma 3.1 *If a weak snapshot* **scan** *S returns a set of values $\bar{v}$, 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 $U_q^j \longrightarrow U_p^i$. Since by assumption, the spans of v_p^i and v_q^j do not intersect, it follows from the definition of a span that $U_q^{j+1} \longrightarrow U_p^i$. Thus $U_q^{j+1} \Longrightarrow U_p^i$, and hence it follows from the transitivity of the partial linearization order that $U_q^{j+1} \Longrightarrow S$, which violates the requirement that each **scan** returns 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 3.1 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 implies 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 at or 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 **scan**s and **update**s, we need to ensure that a **scan** that did not return value v_a^j of process a because v_a^j is too new, will not return a value v_b^i that was written by process 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

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

Figure 1: **update** Operation for Process b.

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. This observation, due to Kirousis, Spirakis, and Tsigas [27], 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. This enables the scanner to distinguish older values from newer ones.

The next section 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 [20].

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 all 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 the **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}. (Colors produced *by* different processes or colors produced *for* different processes are considered different even if they have the same value.) 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 **update**s. In this 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** that started no later than the end of

1. Call **Produce**
2. For all $c \neq b$ atomically read:
   $value_b[c] := \text{VALUE}_c$
   $qcolor_b[c] := \text{QCOLOR}_{cb}$
   $vaside_b[c] := \text{VASIDE}_{cb}$
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 $(data_b[1], \ldots, \text{VALUE}_b, \ldots, data_b[n])$

Figure 2: **scan** Operation for Process b.

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

Figure 3: **Produce** Operation for Process b.

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**.

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 this value from 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 4.1 *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 transitive, some of these operations must be related only by $\Rightarrow$. Since $\Rightarrow$ goes only from **update** to **scan** operations, there are no adjacent $\Rightarrow$ edges. Moreover, since the cycle is minimal, and $\longrightarrow$ is transitive, there are no adjacent $\longrightarrow$ edges and therefore for each consecutive pair A_i and A_{i+1}, if $A_i \Rightarrow A_{i+1}$ then $A_i \not\longrightarrow A_{i+1}$. 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 ($\ell = 0$), since by assumption $A_0 \Rightarrow A_1$, the regularity of a read implies that 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 the regularity of a read 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 4.2 *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 4.3 *U_q^j terminates before S_p^i reads* VALUE_q. *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_q. 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_q because $\text{QCOLOR}_{qp}^j = \mathit{pcolor}_p^i[q]$. So S_p^i must have taken v_q^j from VASIDE_{qp}. This 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 $\mathit{qcolor}_q[p] = \text{QCOLOR}_{qp}$, 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 such a sequence is of one of the following forms:

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

For Case (1), without loss of generality we can assume that U_z^l is the last **update** by process z to be ordered before S_p^i by $\Longrightarrow$. Then 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 starts at or before the **Produce** operation of S_p^i completed. Consequently, U_q^j completed before this **Produce** is completed, and the claim follows.

For Case (2), observe that since $\Rightarrow$ goes only from **update** to **scan** operations, we have that any sequence from S_g^m to S_p^i is either of the form (2.i) $S_g^m \longrightarrow U_z^l \Longrightarrow S_p^i$, or (2.ii) $S_g^m \longrightarrow S_p^i$. By transitivity of $\longrightarrow$, if (2.i) holds then $U_q^j \longrightarrow U_z^l \Longrightarrow S_p^i$, and if (2.ii) holds, then $U_q^j \longrightarrow S_p^i$. This implies that a minimal sequence from U_q^j to S_p^i cannot be of the form of Case (2).

For Case (3), we have again that either (3.i) $S_g^m \longrightarrow U_z^l \Longrightarrow S_p^i$, or (3.ii) $S_g^m \longrightarrow S_p^i$. Since U_q^j must have completed before S_g^m has completed, it follows analogously to the above that a minimal sequence from U_q^j to S_p^i cannot be of the form of Case (3). ■

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 4.3 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_q. 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 S_p^i reads, say $\text{QCOLOR}_{qp}^{j''}$, is the same as PCOLOR_{pq}^i. By Claim 4.3, 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 $j < j''' \leq j''$ 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 $j' < j$. ■

To complete the proof that our **scan** and **update** procedures yield a weak snapshot memory, we need to show that each of the values returned by a weak snapshot **scan** was written by an **update** operation that terminated before the **scan** did. But this is clear by inspection.

