Bounded Round Numbers

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

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

Orli Waarts*
IBM Almaden Research Center
waarts@almaden.ibm.com

Abstract

This paper presents a systematic, modular technique for transforming a large class of unbounded shared-memory algorithms into bounded algorithms. We show that any unbounded algorithm based on a certain *asynchronous rounds* structure can be "compiled" into a bounded algorithm in a way that preserves correctness *and* running time. As evidence that the asynchronous rounds structure is natural for wait-free algorithms, we identify a number of unbounded algorithms in the literature to which our transformation can either be applied directly, or applied after minor modifications. The running times of the resulting algorithms match these of their unbounded counterparts and hence in most cases the resulting algorithms are faster than any other known bounded solutions to the corresponding problems. In particular, we get a bounded consensus algorithm whose running time is $O(n \log^2 n)$ and a bounded snasphot scan algorithm whose running time is $O(n \log n)$.

1 Introduction

Many wait-free shared-memory algorithms have potentially unbounded running time, either because they use randomization to converge, or because they implement long-lived objects that must service an unbounded stream of requests. It is desirable to design such algorithms to consume *bounded space*: employing a bounded number of variables, each holding a bounded range of values. The question is both theoretical and practical. A full understanding of a problem should include establishing whether a bounded solution exists, and if so, whether such a solution is inherently more expensive than an unbounded solution. In practice, any unbounded solution will eventually require more space than is available, and due to increasing processor speeds and network sizes, 'eventually' may occur sooner than the algorithm designer expected.

Often, the easiest way to design a bounded algorithm is first to derive an unbounded algorithm, and then to "hand-craft" it into a bounded algorithm (*viz.*, *e.g.* [1, 8, 9, 10, 14, 15] among many others). The second step is frequently much harder and more error-prone than the first. This difficulty can usually be attributed to an absence of modularity: one cannot reason about functional correctness without reasoning about boundedness, and vice-versa.

This paper presents a systematic, modular technique for transforming a large class of unbounded shared-memory algorithms into bounded algorithms. We show that any unbounded algorithm based on a certain *asynchronous rounds* structure can be "compiled" into a bounded algorithm in a way that preserves correctness *and* running time. As evidence that the asynchronous rounds structure is natural for wait-free algorithms, we identify a number of unbounded algorithms in the literature to which our transformation can either be applied directly, or applied after minor modifications: randomized consensus [2], atomic snapshot scan [3, 7], randomized atomic snapshot scan [6], generalized approximate agreement [12], and long-lived data objects [13]. The running times of the resulting algorithms match those of their unbounded counterparts and hence in most cases the resulting algorithms are faster than any other known bounded solution to the corresponding problems. Here running time is measured by the number of ac-

*During part of this research the third author was in Stanford University and supported by IBM Graduate fellowship, U.S. Army Research Office Grant DAAL-03-91-G-0102, NSF Grant CCR-8814921, and ONR Contract N00014-88-K-0166.

12th ACM Symposium on Principles on Distributed Computing, Ithaca NY

cesses to shared memory.

We consider the class of wait-free algorithms that communicate via single-writer multi-reader atomic registers and proceed in a certain *asynchronous rounds* communication structure. Starting from round 1, at each round the process performs a computation, advances its round number, and proceeds to the next round. A process's actions, and the round to which it advances, do not depend on its exact round number but only on the distance of its current round number from those of other processes. Moreover, the process's actions are not affected by any process whose round number lags behind its own by more than a fixed limit. We also assume (for now) that these round numbers are the algorithm's only source of unboundedness.

Our basic approach is as follows. We view round-based algorithms as employing a *round number abstraction*, which is an abstract data object that maps each process to its current round number. This abstraction provides four basic operations: (1) a process can read another process's current round number, (2) it can determine, up to a fixed maximum, how far apart two round numbers are, (3) it can generate a new round number for itself, and (4) it can write its new round number to a shared variable. We show that any algorithm that uses round numbers in this disciplined way can be transformed, in a correctness-preserving way, into a bounded algorithm.

Our transformation proceeds in two steps. We first show how to implement the round number abstraction using an *intermediate* algorithm in which the round numbers are represented as unbounded *round vectors*. To show that our intermediate implementation is correct, we show that every execution using the intermediate algorithm can be mapped (via a refinement map) to an execution using the round number abstraction. In particular, the *compare* operation returns the same values in both executions. The intermediate algorithm is non-trivial, and is the principal technical contribution of this paper. Translating the intermediate (unbounded) algorithm into a bounded algorithm is done in a straightforward way by recycling values using the *traceable use* abstraction of Dwork and Waarts [10].

It is customary to measure the complexity of shared memory algorithms by the number of memory accesses (reads and writes). Our transformation requires $O(n)$ memory accesses both to take a new round number and to determine the distances among n current round numbers, the same number of accesses required by the original unbounded round-based algorithms[1] and therefore our transformation does not degrade the running time of the original algorithms. Our transformation does require larger registers, however. In particular, to find, in linear time, the distance between *every* pair of current rounds up to a fixed k, we need registers of size $O(n^3 \log n)$. Fortunately, all unbounded round-based algorithms with which we are familiar do not need to find the distance between every pair of current rounds, but only between pairs in which neither element is "too far" behind the highest current round number. For such cases we modify our transformation so that it maintains linear time complexity while requiring registers of size only $O(n \log n)$.

The principal contribution of this work is to provide a powerful abstraction to future algorithm designers, freeing them to reason about functional correctness independently of boundedness. Asynchronous shared-memory algorithms have a well-deserved reputation for conceptual complexity, and we believe that abstractions such as bounded round numbers can alleviate this problem. A secondary contribution is that our transformation yields bounded algorithms that are often conceptually simpler, more modular, and more efficient than their existing "hand-crafted" bounded counterparts.

Because of space limitations, some details are omitted from the proofs.

