A Simple, Memory-Efficient Bounded Concurrent Timestamping Algorithm

Vivek Shikaripura and Ajay D. Kshemkalyani

Computer Science Department, Univ. of Illinois at Chicago,
Chicago, IL 60607, USA

Abstract. Several constructions have been proposed for implementing a Bounded Concurrent Timestamp System (BCTS). Some constructions are based on a recursively defined *Precedence Graph*. Such constructions have been viewed as hard to understand and to prove correct. Other constructions that are based on the *Traceable Use* abstraction first proposed by Dwork and Waarts have been regarded as simple and have therefore been preferred. The Dwork-Waarts (DW) algorithm, however, is not space-efficient. Haldar and Vitanyi (HV) gave a more space-efficient construction based on Traceable Use, starting with only safe and regular registers as building blocks. In this paper, we present a new algorithm by making simple modifications to DW. Our algorithm is simple and is more memory-efficient than the DW and HV algorithms.

1 Introduction

We consider the problem of constructing a Bounded Concurrent Timestamp System (BCTS), which can be described briefly as follows. *There is a set of n asynchronous processors which can perform two types of operations,* ***Label****() and* ***Scan****(). The* ***Label****() operation corresponds to some event and generates a new timestamp (label, hereafter) for the processor performing it. The* ***Scan****() operation enables a processor to obtain a list of the current labels and to determine the order among them, consistent with the real-time order in which the labels were selected. The* ***Label****()/* ***Scan****() operations of a processor may be interleaved or concurrent with a* ***Label****()/* ***Scan****() operation of another processor. The problem is to construct such a system by using shared variables as building blocks but with the restriction that the size and number of all labels/shared variables/registers so used be bounded. The system must also be wait-free [5].*

Bounded Concurrent Timestamping is a well-studied problem. The reader is referred to [2,5] for an introduction. Bounded Concurrent Timestamping abstracts a large class of problems in concurrency control, notably Lamport's first-come first-served mutual exclusion [11], randomized consensus [1], MRMW atomic register construction [16] and fifo-l-exclusion [6].

There are many BCTS algorithms [3,4,5,8,10]. The basic technique used is to first construct an Unbounded Concurrent Timestamp System (UBCTS) in which processors assume monotonically increasing label values (a potentially unbounded number of them) and then convert the UBCTS into a Bounded

P. Bose and P. Morin (Eds.): ISAAC 2002, LNCS 2518, pp. 550–562, 2002.

Concurrent Timestamp System (BCTS) by employing a mechanism to recycle the labels. So far, two recycle mechanisms (and their variants) have been used.

1. *Precedence Graph*: In this mechanism, processors choose labels from the domain of a recursively defined, nested digraph called precedence graph. The domain consists of the set of labels (node names) in the precedence graph. To execute a label operation, a processor first collects all the node labels which have been chosen by other processors and then chooses its new label by selecting a free node such that it is the lowest node "dominating" the collected nodes. A scanner determines the order among the labels by following the edges of the nodes corresponding to the labels. Owing to some special properties of the graph, successive labeling operations are able to continuously choose new nodes without ever running out of nodes. This mechanism, first used by Israeli and Li [9] for sequential timestamp systems, was later extended and used in [4,3,7,10] for concurrent timestamp systems.
2. *Traceable Use*: In the precedence graph, there is a global pool of values from which processors choose their labels. In the Traceable Use mechanism, each processor maintains a separate, local pool of private values. To execute the labeling operation, a processor first collects the private value of every processor, chooses a new, unused private value for itself from its local pool and finally obtains a vector clock from these private values. This vector clock is the new label of the processor. So as to allow other processors to know which one of their private values has been collected, the labeler records in shared memory, every private value that it collects. This enables a processor to trace out which of its private values are not in use at any time by checking shared memory. With this arrangement, a labeler can safely recycle private values. A scanner determines the order among the labels by comparing the vector clocks. This mechanism, known as Traceable Use, is the basis of the algorithms in [5,8].

While the precedence graph mechanism has allowed construction of efficient algorithms, it has been regarded as hard to understand [8,7] and having high conceptual complexity [4,12,14]. The Traceable Use mechanism has been regarded as "simple", but it has not enabled space-efficient constructions so far, since long registers and large amount of shared memory space are required in its implementation. Despite this drawback, Traceable Use mechanism has enjoyed popularity since it renders simple, elegant constructions that are easy to understand and prove correct. We are thus motivated to seek improvements in algorithms [5,8], while retaining the simplicity with which they model BCTS. The presented work is such an attempt.

Several criteria have been used to compare BCTS algorithms.