4.2 Review of the Traceable Use Abstraction

We use a simplified version of the **Traceable Use** abstraction of Dwork and Waarts [20] in order to convert the unbounded weak snapshot described in the previous section into a bounded one. 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 earlier. 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 process 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.

Due to the asynchrony and concurrency, if revealing a color (i.e., writing a new color) were done by a simple write and obtaining a color (i.e., in order to tag a value) were done by a simple read, the colors would not have been traceable. Therefore, to achieve its goal, the **Traceable Use** requires the processes to communicate through three types of wait-free operations: **traceable-read**, **traceable-write** and **garbage collection**.

- **traceable-read** allows the calling process to obtain the current color produced for it by another process; in other words, to *consume* a color. It takes two parameters: the name c of the process from which the color is being consumed (and to which the color belongs), and the name of the color (that is, the shared variable holding the color).[5] It returns the value of the consumed color.
- **traceable-write** allows a process to update a variable containing its colors; in other words, to *reveal* its new colors. It takes two parameters: the name of the variable and a new value for the variable.
- **garbage collection** allows a process 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 (intuitively, the list of where to look for the collector's colors). It returns a list of colors that may currently be in use.

[5] For simplicity, the description here is slightly different from the corresponding one in [20].

The processes will communicate through the **Traceable Use** abstraction as follows. For process b to obtain a color produced for it by another process, b invokes the **traceable-read** operation. To write a color that other processes may need to obtain from it through the **traceable-read** operation, the process will perform the **traceable-write** operation. Thus, if a variable written by b contains a color of $c \neq b$, this color must have been consumed earlier by b; on the other hand, colors of b appearing in variables owned by b (meaning, variables residing in single-writer registers of which b is the writer) do not need to be consumed from anybody. If b writes a new color for itself in one of its variables without consuming it from anybody, we say that this color is *produced* at that time. Finally, to detect which of its colors are in use in the system, the processor will perform **garbage collection**.

Each process has $n-1$ *pools* of available colors, one for each other process. When a color v is produced by a process b for process c, it is removed from the pool. The color v is not available for re-use by b (cannot be returned to the pool) before b invokes **garbage collection** and determines that v is not in use by the end of **garbage collection**.

Before we specify the properties of the **Traceable Use**, some notation is in order. First, 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 obtained through the **traceable-read** operation, revealed through the **traceable-write** operation and passed to the **garbage collection** procedure. For example, in the weak snapshot system the only principal shared variables are PCOLOR_{pq} and QCOLOR_{pq} for any p and q.

We now specify the situation in which a color is considered as being currently in use, and therefore not available for reuse. Let E be a finite prefix of an execution of a system in which communication is performed using the **Traceable Use** abstraction. Let E end at time t. Then a color v belonging to b is *not in use* at time t if in every extension $\hat{E}$ of E, if b does not produce v after time t in $\hat{E}$, then v never appears as a color of b in any principal shared variable after time t of $\hat{E}$. (Recall that colors produced by b for different processes are considered different even if they have the same value; thus b can produce v only for a specific process, say c.)

Finally, for $1 \leq i \leq n$, let TW_i^k (TR_i^k) denote the kth **update** (**traceable-read**) operation performed by processor i. X_i^k denotes a color written by i during TW_i^k. Then **Traceable Use** is required to have the following properties.

- Regularity: For any color X_p^a consumed by TR_i^k, TW_p^a begins before TR_i^k terminates, and there is no TW_p^b such that $TW_p^a \longrightarrow TW_p^b \longrightarrow TR_i^k$.
- Monotonicity: Let $TR_i^k, TR_j^{k'}$ (where i and j may be equal) be a pair of **traceable-read** operations returning the colors X_p^a, X_p^b. If $TR_i^k \longrightarrow TR_j^{k'}$ then $a \leq b$.
- Detectability: If a color v of process b was not seen by b during **garbage collection**, then v will not be in use before b again produces it.
- Boundedness: By taking the local pools to be sufficiently large, it is always possible to find some color not returned as in use by the **garbage collection** procedure.

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 process 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.