1.1 Related Work

The insight that boundedness is best achieved by providing appropriate abstractions is due to Israeli and Li [14], who introduced the notion of *bounded timestamping*. A bounded timestamp system allows processes to take *labels*, or timestamps, consistent with the real-time order of events. Israeli and Li developed a theory of bounded *sequential* timestamping systems, that is, timestamping systems for a world in which no two operations take place concurrently. Dolev and Shavit [8] were the first to define and construct bounded *concurrent* timestamping systems. Using the bounded concurrent timestamp abstraction, the formidable problem of constructing bounded multi-reader multi-writer atomic registers [15] becomes a straightforward exercise [11].

Bounded concurrent timestamps are not directly applicable to the class of algorithms studied here. Bounded timestamping systems are designed to reflect the real-time order of events, and hence a new label taken by a process has to be bigger than all current labels taken beforehand, regardless of the process' own current label. Round-based algorithms, on the other

[1] The choice of new round number in the original algorithm depends on the distance to the maximum round number of any process in the system; determining this requires reading all round numbers.

hand, are heavily based on the assumption that slow processes must observe fast processes, and hence they require that a new round number taken by a process whose current round number lags behind *should not be bigger* than the highest current round numbers.

The round number abstraction is strictly more powerful than timestamping. Round numbers provide a notion of *distance*: up to a fixed limit, one can determine how far apart two round numbers are. In particular, in contrast to bounded timestamping, bounded round numbers can be equal. Also, as will be shown in the full paper, bounded timestamping can be implemented using the round number abstraction and one additional single-writer-multi-reader register per process, the size of a label. We see no similarly simple construction of bounded round numbers from bounded timestamps.

A first step in bounding a round-based algorithm was taken by Attiya, Dolev and Shavit [5]. They "hand-crafted" the unbounded round-based randomized consensus algorithm of Aspnes and Herlihy [2] into a bounded algorithm. This bounded construction is not a direct implementation of the round number structure of the original algorithm, and therefore its proof of correctness requires reasoning about functional correctness and boundedness simultaneously. The same is true of Herlihy's bounded construction for long-lived objects [13]. In particular, neither of these techniques could be used to bound algorithms in which a process's behavior depends on certain finite-state information about its round number, such as its parity. Moreover, the techniques are based on snapshot scan, for which no known deterministic bounded algorithm runs faster than $\Omega(n^2)$.[2]

1.2 Applications

Our transformation can be applied to numerous unbounded algorithms in the literature, yielding bounded algorithms with matching performance. Here we give several examples.

The first application is to the fastest known algorithm for randomized consensus, due to Aspnes and Waarts [4]. This algorithm runs in $O(n \log^2 n)$ steps and has an unbounded round structure introduced by Aspnes and Herlihy [2]. All known bounded solutions to the problem require $\Omega(n^2)$ steps. It may be possible to bound the algorithm of [4] using a similar technique to the one of [5], but this may increase the running time of the algorithm since it relies on atomic snapshots. As shown in the full paper, our transformation can be applied to a slightly modified version of the unbounded consensus protocol of [4] to give a bounded consensus with matching running time that uses $O(n \log n)$ size registers. This closes the gap between the fastest unbounded and bounded known solutions for randomized consensus.

A second application is to the new unbounded atomic snapshot algorithm of Attiya and Rachman [7]. This algorithm requires $O(n \log n)$ steps to perform an update or take a snapshot, improving on the $\Omega(n^2)$ complexity of earlier algorithms. This algorithm can be bounded using our bounded round number abstraction.

The earliest atomic snapshot algorithm in the literature, due to Aspnes and Herlihy [3], is unbounded and requires $O(n^2)$ steps to scan the memory or to update a value. Attempts to bound this algorithm led to a totally different snapshot algorithm [1] with matching running time and registers of size $O(nv)$, where v is the size of a value. Using our transformation, we show that the earlier algorithm can indeed be bounded, deriving a bounded snapshot algorithm whose time complexity matches the other two and that uses registers of size $\max\{O(nv), O(n \log n)\}$.

In the above applications the bounded round numbers were used to eliminate the use of variables that may assume an unbounded number of values. Bounded round numbers can sometimes be used to eliminate a second kind of unboundedness: using an unbounded number of variables. This kind of unboundedness typically arises when designing long-lived data objects. In [13], Herlihy implemented a long-lived randomized wait-free data object by letting the processes execute an unbounded sequence of consensus protocols, called *rounds*, which determine the order in which concurrent operations are applied. Each consensus round uses a different variable tagged by the round number. Bounding the number of variables needed can be done by tagging each variable with a bounded round number. The original algorithm used a similar technique to that used by Attiya, Dolev, and Shavit [5] to bound the round numbers; our bounded round number abstraction can be applied directly to give a simpler and more efficient construction.

Additional applications such as generalized approximate agreement [12] and randomized snapshots [6], are described in the full paper.

[2] Recently, Attiya and Rachman [7] developed an *unbounded* atomic snapshot algorithm that runs in $O(n \log n)$. They observe that our construction can be used to bound their algorithm, as well as the randomized atomic snapshot algorithms appearing in [6].

2 Round Number Abstraction

Many shared memory algorithms proceed in asynchronous rounds of communication. Informally, the round number counts the number of times a block of code has been performed. Initially all processes are at round 1. At each round the process performs a computation, and then advances its round number to a new round. Often a process's actions do not depend on its exact round number but only on its distance from the round numbers of other processes.

Therefore we can think of these round-based algorithms as using a round number *abstraction* that associates each process with its current round number. Round numbers have the order type of the integers. This abstraction provides four basic operations:

- A process may read another process's round number from a shared variable.
- The **compare** operation takes two round numbers. If the absolute value of their difference is less than or equal to some fixed k, then it returns their difference, otherwise it returns $k+1$.
- The **advance** operation takes a set of round numbers S $= \{r_1, \ldots, r_n\}$, and returns a new round number for p equal to $\max\{r_p + 1, \max(S)\}$.
- A process may write its new round number to a shared variable.