- *Label/Register size*: The maximum size in bits of a register owned by any processor.
- *Label time complexity*: The maximum number of shared variable accesses in a single execution of the **Label**() operation.
- *Scan time complexity*: The maximum number of shared variable accesses in a single execution of the **Scan**() operation.

– *Memory overhead*: Every processor accesses two types of memory – shared memory and local memory. Shared memory includes all shared variables. Local memory includes local variables and private pools of values used by a processor, which are kept in local space.

Tables 1 and 2 compare the performance of the BCTS algorithms.

Table 1. Comparison of BCTS algorithms based on the precedence graph [8].

Algorithm	Register size (bits)	Label time complexity	Scan time complexity	Total shared space (bits)
Dolev-Shavit [3]	$O(n)$	$O(n)$	$O(n^2 \log n)$	$O(n^3)$
Gawlick et. al. [7]	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n^4)$
Israeli-Pinhasov [10]	$O(n^2)$	$O(n)$	$O(n)$	$O(n^4)$
Time-lapse [4]	$O(n)$	$O(n)$	$O(n)$	$O(n^3)$

Table 2. Comparison of BCTS algorithms based on the Traceable Use mechanism.

Algorithm	Register size (bits)	Label time complexity	Scan time complexity	Total shared space (bits)	Local space (bits)
Dwork-Waarts [5]	$O(n \log n)$	$O(n)$	$O(n)$	$O(n^5 \log n)$	$O(n^2 \log n)$
Haldar-Vitanyi [8]	$O(n \log n)$	$O(n)$	$O(n)$	$O(n^3 \log n)$	$O(n^2 \log n)$
This paper	$O(n \log n)$	$O(n)$	$O(n)$	$O(n^3 \log n)$	$O(n \log n)$

The DW algorithm [5] has three main drawbacks: (i) each processor uses $O(n^2)$ SWMR registers, so that the total overhead of shared memory is $O(n^5 \log n)$ SWSR safe bits, (ii) an extra, expensive garbage collection mechanism is required for tracking down private values not in use, and (iii) long registers of size $O(n \log n)$ bits are used in the construction. Haldar and Vitanyi (HV) [8] improved the first two drawbacks by giving an independent construction using only regular and safe registers as building blocks.

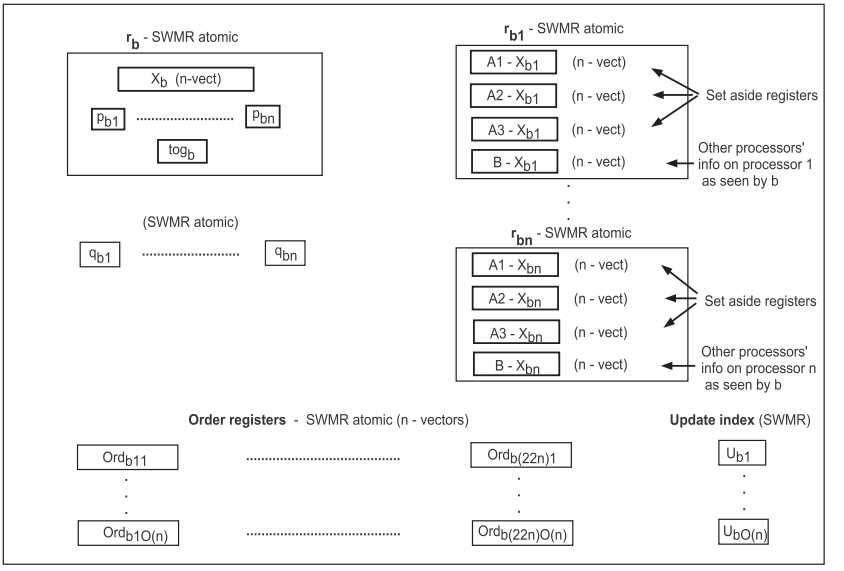
In the present work, we improve the algorithm in [5] by proposing a new, modified design for the **Label**() and **Scan**() operations. Unlike [5,8], while our **Label**() operation uses the RWRWR handshake technique of [5,13] to collect private values, our **Scan()** operation collects processor labels by capturing a snapshot of the labels as they existed at a single point in time [4,7]. Every processor in our construction uses two pools of integers, a private value pool and a tag pool, each of size $O(n)$ integers. The size of the private pool of values is $3n - 1$ integers in our case, which is better than the $2n^2$ in [8] and the $22n^2$ in [5]. Our algorithm is inspired by a detailed study of existing BCTS algorithms; we integrate many of the ideas embedded in the algorithms of [4,5,7] into our construction. As can be seen from the tables, although the Time-lapse algorithm

[4] based on the precedence graph is more efficient, it does not enjoy the simplicity of approach of our construction based on the Traceable Use abstraction.