1.a. For all $1 \leq c \leq n$ $X[c] :=$ **garbage collection**($\text{PCOLOR}_{bc}, \text{QCOLOR}_{cb}$)
1.b. For all $c \neq b$, choose from your local pool for c $pcolor_b[c] \notin X[c]$
2. **traceable-write** ($\text{PCOLOR}_b, pcolor_b$)

Figure 4: **Produce** Operation for Process b.

[20] presented an implementation of the **Traceable Use** under three restrictions. Since the version we present here is a simplification of their abstraction, two of their restrictions (called Limited Indirection and Limited Values) are already imposed by the simplification. The remaining restriction is:

- Conservation: Let TR_b^k consume a color v from c, and let $TR_b^{k'}$ be the first **traceable-read** operation of b from c to follow TR_b^k. Denote by t' the time in which $TR_b^{k'}$ starts. Then if v appears in a principal shared variable of b at some time $t \geq t'$, then it appears in this variable throughout the period $[t', t]$.

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 Section 4.4 we show how to modify this algorithm so that registers of size $O(n + v)$ will suffice.

It is easy to see that the **Traceable Use** abstraction allows us to convert our unbounded solution to a bounded one. In the unbounded solution each time a process performed a **scan**, it produced a new color for each updater. In the bounded solution, the scanner will draw its colors from bounded pools, one pool for each updater. (Again, colors produced by different scanners or for different updaters are considered different.) The algorithm is exactly like the unbounded one, except that the processes communicate through the **Traceable Use** operations. Roughly speaking, to get hold of a color produced for it by another process, the process uses the **traceable-read** operation; to write its new colors, it uses the **traceable-write** operation, and to find a color it can produce again, it performs **garbage collection**.

Using the **Traceable Use** abstraction, the process will be able to narrow down the set of its colors suspected by it as colors that may tag values of another process, say c, to contain no more than 6 values (discussed below). And we take the pools to be of size 7, thereby ensuring that whenever the process wants to perform a **scan** , it can find, for each updater, a color it can reuse.

More specifically, in the bounded solution, every process has $n - 1$ pools of available colors, one pool for each other process. Each pool contains 7 colors. When a scanner b wants to produce a new color for updater c, it performs *garbage collection* to determine which of its colors may still tag values of c and which can be reused without causing confusion. In this way we ensure that when a process b produces v as its color for c for the kth time, there is not, and never will be, anything in the system that contains v from when it was produced by b for c for the $(k-1)$st time.

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 2 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 $c \neq b$, $qcolor_b[c] :=$ **traceable-read**(c, PCOLOR_{cb}).

Next we show that the bounded construction is indeed a weak snapshot algorithm. Observe that the proof of Lemma 4.1 applies directly for the bounded weak snapshot due to the regularity of **Traceable Use**. The proof of Claim 4.3 holds because of the regularity and monontonicity properties of **Traceable Use**. In the proof of Lemma 4.2 all statements are true up to the statement "By Claim 4.3, U_q^j does not read PCOLOR_{pq}^i, and hence we have that $\text{QCOLOR}_{qp}^j \neq \text{PCOLOR}_{pq}^i$". This is not necessarily correct any more because we recycle the colors. Clearly, if $\text{QCOLOR}_{qp}^j \neq \text{PCOLOR}_{pq}^i$, the rest of the proof holds. Consider the case where $\text{QCOLOR}_{qp}^j = \text{PCOLOR}_{pq}^i$. By Claim 4.3, U_q^j did not read PCOLOR_{pq}^i and it terminates before S_p^i does. By the Detectability property of **Traceable Use**, this implies that in the end of the **garbage collection** step executed by S_p^i, QCOLOR_{qp} contains a color $\text{QCOLOR}_{qp}^{j_1} \neq \text{PCOLOR}_{pq}^i$, where $j_1 > j$. The rest of the proof follows analogously, with j_1 replacing j. ■

Letting n be the number of processes in the system, the implementation of **Traceable Use** given in [20] requires $O(n)$ steps per a **traceable-write** (that is, $O(n)$ reads and writes of shared variables) and $O(1)$ step to consume one color using the **traceable-read** operation. In our particular case a trivial modification of the implementation in [20] reduces the cost of **garbage collection** to $O(k)$, where k is the number of variables passed as parameters to the **garbage collection** procedure; moreover when b invokes **garbage collection** to detect how many of its colors for c are in use, **garbage collection** detects at most 6 colors. (The above follows because, using the notations of [20], when a process b wants to detect which of its colors for c may currently be in use, it only needs to read $Ai\text{-}PCOLOR_{bc}[b]$, for $i = 1, 2, 3$ (colors set aside by b for c when b detects that c is trying to consume b's color for c), $Bi\text{-}PCOLOR_{cb}[b]$ for $i = 1, 2$ (colors written by c for b when c is trying to consume b's color for it), and QCOLOR_{cb}; moreover, each **garbage collection** needs to read these variables only once because in our application colors can only be in what [20] call 'direct use' (see Section 8.3 of [20]).)