We require that these operations be used in the following disciplined way. Each process starts a round by *collecting* the current round numbers into a set S (i.e., reading them one-at-a-time in an arbitrary order), applying **compare** to elements of S, applying **advance** to S to generate a new round number, and finally installing that newly-generated round number for itself. All round-based shared memory algorithms known to us either use this round number abstraction in this way, or can be easily modified to do so.

Notice that if we were to replace the **advance** operation with an **increment** operation that simply returns the next round number, then no bounded implementation would be possible. Once a process falls ℓ rounds behind another, then the slower process must perform at least ℓ increments to catch up. Since ℓ cannot be bounded, neither can any implementation.

3 The Unbounded Algorithm

In this section we give an unbounded *intermediate algorithm* for the round number abstraction in which round numbers are implemented by *round vectors* and the abstract **advance** and **compare** operations are implemented by the *intermediate advance* and *intermediate compare* operations respectively. In Section 4, we show how to bound this intermediate algorithm.

3.1 Intuition

To argue that the intermediate algorithm correctly implements the round number abstraction, we will give a refinement mapping that carries round vectors to round numbers. Any execution of a protocol using the intermediate algorithm corresponds to some execution of that protocol using the round number abstraction. Moreover, the round vector returned by each call to **intermediate advance** maps to the round number returned by the corresponding call to **advance**, and the difference observed by each call to **intermediate compare** is the same as that observed by the corresponding call to **compare**.

Recall that a process performing the **advance** operation, may either increase its current round number or copy the round number of another process. In the first case we say that the process performs a *positive advance*, whereas in the second case we say that it performs a *flat advance*. The refinement mapping carries the initial round vectors to round 1. In general, if r_b is taken by a positive advance, then it is mapped to $r+1$, where r is the largest image under the mapping of the round vectors collected by b when it generated r_b. If r_b was taken by b through a flat **intermediate advance** by copying a round vector r_c, then r_b is mapped to the same round as r_c is mapped.

3.2 The Construction

Each process has a current round vector and a current private value. Each process draws its private values from a separate local pool of values, and thus has its *own* private values. Private values of different processes are always considered distinct. A process introduces a new current value for itself whenever it performs a positive **intermediate advance**. For the intermediate algorithm, the values are simply integers, and the pool initially contains all natural numbers. Over time, the values introduced by a process form a strictly increasing sequence. Thus, a *more recent* value for a process is a larger one. Not every element in the pool will necessarily be introduced; sometimes numbers are skipped.

Process b's current round vector, denoted by r_b, contains a vector of n entries and two additional fields. The

c-th entry of the vector, denoted by $r_b[c]$, is a record containing the following fields. The *recent* field of $r_b[c]$ contains a private value of c, the *count* field contains an integer between 1 and $k+2$, and the *leader* field is a boolean variable. The first additional field, called the *source*, contains the id of a process; the second additional field, called *val*, contains a private value of the process named in the source field. We refer to these fields of $r_b[c]$ as $r_b[c].recent$, $r_b[c].count$, and $r_b[c].leader$, and to the additional fields as $r_b.source$ and $r_b.val$. Initially, $r_b[c] = (0, 1, True)$ for all c, $r_b.source = b$, and $r_b.val = 1$.

Denote by r_b^i the ith round vector of process b. There is a total order on the round vectors defined by the round numbers to which they are mapped. The *difference* between two round vectors is defined as the difference between the round numbers to which they are mapped. Given a set of n round vectors, one for each process, we say that a process is a *leader* if its round vector is maximal in the set. In this case we also say that the round vector is a leader. We say that a round vector r_b^i *observed* a round vector r_c^j if r_b^i was taken by b through a positive **intermediate advance** and r_c^j was one of the round vectors collected in that **intermediate advance**, or r_b^i was copied by b from some r_d through a flat **intermediate advance** and r_d observed r_c^j. A private value of a process is observed by r_b^i if it was contained in any vector observed by r_b^i. A vector (value) is observed by an advance operation if it is observed by the round vector created by this advance.

Roughly speaking, r_b^i's entries contain the following values. $r_b^i[c].recent$ contains the most recent private value of c observed by r_b^i. $r_b^i[c].count$ estimates intuitively how many times b moved since the round in which c introduced the private value appearing in $r_b^i[c].recent$. Slightly more precisely, $r_b^i[c].count$ gives the length of a sequence of positive **intermediate advances**, starting with the positive **intermediate advance** of c that introduced the value appearing in $r_b^i[c].recent$, and ending with the positive **intermediate advance** that created r_b^i, where each element in the sequence (except the first) observed its predecessor. (Process b may have taken r_b^i through a flat **intermediate advance**; in which case by the 'positive **intermediate advance** that created r_b^i' we refer to the positive **intermediate advance** that created the round vector that was finally copied into r_b^i.) $r_b^i[c].leader$ is *True* if and only if c was a leader in the set observed by r_b^i and in addition, the round vector of c in that set was taken by c through a positive **intermediate advance**. The field $r_b^i.source$ contains the name of the process that created r_b^i through a positive **intermediate advance**, while $r_b^i.val$ contains the new private value that $r_b^i.source$ introduced for itself in the positive **intermediate advance** that created r_b^i.

The code for the **intermediate compare** operation appears in Figure 1. The **intermediate compare** operation correctly returns differences up to k. For differences greater than k it returns $\pm(k+1)$. A difference of $\pm(k+1)$ is not guaranteed to be accurate, although the sign is always correct and the magnitude returned is a lower bound on the magnitude of the actual difference.

Define the *c-difference* of r_1 from r_2 as $(r_1[c].recent - r_2[c].recent) + (r_1[c].count - r_2[c].count)$ provided $r_1[c].recent \geq r_2[c].recent$, and -1 otherwise. Intuitively, the c-difference of r_1 from r_2 says how many times process 1 moved since the time process 2 took r_2 upto the time process 1 took r_1, according to a third (not necessarily distinct) party c. In order to find how far ahead a given round vector r_1 is of another round vector r_2, **intermediate compare** looks at all their c-differences for which $r_2[c].leader = True$ and takes the maximum.