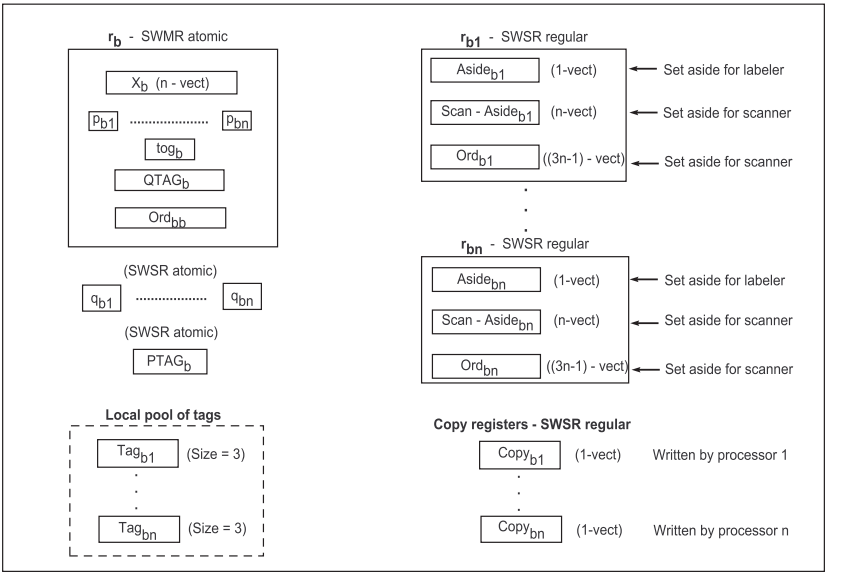
2 The Dwork-Waarts Construction

We first review the structure of shared memory in the DW algorithm. Figure 1 (top) sketches the shared variables associated with any processor b. X_b, known as the principal shared variable, is the label (timestamp) of processor b. It is a vector of length n whose components are the private values collected from every processor. The $A1{-}X_{b*}$, $A2{-}X_{b*}$, $A3{-}X_{b*}$, and $B{-}X_{b*}$ registers are auxiliary shared variables needed for the construction. The p_{b*}, q_{b*} and tog_b registers implement the RWRWR handshake mechanism of [5,13] which can detect concurrent writes while the processor b is reading. The Ord_{b*} registers maintain the ordering information among the pool of private values of b. As the shared variables X_b, tog_b and handshake bits p_{b*} are fields of one SWMR atomic register r_b, all of them can be read/written in one atomic operation. Similarly, $A1{-}X_{b*}$, $A2{-}X_{b*}$, $A3{-}X_{b*}$ and $B{-}X_{b*}$ can be read/written atomically from r_{b*}.

The central idea of the DW algorithm is as follows: Each processor b maintains an integer pool of private values – $v_{b(1)}, \ldots\ldots\ldots, v_{b(11n^2)}$ from which it draws values for its label. The relative order between $v_{b(i)}$, $v_{b(j)}$ (i.e., which of the two was taken up before the other), for all i, j, is stored in Order registers (Ord_{b*} in Figure 1) by b. Whenever b executes a labeling operation, it assumes a new label, a vector clock consisting of n components – the current private value collected from each processor plus a new private value v drawn from its own local pool (assuming it is not empty). The value v so drawn is now written in the Order registers Ord_{b*} as the largest of b's values. The key to the algorithm is to ensure that when b takes up v, v is indeed "new" in the sense that it does not currently appear and will not appear later in the label of any processor in the system. To see why this is critical, consider the following scenario. A scanner c collects labels from all processors but goes to sleep temporarily before determining their relative ordering. Meanwhile an updater b wishing to assume a new label, cannot take up any of its private values v_k which c may have obtained from b (or indirectly from another processor d). If b, before c woke up, were indeed to take up v_k and update register Ord_{b*} with v_k as the largest value, any of c's subsequent computations making reference to newly updated Ord_{b*} would be erroneous. Hence, b would have to make sure that it does not take up a value which is already in use. The problem of determining which values are in use in the system at any time is not trivial. The DW algorithm solves this problem by requiring that every private value read in an operation, either **Label()** or **Scan()**, be recorded in shared memory somewhere, hence, *all private values consumed in read operations are recorded in shared variables.* A processor can trace out which of its values are in use at any time, by scanning all the shared variables (this is known as garbage collection). Only values not found in shared memory (and hence not in use) are valid new values. It turns out that in order



Shared variables and registers at Processor b in the DW algorithm [1]



Shared variables and registers at Processor b in proposed algorithm

Fig. 1. Data structures for the DW algorithm [5] and the proposed algorithm.