Clearly, each **Produce** operation requires $O(n)$ invocations of **garbage collection**, each of which costs $O(1)$, and one invocation of **traceable-write**, and thus takes $O(n)$ steps. Each **scan** requires one invocation of **Produce** and $O(n)$ simple reads, and thus takes $O(n)$ steps. Each **update** requires one simple write and $O(n)$ invocations of **traceable-read**, and hence takes $O(n)$ steps.

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 variable 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 $c \neq b$, QCOLOR_{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:
$value_b[c] := \text{VALUE}_c$
$qcolor_b[c] := \text{QCOLOR}_{cb}$

3'. For all $c \neq b$
If $qcolor_b[c] \neq pcolor_b[c]$
then $data_b[c] := value_b[c]$
else read $data_b[c] := \text{VASIDE}_{cb}$

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

2'. For all $c \neq b$
    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 time complexity of the modified algorithm is the same as the original one.

The next step is to show that the modified algorithm is a weak snapshot algorithm.

Clearly, 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 the **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 a **scan** S_p^i reads VALUE_q^k and $\text{VASIDE}_{qp}^{k'}$ and returns the later, where the value of VASIDE_{qp}^{k} is different from the value of $\text{VASIDE}_{qp}^{k'}$. We now show that this cannot happen. Suppose otherwise. Since VASIDE_{qp} is written before VALUE_q by an **update** and after it by a **scan**, 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 of **Traceable Use**, U_q^k consumes color PCOLOR_{pq}^i. By Monotonicity of **Traceable Use**, for all $k < k_1 \leq k'$, $U_q^{k_1}$ consumes PCOLOR_{pq}^i and hence saw $qcolor_q[p] = \text{QCOLOR}_{qp}$ when performing step 2 of the **update** operation. Hence none of these $U_q^{k_1}$s changed the value in VASIDE_{qp}, i.e. the value of VASIDE_{qp}^{k} is the same as the value of $\text{VASIDE}_{qp}^{k'}$. Contradiction. ■

5 Applications

In this section, we explore two applications of the weak snapshot: bounded concurrent timestamping and randomized consensus. First we take the bounded concurrent timestamping protocol of Dolev and Shavit [19], 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**. The resulting protocol has running time, label size, and register size all $O(n)$.

We then take the elegant randomized consensus protocol of Aspnes [6], 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

Our definition of a concurrent timestamping system is a slightly stronger version (due to Gawlick [21]) of the one given by Dolev and Shavit in [19]. In a *concurrent timestamping system*, processes repeatedly choose *labels*, or timestamps, reflecting the real-time order of events. More precisely, each process i has a label, denoted by ℓ_i. There are two kinds of operations: **labeling** generates a new timestamp for the calling process, and **scan** returns an indexed set of labels $\bar{\ell} = \langle \ell_1, \ldots, \ell_n \rangle$ and an irreflexive total order $\prec$ on the labels.

For $1 \leq i \leq n$, let L_i^k (S_i^k) denote the kth **labeling** (**scan**) operation performed by process i (process i need not keep track of k, this 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. In order to handle initial conditions, we assume that each processor has an initial label denoted by ℓ_i^0. To avoid distinguishing between these initial labels and the labels assigned by **labeling** operations we say that label l_i^0 was assigned to process i by a fictitous initial **labeling** operation L_i^0 that took place just before the beginning of the execution. Moreover, we define the L_i^0s for all i as concurrent. Correctness is defined by the following axioms:

- Ordering: There exists an irreflexive total order $\Longrightarrow$ on the set of all **labeling** operations,[6] such that:
  - Precedence: For any pair of **labeling** 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: Let $S_i^k, S_j^{k'}$ (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 $a \leq b$.
- Extended Regularity: For any label l_p^a returned by S_i^k, if $S_i^k \longrightarrow L_q^b$, then $L_p^a \Longrightarrow L_q^b$.