The code for the **intermediate advance** operation appears in Figure 4. In order to generate a new round vector the process, say b, is first required to identify the leaders. The code for determining whether a process is a leader appears in Figure 2. If b is not among the leaders, it takes as its own new round vector a round vector of one of the leaders. Otherwise, it generates a succesor to its current round vector as described in Figure 3. Roughly speaking, the round vector generated by b as a successor to its current round vector shows that, according to each process c, process b made at least one more move than indicated by any other current round vector.

3.3 Proof of Correctness

Given an execution of an algorithm that uses the round-based abstraction, and an execution of the corresponding intermediate algorithm, we say that the two executions *correspond* if (1) operations related to the round numbers implementation (**advances**, **compares**, reads and writes of round vectors/numbers) appear in both executions in the same time intervals; and (2) operations that are unrelated to the round number abstraction, such as simple reads, writes and coin flips, appear in both executions in the same time intervals, and return the same results. Recall that by definition, in an algorithm that uses the round number abstraction, the actions of a process do not depend on its exact round number but only on its distance from the round numbers of the other processes, and, perhaps, on some finite-state information, such as the parity of the round.

We view a process' initial round vector (resp., round number), as being returned by a positive **intermediate advance** (resp., **advance**). Thus, to prove that our intermediate construction implements the round number abstaction, it is enough to show:

Theorem 3.1 *Consider an execution E of the intermediate algorithm. There is a corresponding execution E' of the original algorithm such that the following hold:*

1. *The round vector returned by each call to* **intermediate advance** *in E maps, under the refinement mapping defined in Section 3.1, to the round number returned by the corresponding call to* **advance** *in E'.*
2. *The difference observed by each call to* **intermediate compare** *in E is the same as the one that is observed by the corresponding call to* **compare** *in E'.*

Proof: Consider the first write of a round vector that is mapped to r. Assume this write is done by process fr. ('fr' stands for 'the first process to write a round vector mapped to r'.) Observe that this round vector must have been taken through a positive **intermediate advance**, and hence it must contain a new private value for fr. Denote this private value by v_{fr}. We prove the theorem together with two additional technical statements. Define the *c-distance of round vector r_1 from round vector r_c* to be $r_1[c].recent + r_1[c].count - r_c.val$ provided $r_1[c].recent \geq r_c.val$ and $r_c.source = c$ (that is, provided r_1 observed r_c directly or indirectly and r_c was taken by a positive **intermediate advance**). If $r_1[c].recent < r_c.val$ (that is, if r_1 did not observe r_c), of if $r_c.source \neq c$, then define the c-distance to be $\perp$.

3. *Let r_c^l be taken by c through a positive* **intermediate advance** *and mapped to round r. For all $i \geq 1$, if the c-distance of r_b^j from r_c^l is i, then r_b^j is mapped to a round $\geq r + i$.*
4. *Let fr be the first process to write a round vector r_{fr} that is mapped to round r. For all $1 \leq i \leq k+2$, if a round vector r_b^j is mapped to $r+i$, then the fr-distance of r_b^j from r_{fr} is $\geq i$, and if in addition, $i = 1$, then $r_b^j[fr].leader = True$. If a round vector r_b^j is mapped to $r + i$ for $i \geq k + 2$, then the fr-distance of r_b^j from r_{fr} is at least $k + 2$.*

We define a *step* as (1) the performance of one of the operations that are not related to the implementation of the round number abstraction, or (2) an invocation or completion of one of the operations related to the round number abstractions (we separate each of the latter to two steps since they are not necessarily atomic). Let (E, t) denote the prefix of execution E upto the first point in which t steps are completed. We construct E' inductively so that for any t, (E', t) corresponds to (E, t) so that all four statements hold.

$(E', 0)$ will be the initial state of an execution of the original algorithm. We first show that $(E', 0)$ corresponds to $(E, 0)$ and all four statements hold. Recall that by convention, we view all initial round vectors/numbers as taken by positive **advance** just before the execution begins. So our base case considers each of the **intermediate advance** that yields an initial round vector. By convention, each initial vector is mapped to round number 1. Thus, each initial call to **intermediate advance** returns an initial round vector that is mapped to round number 1. On the other hand, the initial round numbers in E' are also 1, and therefore also an initial call to **advance** returns round number 1, and Statement 1 follows. Statement 2 holds vacuously, since there have been no invocations of **intermediate compare** at $(E, 0)$. Statement 3 holds vacuously since for any c, the c-distance between any two initial round vectors is 0. Statement 4 holds vacuously since it considers round vectors that are mapped to different round numbers, but all initial round vectors are mapped to the same round number.

For the inductive step, assume (E', t) corresponds to (E, t) so that the four statements hold, and we will show how to extend (E', t) so that $(E', t + 1)$ corresponds to $(E, t+1)$ and the four statements continue to hold. Consider the $(t + 1)$th step in E. Since (E', t) corresponds to (E, t) so that all four statements hold, an analogous step can be taken also as step $t + 1$ of E'. That is, if step $t + 1$ of E is a coin flip of process p, there is an extension of E' in which step $t + 1$ is the same coin flip of process p with the same result; if step $t + 1$ of E is a simple read (write) operation of a certain register done by process p, again there is an extension of E' in which step $t + 1$ is the same read (write) and with the same result. Similarly, if step $t + 1$ in E is an invocation or a completion of an **intermediate advance**, **intermediate compare**, read or write of a round vector, there is an extension of E' in which step $t+1$ is the invocation or a completion of an **advance**, **compare**, read or write of a round number respectively. Moreover, observe that if step $t + 1$ of E is a completion of a read that returns some round vector r_i, then there is an extension of E' in which step $t+1$ is a completion of a read that returns the round number ri to which r_i is mapped.