Label() procedure, code for processor b
1. For all c, $val[]_c$ = **traceable-collect**($X_c[c]$)
// collect private values from all other processors except b in $val[]_c$
2. Choose v_b = smallest element in b's pool of available private values in Ord_{bb}
3. $new{-}l = (val_1, val_2, \ldots, v_b, \ldots, val_n)$
4. **traceable-write**(X_b, $new{-}l$) // write $new{-}l$ into X_b

Scan() procedure, code for processor b
1. The set of labels is S = **traceable-collect**(X)
2. To decide the order between two labels lab_i and lab_j:
Read the Order register Ord_{kb} where k is the most significant index in which lab_i differs from lab_j.
Do the same for all labels

traceable-collect(X) of RWRWR, code for processor b
Perform handshake
1. For each c, Collect p_{cb}
2. For each c, Write $q_{bc} = p_{cb}$

Remainder of the first Read-Write-Read
3. For each c, Collect $X_c[c]$, tog_c
4. For each c, Write $B{-}X_{bc} = X_1[c], X_2[c], \ldots, X_n[c]$
5. For each c, Collect $X_c[c]$, p_{cb}, tog_c
6. For each c, if $q_{bc} = p_{cb}$ and tog_c is unchanged since line 3
then $Y_c = X_c$
else $Y_c = null$

Remainder of the second Read-Write-Read
7. For each c for which $Y_c = null$
Write $B{-}X_{bc} = X_1[c], X_2[c], \ldots, X_n[c]$
8. For each c, Collect $X_c[c]$, p_{cb}, tog_c
9. For each c for which $Y_c = null$
if $q_{bc} = p_{cb}$ and tog_c is unchanged since line 5
then $Y_c = X_c$

10. For each c such that $Y_c = null$
Read $Y_c = A1{-}X_{cb}$
11. Return $(Y_1, Y_2, \ldots, Y_n)$

Fig. 2. Scan(), **Label()** and **traceable-collect** procedures, DW algorithm [5].

to implement this mechanism, known as Traceable Use, every processor must maintain a separate, private pool of size $22n^2$ integers.

Though simple, the DW algorithm has the following drawbacks identified in [8]. (1) If b takes up a new private value every time it executes a **Label**(), its local pool will soon become empty unless it stops once in a while to perform garbage collection, allowing it to return those private values which are not in use, back into its private pool. To avoid this situation, garbage collection is performed as an ongoing, background operation, a constant number of steps during every **Label**() operation on one half of the pool, while exhausting the other half. That is, garbage collection which is of $O(n^2)$ complexity, is amortized over **Label**()

operation, which is of O(n) complexity. This implies that **Label**() is not truly linear. (2) To avoid having to use a long register of size $O(n^2 \log n)$ bits for maintaining the ordering information among its $22n^2$ integers, b uses $22n$ SWMR Order registers $Ord_{b(1)}$ to $Ord_{b(22n)}$, each having n values. But distributing the ordering information over $22n$ separate SWMR Order registers does not allow b to update its Order registers atomically, which can be a problem if there is a concurrent read of Order registers by a scanner c. Hence, b maintains $O(n)$ sets of the Order registers and update indices $U_{b1} \ldots\ldots U_{bO(n)}$ to maintain the status (i.e., whether being read currently) of the corresponding Order register. Such an arrangement requires every processor b to use $O(n^2)$ SWMR registers. Implementing this system using fundamental level SWSR safe bits is not space-efficient. An analysis in [8] reveals that since it takes $O(n^2 \log n)$ SWSR safe bits to construct a single SWMR atomic register of size $O(n \log n)$, each processor b would need $O(n^4 \log n)$ SWSR safe bits and hence the whole system would need $O(n^5 \log n)$ SWSR safe bits. (3) Each processor needs to maintain a pool of values of size $22n^2$ integers in local memory, which means that there is an overhead of $O(n^2 \log n)$ bits of local memory on every processor.