Dolev and Shavit describe a *bounded* concurrent timestamping system that uses atomic multi-reader registers of size $O(n)$ and whose **scan**[7] and **labeling** operations take time $O(n^2 \log n)$ and $O(n)$ respectively. Their **labeling** operation was simply performing a **collect** of all labels[8] and then writing a new label based on the ones collected; their **scan** was more involved. They mentioned 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 **labeling** 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 $O(nv)$, where v is the size of the 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 labels are of size n). Second, since both **update** and **scan** operations of the snapshot take $O(n^2)$ steps, so do both **labeling** and **scan** operations of the resulting timestamping system. Hence, while the number of steps performed by a timestamping **scan** in the resulting timestamping system improves, the running time of the **labeling** operation increases.

We show that one can replace the atomic snapshot abstract object in the Dolev-Shavit timestamping system by the weak snapshot object. (More specifically, we replace the **collect** performed during the **labeling** operation of [19] by a weak snapshot **scan**, replace their simple writing of the new label with a weak snapshot **update**, and replace their entire original **scan** with a weak snapshot **scan**.) 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.

[6] Observe that this order does not have to be consistent with the partial linearization order on the weak snashot **update** and **scan** operations.

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

[8] We say that a process **collect**s a variable X if it reads X_c for every process c.

Theorem 5.1 *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). Next we show the ordering property.

First we show that for each execution of our algorithm, there exists a corresponding execution of the Dolev-Shavit algorithm that produces the same sequence of **labeling** operations (refered to as the *corresponding* execution; if there is more than one, choose one arbitrarily). In our algorithm, when a process performs a **labeling** operation it reads the labels of the other processes using a weak snapshot **scan**, and then writes the new label using a weak snapshot **update**, while in the original algorithm of Dolev and Shavit these labels are obtained using a simple **collect**, and written using a simple write. However, since when a process performs a **collect**, it reads all labels once each in an arbitrary order, and since by definition, each label returned by a weak snapshot **scan** was written by a **labeling** operation, say L_p^k, that terminated before the **scan** did and such that no later L_p terminated before the **scan** started, we have that the set of labels read by a weak snapshot **scan** can also be read by a **collect** executed in the same time interval. The claim follows because the label written by a **labeling** operation in the Dolev-Shavit algorithm depends only on the set of labels collected during this operation.

The above claim immediately implies that there exists an irreflexive total order on the **labeling** operations in the execution of our algorithm that is consistent with the precedence relation on the **labeling** operations; this is simply the total order guaranteed by the Dolev-Shavit algorithm for the corresponding execution. More specifically, given an execution, the total order on the **labeling** operations will be as follows: if one **labeling** operation reads (through a weak snapshot **scan**) the label produced by another **labeling** 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 taken by the **labeling** operations.

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 timestamping algorithm is just a weak snapshot **scan**. Consider a weak snapshot **scan**, say S, performed in an execution E of our algorithm, that returns a set of labels $\bar{\ell}$. To compute the order on 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 on the **labeling** operations defined above.

Define a modified execution $E1$ of our algorithm where we stop each process in E after it completes the **labeling** operation that generates its label in $\bar{\ell}$. Observe that the transitivity of the partial order on the operations of the weak snapshot memory implies that none of the **labeling** operations that appears in both $E1$ and E can observe in E (using a weak snapshot **scan**) **labeling** operations that do not appear in $E1$. Thus, the set of current labels in the end of $E1$ is still $\bar{\ell}$. Consider an execution $\hat{E}1$ of the Dolev-Shavit algorithm that corresponds to our modified execution, and extend this corresponding execution by executing at the end a **scan**, say $\hat{S}$, of the Dolev-Shavit algorithm. Clearly, $\hat{S}$ returns the same labels as in $\bar{\ell}$. The order of the labels computed by the Dolev-Shavit **scan** $\hat{S}$ is consistent with the total order on the **labeling** operations in $\hat{E}1$. Since the order determined on $\bar{\ell}$ by S is exactly the same as their order determined by $\hat{S}$, and since the total order on the **labeling** operations in $E1$ was defined as identical to their order in $\hat{E}1$, we have that the order on $\bar{\ell}$ determined by S is consistent with the total order on the **labeling** operations in $E1$.