Clearly, if step $t + 1$ of E is not a return of an **intermediate advance** or a return of an **intermedi-**

ate compare, then $(E', t+1)$ constructed above corresponds to $(E, t+1)$ and the four statements hold. Therefore, to complete the proof we have to show that if step $t+1$ of E is the completion of an **intermediate advance** or an **intermediate compare** then Statements 1, 2, 3, and 4 still hold. Specifically, if step $t+1$ of E is the return of an **intermediate advance** we need only prove Statements 1, 3, and 4. If step $t+1$ is the return of an **intermediate compare**, we need only prove Statement 2.

Assume first that the $t+1$st step is a return of an **intermediate advance** of some process, say process p. Denote this **intermediate advance** by A. Consider the extension of E' in which the $t+1$ step is a return of an **advance** operation done by p. Denote this **advance** by A'. Denote the set of round vectors that are the input of A by: $r_1, \ldots, r_n$, and for each i, denote by ri the round number to which r_i is mapped. Recall that the input to A is the result of a collect in E preceding A, and the input to A' is the result of a collect in E' preceding A'. We observed above that the set returned by a sequence of reads in E' is simply the round numbers to which the round vectors returned by the corresponding sequence of reads in E are mapped. Therefore, the input given to A' will be the round numbers $r1, \ldots, rn$, to which the above $r_1, \ldots, r_n$ given to A are mapped.

We will prove Statements 1, 3, and 4 for the case where the **intermediate advance** returned in A is a positive **intermediate advance** of p. The case in which the **intermediate advance** returned in A is flat will immediately follow.

Proof of Statement 1

Recall that the property of a round vector being a leader among a set of round vectors is determined by the mapping. If a round vector is detected by **intermediate advance** as being a leader, then by the inductive hypothesis the vector is truly a leader, as determined by the mapping.

Now, by definition, when a process performs a positive **intermediate advance**, its new round vector is mapped to a round number that is greater by 1 than the maximum among all round vectors observed by the **intermediate advance** (more precisely, greater by 1 than the maximum among all the round numbers to which the round vectors observed by this **intermediate advance** are mapped). Since A is a positive **intermediate advance** of process p, A must have detected round vector r_p as maximal among the round vectors observed by it. Thus, the new round vector returned by A will be mapped to $rp+1$, where rp is the round number to which r_p was mapped when given to A. It remains to show that the new round number returned by A' is also $rp+1$. However, since A detected r_p as a leader using invocations of the **intermediate compare** operation, the inductive hypothesis implies that A' will detect rp as a leader using the corresponding invocations of the **compare** operation. Thus also A' will be a positive **advance**, and hence will return round number $rp+1$, and we are done.

Proof of Statement 3

Let r'_p denote the new round vector returned for p by a positive **intermediate advance** A, and let v_p denote $r'_p.val$. Note that since A is positive, $r'_p.source = p$ and $r'_p.val$ is a private value of p.

Clearly, the inductive hypothesis implies that the statement holds for any $b, c \neq p$. If $c = b = p$, the statement holds vacuously since the c-distance of a vector from itself is less than 1. Also if $c = p \neq b$ the claim vacuously holds since by definition of a c-distance, r^j_b can be of p-distance > 0 from r'_p only if the value for p in $r^j_b[p].recent$ is at least as recent as v_p, and this can't happen at $(E, t+1)$, since r'_p has just been returned.

To complete the proof of the statement, we consider the case where $p = b \neq c$. Thus, assume that r^l_c is mapped to round r, the c-distance of r'_p from r^l_c is i, for some $i \geq 1$, and we will show that r'_p is mapped to round $\geq r+i$.

Let $v_c = r^l_c.val$. (Since r^l_c was taken by a positive **intermediate advance**, v_c is a private value of process c.) By definition of c-distance it follows that $r'_p[c].recent + r'_p[c].count = v_c + i$. We proceed by induction on i. If $i = 1$ then r'_p observed r^l_c, which is mapped to r, so r'_p is mapped at least to $r+1$. If $i > 1$ then, by the construction of the **successor** operation, r'_p observed some r_d with c-distance to r^l_c of $i-1$. By induction, r_d is mapped at least to $r+i-1$. It follows by definition of the mapping and the fact that r'_p is produced by a positive **intermediate advance** that r'_p is mapped at least to $r+i$.

Proof of Statement 4

The inductive hypothesis implies that the statement holds for any $fr, b \neq p$. If $fr = p$, the statement holds vacuously since this means that p is the first process to write a round vector that is mapped to round number r, and hence at $(E, t+1)$ there is no process whose round vector is mapped to a round number greater than r. Therefore, we only need to consider the case where $b = p \neq fr$.

Let r'_p denote the new round vector for p returned at $(E, t+1)$. Let r'_p be mapped to round $r+i$, where $i \geq 1$.

First consider the case $i > 1$. Since r'_p is mapped to $r+i$, r'_p must have observed some round vector, say r_d, that is mapped to $r+i-1$. The inductive hypothesis on the statement implies that the fr-distance of r_d is at least $i-1$, if $i-1 \leq k+2$, and at least $k+2$ otherwise. Lines 2, 3, and 4 of the **successor** operation imply therefore that if $i \leq k+1$, the fr-distance of r'_p is at least i and otherwise, for $i > k+1$, its fr-distance is at least $k+2$, and the claim follows.

Next consider the case where $i = 1$, that is, where r'_p is mapped to $r+1$. By definition of the mapping this means that the maximal round vector observed by r'_p was mapped to r. Since r'_p was returned by a positive **intermediate advance** A, r_p must have been detected as a leader by A. The inductive hypothesis implies that r_p was indeed maximal in the set of round vectors observed by A. Hence r_p is also mapped to r. Since A starts only after r_p is written, and the latter is written not before fr writes r_{fr}, we have that A starts only after r_{fr} is written. Therefore, r'_p must have observed r_{fr}. Recall that r_{fr} was taken by a positive **intermediate advance**. It follows from the code for the **successor** operation that $r'_p[fr].recent$ is at least as recent as $r_{fr}.val$, the private value that r_{fr} has for fr. Thus, the fr-distance of r'_p from r_{fr} is at least 1.

Next we show that $r'_p[fr].leader = True$. Since fr is the first process to write a round vector mapped to r, it follows from the definition of the mapping that fr must have taken r_{fr} through a positive **intermediate advance**. Hence, r_{fr} is the result of a **successor** operation done by the **intermediate advance** that returned r_{fr}, and it follows from line 9 of the **successor** operation that $r_{fr}.source = fr$. In addition, remember that r_{fr} was a leader among the set of round vectors observed by A, and hence by the inductive hypothesis must have been detected as such by A. Since $r_{fr}.source = fr$, and r_{fr} is detected by A as a leader, it follows from Lines 5 and 6 of the successor operation that $r_p[fr].leader = True$.

Proof of Statement 2:

Now we assume that the $t+1$st step of E is a return of an **intermediate compare** operation done by some process, say process p, and we will prove Statement 2. Denote this **intermediate compare** by C. Consider the extension of E' in which the $t+1$st step is a return of an **intermediate compare** operation done by p. Denote this **intermediate compare** by C'. Denote the two round vectors that are the input of C by r_1, r_2, and for each i, denote by ri the round number to which r_i is mapped. Recall that the input to C is the result of a sequence of reads in E preceding C, and the input to C' is the result of a sequence of reads in E' preceding C'. We observed above that the set returned by a collect in E' is simply the round numbers to which the round vectors returned by the corresponding collect in E are mapped. Therefore, the input given to C' will be the round numbers $r1, r2$, to which the above r_1, r_2 given to C are mapped. Without loss of generality, assume $r1 \geq r2$.

To complete the proof we need to show that the value returned by C is exactly the difference $r1 - r2$, unless this difference exceeds k, in which case the value returned is $k+1$. Denote by A the **intermediate advance** operation of process 2 that returned r_2. The **intermediate compare** operation implies that the value returned as the difference of r_1 from r_2 is the maximum difference among all c-differences of r_1 from r_2 computed for each c such that $r_c.leader = True$. Note that this means that r_c was obtained by a positive **intermediate advance** and was observed by r_2 to be a leader. First we show that for each such r_c the c-difference of r_1 from r_2 is at most $r1 - r2$.

Since r_2 is mapped to $r2$, it follows from the definition of the mapping that the maximal round vector observed by r_2 was mapped to $r2-1$. Since r_c was observed by r_2 as a leader, and since by the inductive hypothesis this means that r_c was indeed a leader among the set of round vectors observed by r_2, we have that r_c is mapped to $r2-1$. Therefore, it follows from Statement 3 that the c distance of r_1 from r_c is at most $r1 - r2 + 1$. On the other hand, since r_2 observed r_c, the c-distance of r_2 from r_c is at least 1. Finally, if $r_1[c].recent - r_2[c].recent < 0$, then the c-difference of r_1 and r_2 is $-1 \leq r1 - r2$. If on the other hand, $r_1[c].recent - r_2[c].recent \geq 0$ then, since r_2 observed r_c so did r_1, and therefore the c-difference of r_1 and r_2 is exactly the c-distance of r_1 from r_c minus the c-distance of r_2 from r_c. Thus, the c-difference of $r1$ from $r2$ is at most $r1 - r2$.

Denote by fr the first process to write a round vector mapped to $r2-1$. Combining Statements 3 and 4 we get that the fr-distance of r_2 from r_{fr} is exactly 1, and $r_2[fr].leader = True$. Combining Statement 3 and 4 again we also get that the fr-distance of $r1$ from r_{fr} is exactly $r1 - r2 + 1$ if $r1 - r2 + 1 \leq k+1$, and at least $k+2$ otherwise. Thus **intermediate compare**(r_1, r_2) will return the value $r1 - r2$ provided this value is not greater than k, and otherwise will return $k+1$.

■

intermediate compare(r_1, r_2)

```
1. For each c
2.     If r2[c].leader = True
       and r1[c].recent ≥ r2[c].recent
3.         then temp[c] :=
           (r1[c].recent − r2[c].recent)
           +(r1[c].count − r2[c].count)
4.         else temp[c] := −1
5.     MAX := max { temp[1], ..., temp[n]}
6.     If MAX ≥ 0
7.         then return ( min { MAX, k + 1} )
8.         else for each c
9.             If r1[c].leader = True
               and r2[c].recent ≥ r1[c].recent
10.                then temp[c] :=
                   (r2[c].recent − r1[c].recent)
                   +(r2[c].count − r1[c].count)
11.        MAX := max { temp[1], ..., temp[n]}
12.        Return(-min { MAX, k + 1})
```

Figure 1: The **intermediate compare** Operation

4 Bounding the Unbounded Construction

In this section we show how to implement the intermediate algorithm using only a bounded number of values. The unbounded round vectors will be implemented by bounded round vectors and the **intermediate advance** and **intermediate compare** operations will be implemented by **bounded advance** and **bounded compare** operations respectively. We refer to the resulting algorithm as the *bounded* algorithm.

leader$(c, \{r_1, r_2, \ldots, r_n\})$

```
1. Lead := True
2. For each d ≠ c:
3.     temp:= intermediate compare(r_c, r_d)
4.     If temp< 0
5.         then Lead := False
6. Return(Lead)
```

Figure 2: The **leader** Primitive

successor$(\{r_1, r_2, \ldots, r_n\})$

```
1. For each c
2.     temp[c].recent :=
       max{r1[c].recent, ..., rn[c].recent}
2.5.   temp[c].recent :=
           max{r1.val, ..., rn.val, temp[c].recent}
           where the max is over all ri
           for which ri.source = c
3.     temp[c].count :=
           max{r1[c].count, ..., rn[c].count, 0} + 1
           where the max is taken over all ri
           for which ri[c].recent ≥ temp[c].recent
4.     If temp[c].count > k + 1,
           then temp[c].count := k + 2
5.     If leader(c, {r1, r2, ..., rn}) and
       rc[c].source = c
6.         then temp[c].leader := True
7.         else temp[c].leader := False
8. temp.val := temp[b].recent + 1
9. temp.source := b
11. Return (temp);
```