To complete the proof of the ordering property we show that the total order on the **labeling** operations in $E1$ is consistent with their order in E. (Thus, we consider only **labeling** operations that appear in both executions $E, E1$ and the two total orders defined on these operations.) However, the only way these two

total orders could be inconsistent is if there exists a **labeling** operation that appears in both $E1$ and E and which read in E (using a weak snapshot **scan**) a label assigned by a **labeling** operation that does not appear in $E1$. As mentioned above, this is impossible.

Finally we show that our system satisfies the extended regularity property. Recall that a property of our weak snapshot algorithm is that a value returned by a **scan** was written by an **update** operation that terminated before the **scan** does. Thus, since in our timestamping system, the last operation performed by a **labeling** operation is writing a new label, and since this is done by a weak snapshot **update**, each label returned by a timestamping **scan** of our system is assigned by a **labeling** operation, say L_1, that terminated before the **scan** does, and hence before the beginning of any **labeling** operation, say L_2, preceded by the **scan**. The precedence property of timestamping implies immediately that $L_1 \Longrightarrow L_2$. ■

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 [6]. 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.

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 this protocol, expressed in primitive reads and writes, is $O(n^2(p^2+n))$, where p is the number of processes that actually participate in the protocol.

The shared counter at the heart of this protocol is *linearizable* [24]: 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

FUNCTION SharedCoin

repeat
1. c := read(counter)
2. if $c \leq (K+n)$ then decide 0
3. elseif $c \geq (K+n)$ then decide 1
4. elseif $c \leq -K$ then decrement(counter)
5. elseif $c \geq K$ then increment(counter)
6. else
7. if LocalCoin=0 then decrement(counter)
8. else increment(counter)

Figure 5: Robust Weak Shared Coin Protocol (Aspnes[5])

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 [6]. Recall the definition of 't_{scan}' from Section 3. Our discussion there implies that a **scan** observes all **update**s that terminate before the t_{scan} of the **scan**, and does not observe any **update** that starts after the t_{scan} of the **scan**.

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

Lemma 5.2 *If R_p^i returns value $v \geq K+n$, then all reads whose t_{scan} is not smaller than the t_{scan} 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}s of R_p^i, R_q^j by t_p, t_q respecively. Clearly, any difference in the values returned by R_q^j and R_p^i must be caused by **update**s observed by exactly one of them (not necessarily the same one for each such **update**). It follows from the definition of t_{scan} that each **update** that completed before time t_p was observed by R_p^i, *i.e.* all these **update**s are ordered before it by $\Longrightarrow$. Moreover, since $t_q \geq t_p$, it follows that each such **update** was also observed by R_q^j. Thus, only **update**s that completed after t_p may be seen by exactly one of R_p^i, R_q^j.

Consider the set S containing all **update**s that completed at or after t_p excluding, for each $z \neq p$, the first such **update**. Since each of the **update**s in S started after t_p, none of them was observed by R_p^i. Let $S1$ denote a subset of S that contains all **update**s in S that were observed by R_q^j. To complete the proof it is enough to show that each **update** in $S1$ was an increment. This immmediately gives a contradiction to our assumption that R_q^j returns a value $< K+1$

Since processes alternate between reading and modifying the counter, any **update** U_z^k in $S1$ must follow a read R_z^k that started at or after t_p and hence the t_{scan} of this read (t_z) is not smaller than t_p. In addition, any such R_z^k completes before t_q (because, since U_z^k is observed by R_q^j, it must have started at or 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 modification that follows such a read (R_z^k) must be an increment (see Step 5), and we are done. ■

The protocol also uses an atomic snapshot to make the protocol's running time depend on p, the number of active processes. For this 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 process 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 5.3 *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. ■

Acknowledgements

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

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 which 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 [20]. Alternatively, we could have used the weaker primitives of Vitanyi and Awerbuch [35], Tromp [34], Kirousis, Spirakis, and Tsigas [27], or of Singh [33].

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 **labeling** 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 [2].

In conclusion, we can generalize our approach as follows. Consider a concurrent object with the following sequential specification.[9]

[9]This definition is similar to Anderson's notion of a *pseudo read-modify-write* (PMRW) operation [5]. Anderson, however, requires that all mutators commute, not just those applied by different processes.

- 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 ([7]). 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 that (1) 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.