Figure 3: The **successor** Primitive; Code for Process b

The running time of **bounded advance** and **bounded compare** is linear, and hence is the same as the running time of **advance** and **compare** of the original algorithm. Thus, bounding an algorithm that uses the round number abstraction preserves its running time. The registers are of size $O(n^3 \log n)$. In the next section we show that for round-based algorithms that satisfy some additional restrictions, the registers' size can be reduced to $O(n \log n)$, while preserving linear time complexity. All round-based algorithms with which we are familiar satisfy these restrictions.

intermediate advance$(\{r_1, r_2, \ldots, r_n\})$

```
1. If leader(b, {r1, r2, ..., rn})
2.     then return(successor({r1, r2, ..., rn}))
3.     else for each c ≠ b
4.         If leader(c, {r1, r2, ..., rn})
5.             then return(rc)
```

Figure 4: The **intermediate advance** Operation; Code for Process b

The transformation described in this section is achieved in a straightforward fashion by using the **Traceable Use** abstraction of [10] to recycle the private values. In the bounded implementation, each process maintains a bounded number of private values and a record of the distance between each pair of values. We use the **Traceable Use** abstraction to enable the process to determine which of its private values currently exist in the system (*i.e.,* may appear even if they are not reissued).

The **Traceable Use** algorithm provides a **traceable read** operation for reading round vectors, and a **traceable write** operation for writing them. To transform the code for the intermediate algorithm to the code of the bounded algorithm, replace each read of a round vector with a **traceable read** and each write with a **traceable write**.

In addition to having **traceable read** and **traceable write** replace round vector reads and writes, the code for **bounded advance** and **bounded compare** is slightly modified as follows. Each process maintains a single *distance register* containing the distance information among its private values. The register contains an array of pairs (v, j) so that v is a private value and j describes the positive distance of v from the private value in the preceding pair if this distance is less than $k+2$; otherwise, $j = k+2$.

When a process performs a **bounded advance** or a **bounded compare**, it first collects the round vectors of all the processes (using **traceable read**) and then collects all the distance registers. It then has all the information it needs to compute the distance relationships among the set of round vectors.

If, after comparing the collected round numbers, the process decides to increase its round vector (*i.e.,* to perform a positive **intermediate advance**), it does not issue a new private value but reissues one of its private values that is not in the system anymore; the distance between this private value and other existing private values is recorded correctly in its distance register. This completes the description of the bounded implementation of the round number abstraction.

It is easily seen that **bounded advance** and **bounded compare** implement **intermediate advance** and **intermediate compare** respectively and hence:

Theorem 4.1 *Consider an execution E of the bounded algorithm. There is a corresponding execution E' of the original algorithm such that the following hold:*

1. *The round vector returned by each call to* **bounded advance** *in E maps to the round number returned by the corresponding call to* **advance** *in E'.*
2. *The difference observed by each call to* **bounded compare** *in E is the same as the one that is observed by the corresponding call to* **compare** *in E'.*

Finally we discuss the complexity of the bounded algorithm. Using the **Traceable Use** implementation of Dwork and Waarts, the write or read of a single round vector requires n steps. It is also possible to collect n round vectors, one for each process, in n steps, using an operation they call **traceable collect**. Employing the **Traceable Use** involves also using its **garbage collection** operation to determine which values are in use. We say that a private value travels a distance d if there was a chain of d processes each of which copied this value from the process preceding it in this chain. The complexity of **garbage collection** depends on how far a value can 'travel'.[3] In particular, if a private value can travel distance d then **garbage collection** performs d collects, during each of which it reads $O(n^2)$ registers.

It can be shown that in our bounded algorithm a value never travels a distance greater than $O(n)$. Using this observation, we see that each process needs to have a pool of $\Theta(n^3)$ private values. The pool is split into two halves. At any time (except the very first time the pool is used) one half is exhausted and the other half active. Garbage collection is performed on the exhausted half while the active half is being consumed. **Garbage collection** requires n big collects each of which reads $O(n^2)$ registers. Each big collect may return at most $6n^2$ values as being in use. Thus, at least n^3 values are returned to the exhausted half for re-use each time **garbage collection** is performed. The cost of **garbage collection** is $O(n^3)$ steps. These steps (and not just their cost) are amortized over the $O(n^3)$ **bounded advance** operations needed to exhaust half of the pool, thereby preserving linear time complexity.

5 Reducing the Register Size

In this section we describe implementations of **advance** and **compare**, in which the size of the registers is only $O(n \log n)$, while the running time of each operation remains linear. The implementation is tailored for algorithms that satisfy some additional restrictions.

[3] The implementation of [10] assumes that this distance is at most two. As mentioned in that paper, the implementation can easily be extended to handle travelling for further distances.

In particular, we consider algorithms in which a process does not need to measure the distances of arbitrary pairs of rounds, but only the distance between rounds that are at most k behind the leader. During an **advance** operation a process must invoke the **compare** operation to determine if it is a leader. If during such a **compare** operation the process detects that none of the round numbers collected by it is a leader anymore, it has only to acquire the identity of any one process that has a round number at least as large as the maximum round number at the beginning of the **advance** operation. All round-based algorithms known to us satisfy these restrictions.

We first describe how to implement the **intermediate advance** operation in linear time using registers of size $O(n \log n)$. We refer to this implementation as the *efficient bounded advance* operation.

When a process c performs an **intermediate advance** operation, it is required for each process b to find the maximum value for b among all the values for b that it has seen. If the distance information is maintained in registers of size at most $O(n \log n)$, then this could potentially require n reads for each other process, for a total of $O(n^2)$ reads. However, inspection of the **intermediate advance** operation immediately shows that all that process c really needs to know is (1) whether or not c itself is a leader; if so, (2) for each process b, has b introduced for itself a newer value than the one c had for b; and if c is not a leader, (3) the identity of any one process that has a round vector at least as large as the maximum round vector at the time c began the **intermediate advance** operation.

To this end, we define a special auxiliary distance register DISTANCE$_b$ for each process b. When b performs a positive **intermediate advance**, it writes a new round number, which includes a new value v_b that b introduces for itself. Just before doing this write, b writes to DISTANCE$_b$ v_b and the list of b's 3 previous values, together with their distances from v_b ($k+2$ for each distance greater than $k+1$).

Now, when a process c performs an **efficient bounded advance** operation, it first collects, using **traceable collect**, the round vectors of all the other processes. It then collects (using a simple collect, *i.e.*, not using **Traceable Use**) all the DISTANCE registers. Let S be the set of round vectors collected, and let M be the maximum round vector in S. Let $\mathcal{O}$ be the set of DISTANCE registers collected.

We say a round vector $r \in S$ is *ruled out by* $\mathcal{O}$ if for some b, $r[b].recent \notin$ DISTANCE$_b$. Round vectors in S that are not ruled out by $\mathcal{O}$ are called *contenders*.

First, c looks for a b such that for all $r \in S$, $r[b].recent$ contains neither b's most recent value v_b, as determined by $\mathcal{O}$, nor the first predecessor of this value, again as determined by $\mathcal{O}$. In other words, $r[b].recent = \hat{v}_b$ where $\hat{v}_b$ is not one of the two most recent values introduced by b for itself, as determined by DISTANCE$_b$. It then reads b's round number, using the **traceable read**, and returns the result as its new round vector. This is a flat **efficient bounded advance**.

As we show below, if no b as described above exists, then the set of contenders cannot be empty. Process c compares the contenders as described in the intermediate algorithm, using the information in $\mathcal{O}$ to do so, and from this computes M. If c's round number in S was ruled out, it takes M as its new round vector. This is a flat **efficient bounded advance**. If c's round vector in S was not ruled out, it compares this to M using the information in $\mathcal{O}$. If they are equal, then, using the information in $\mathcal{O}$, c determines the newest value seen in S for each process b, and computes the appropriate offset. This is a positive **efficient bounded advance**. Otherwise (c's round vector in S is less than M), c takes M as its new round vector, performing a flat **efficient bounded advance**.

Lemma 5.1 *Let $R \in S$, and let b be any process. If $R[b].recent = v_b$, then $M[b].recent = v'_b$ where v'_b is either v_b or one of its two predecessors.*

Proof: Suppose the lemma is false. We abuse notation by letting M also denote the process from which c obtained M. First, assume M was obtained by a positive advance. Then b introduced for itself three new values after M began the **traceable read** of b's round number during the **advance** that resulted in round vector M, and before c completed its scan. Thus, b made at least two positive advances in that interval. Let M be mapped to r. During its scan M had a round number mapped to $r-1$. Since b was performing positive advances, the first such advance completely within the interval is mapped to at least r. The second such advance is mapped to at least $r+1$. Thus, if $R[b].recent$ is three steps ahead of $M[b].recent$ then R is mapped to at least $r+1$, contradicting the fact that M is the maximum round number in S.

Finally, we can relax the assumption that M was obtained by a positive advance, simply by replacing the process M with the process that first constructed M during a positive advance. Nothing in the argument changes. ■

Corollary 5.1 *If M is ruled out, then there exists a b so that for all $R \in S$, $R[b].recent$ contains neither b's*

most recent value v_b, as determined by $\mathcal{O}$, nor the first predecessor of this value, again as determined by $\mathcal{O}$.

Lemma 5.2 *If for all $R \in S$, $R[b]$.recent contains neither b's most recent value v_b, as determined by $\mathcal{O}$, nor the first predecessor of this value, again as determined by $\mathcal{O}$, then during the* **efficient bounded advance** *operation of c, b has made a complete positive* **efficient bounded advance** *and wrote its resulting new round vector.*

Proof: It is immediate that b has introduced for itself two new values since the beginning of c's **advance** operation. Since b introduces a new value for itself only during a positive advance, at least the second value was introduced during a positive advance that takes place completely within the time interval of the **advance**. ■

Observe that the above lemma implies that the round vector b has when c completes its **efficient bounded advance** is at least as large as the maximum round vector at the time c began the **efficient bounded advance**. Putting this together, either we can find a process b that has taken a positive **efficient bounded advance** since the beginning of c's **advance** operation or M is not ruled out by $\mathcal{O}$, and will be recognized as the maximum and can be compared with c's round vector. In either case, **efficient bounded advance** implements **intermediate advance**, and we are done.

This completes the description of **efficient bounded advance**. The implementation of the restricted compare is analogous and omitted from the abstract.

Acknowledgement

The authors are grateful to Nir Shavit for encouraging this line of research.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots. Ninth ACM Symposium on Principles of Distributed Computing, 1990.

[2] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, September 1990.

[3] 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*, pages 340–349, July 1990.

[4] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 137–146, October 1992.

[5] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August 1989.

[6] H. Attiya, M. Herlihy, and O. Rachman. Efficient atomic snapshots using lattice agreement. Technical report, Technion, Haifa, Israel, 1992. A preliminary version appeared in proceedings of *the 6th International Workshop on Distributed Algorithms*, Haifa, Israel, November 1992, (A. Segall and S. Zaks, eds.), Lecture Notes in Computer Science #647, Springer-Verlag, pp. 35–53.

[7] H. Attiya and O. Rachman. Atomic Snapshots in $O(n \log n)$ Operations. *These Proceedings.*

[8] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible! In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 454–465, 1989.

[9] C. Dwork, M.P. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. In *In Proceedings of the Israeli Symposium on the Theory of Computing and Systems*, May 1992.

[10] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent time-stamp systems are comprehensible! In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 655–666, May 1992. Preliminary version appears in IBM Research Report RJ 8425, October 1991.

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

[12] M.P. Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, July 1991.

[13] M.P. Herlihy. Randomized wait-free concurrent objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, August 1991.

[14] A. Israeli and M. Li. Bounded time-stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 371–382, October 1987.

[15] P. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 223–243, 1986. Errata in SIGACT News 18(4), Summer, 1987.