References

[1] K. Abrahamson. On Achieving Consensus Using a Shared Memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pp. 291-302, August 1988.

[2] K. Abrahamson. On Achieving Consensus Using a Shared Memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pp. 291-302, August 1988.

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

[4] J. Anderson. Composite Registers. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pp. 15-30, August 1990.

[5] J. Anderson and B. Groselj. Beyond Atomic Registers: Bounded Wait-free Implementations of Non-trivial Objects. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, October 1991, Delphi, Greece.

[6] J. Aspnes. Time- and Space-Efficient Randomized Consensus. To appear in the *Journal of Algorithms*. An earlier version appears in *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pp. 325-331, 1990.

[7] J. Aspnes and M. P. Herlihy. Wait-Free Data Structures in the Asynchronous PRAM Model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, July 1990, pages 340–349, Crete, Greece.

[8] J. Aspnes and M. P. Herlihy. Fast Randomized Consensus Using Shared Memory. In *Journal of Algorithms*, 11(3):441-461, 1990.

[9] J. Aspnes and O. Waarts. Randomized Consensus in Expected $O(n \log^2 n)$ Operations per Processor. To appear in *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, October 1992.

[10] H. Attiya, D. Dolev, and N. Shavit. Bounded Polynomial Randomized Consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pp. 281-294, 1989.

[11] H. Attiya, N. A. Lynch, and N. Shavit. Are Wait-Free Algorithms Fast? In *Proceedings of the 9th 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.

[12] G. Bracha and O. Rachman. Randomized Consensus in Expected $O(n^2 \log n)$ Operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, 1991, Greece.

[13] G. Bracha and O. Rachman. Approximated Counters and Randomized Consensus. Technical Report 662, Computer Science Department, The Technion, December 1990.

[14] J.E. Burns and G.L. Peterson. Constructing Multi-reader Atomic Values from Non-atomic Values. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[15] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *Acm Trans. on Computer Systems 3:1,1985*, pp. 63–75.

[16] B. Chor, A. Israeli, and M. Li. On Processor Coordination Using Asynchronous Hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[17] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. In *Communications of the ACM 8:165*, 1965.

[18] D. Dolev, C. Dwork, and L Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. In *Journal of the ACM 34:1*, pp. 77–97, January, 1987.

[19] D. Dolev and N. Shavit. Bounded Concurrent Time-Stamp Systems are Constructible! In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pp. 454-465, 1989. An extended version appears in IBM Research Report RJ 6785, March 1990.

[20] C. Dwork and O. Waarts. Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible!, In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pp. 655-666, 1992. Preliminary version appears in IBM Research Report RJ 8425, October 1991.

[21] R. Gawlick. Concurrent Timestamping Made Simple. M.Sc. Thesis, MIT, May 1992.

[22] R. Gawlick, N. Lynch, and N. Shavit. Concurrent Timestamping Made Simple. In *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.

[23] M. P. Herlihy. Wait-free Synchronization. In *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[24] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[25] A. Israeli and M. Li. Bounded Time Stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987.

[26] A. Israeli and M. Pinhasov. A Concurrent Time-Stamp Scheme which is Linear in Time and Space. Manuscript, 1991.

[27] L. M. Kirousis, P. Spirakis and P. Tsigas. Reading Many Variables in One Atomic Operation Solutions With Linear or Sublinear Complexity. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, 1991.

[28] L. Lamport. Concurrent Reading and Writing. In *Communications of the ACM*, 20(11):806–811, November 1977.

[29] L. Lamport. The Mutual Exclusion Problem, Part I: A Theory of Interprocess Communication. In *J. ACM 33*(2), pp. 313-326, 1986.

[30] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pp. 590–599, 1991.

[31] G. Peterson, Concurrent Reading While Writing. In *ACM Transactions on Programming Languages and Systems 5*(1). pp. 46-55, 1983.

[32] M. Saks, N. Shavit, and H. Woll. Optimal Time Randomized Consensus - Making Resilient Algorithms Fast in Practice. In *Proceedings of the 2nd Symposium on Discrete Algorithms*, pp. 351-362, 1991.

[33] A. Singh. IPL 1992.

[34] J. Tromp. How to Construct an Atomic Variable. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, *LNCS 392*, 1989.

[35] P. M. B. Vitanyi and B. Awerbuch. Atomic Shared Register Access by Asynchronous Hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986.