3 A Memory-Efficient Algorithm

3.1 Intuition and Basic Idea

As discussed in the previous section, a processor b can reuse a private value v_k only if it determines that v_k is not in use by any processor in the system. The Traceable Use Abstraction used in DW enables b to track down any of its private values v_k which are in use by another processor d, even if v_k propagated through many levels of indirection via other processors to d. Hence DW allows arbitrary levels of indirection of propagation for v_k and if v_k is detected to be in use, it will not be recycled. But as noted in [8], the propagation of private values is restricted to only one level of indirection in BCTS and not to arbitrary levels. Hence, the complete power of Traceable Use is redundant for BCTS. For example, in the scenario described in Section 2, for b to recycle v_k, it is enough to check for the following – v_k does not appear currently in any existing label, v_k will not appear in a label that some processor might write in the future, v_k does not appear in a label that is returned by any concurrent or future scan of b's label. It is not necessary for b to keep track of more than one level of indirection of propagation of its private values.

We now make an observation. In Figure 2, both the **Label()** and **Scan()** procedures collect labels from other processors. While **Scan()** collects whole labels, **Label()** collects only the private value of the processors. However, both operations employ a common **traceable-collect** routine for collecting values. Intuitively, from the point of view of a processor b, while it is necessary for b to keep track of its private values that other labelers consumed from it (which would go towards making up new labels), it is not necessary for b to keep track of private values that scanners consumed from it (as long as the scanners are able to determine, in some other way, the same order imposed on the labels by

Label() procedure, code for processor b
1. For each $c \neq b$, val_c = **traceable-collect**(X_c) // collect private values
2. Read $localOrder = Ord_{bb}$. // copy b's order into local variable $localOrder$
   $y[] = Scan{-}Aside_{b*}[b] \bigcup Aside_{b*} \bigcup Copy_{b*} \bigcup X_b[b]$
   //garbage collect private values
   Choose v_b = smallest element in $localOrder$ not in $y[]$
   Reorder elements in $localOrder$ with v_b as the largest element
3. $new{-}l = (val_1, val_2, \ldots, v_b, \ldots, val_n)$ // this is new label of b
4. **traceable-write**(X_b, $new{-}l$) // write new label into X_b

Fig. 3. Label() procedure.

traceable-collect(X) of RWRWR, code for processor b // similar to that of [5]
Perform handshake
1. For each c, Collect p_{cb}
2. For each c, Write $q_{bc} = p_{cb}$

Remainder of the first Read-Write-Read
3. For each c, Collect $X_c[c]$, tog_c
4. For each c
   Write $Copy_{cb} = X_c[c]$
5. For each c, Collect $X_c[c]$, p_{cb}, tog_c
6. For each c, if $q_{bc} = p_{cb}$ and tog_c is unchanged since line 3
   then $Y_c = X_c[c]$
   else $Y_c = null$

Remainder of the second Read-Write-Read
7. For each c for which $Y_c = null$
   Write $Copy_{cb} = X_c[c]$
8. For each c, Collect $X_c[c]$, p_{cb}, tog_c
9. For each c for which $Y_c = null$
   if $q_{bc} = p_{cb}$ and tog_c is unchanged since line 5
   then $Y_c = X_c[c]$

10. For each c such that $Y_c = null$
    Read $Y_c = Aside_{cb}$
11. Return $(Y_1, Y_2, \ldots, Y_n)$

Fig. 4. traceable-collect RWRWR procedure.

the labelers). This is because *in the **Label()** operation, the collected values go back into the system as the new label of a processor but in **Scan()**, the collected labels do not go back.* While the **Label()** operation introduces a new label into the system and hence changes the system, the **Scan()** operation only makes an inference about the order of labeling events and hence does not change the system. Therefore, the approach taken in [5,8], of using the same **traceable-collect** routine for tracking down private values in both **Label()** and **Scan()** operations, does not seem natural.

We propose a new, modified design. In our algorithm, a processor does not keep track of labels that scanners consumed from it. Rather, the scanner notes

traceable-write(X_b, new-X) of RWRWR, code for processor b
1. For all c, Read q_{cb}
   // read each processor's b-th q to check for overlapping Label operation
2. For all $c \neq b$, Read $qtag_b[c] = PTAG_c[b]$
   // read each processor's b-th component of $PTAG$ to check for overlapping Scan
3. For all c, if $p_{bc} = q_{cb}$ // if labeling operation of processor c overlaps, do as follows
   Write $Aside_{bc} = X_b[b]$ // set aside private value for overlapping labeler c
4. For all $c \neq b$, if $qtag_b[c] \neq QTAG_b[c]$
   // if Scan() operation of processor c overlaps, do as follows
   Write $Scan{-}Aside_{bc} = X_b$ //set aside label for overlapping scanner c
   Write $Ord_{bc} = Ord_{bb}$ // set aside order for overlapping scanner c
5. Atomically write to r_b: // update all fields of r_b atomically
   X_b = new-X
   $tog_b = \mathbf{not}(tog_b)$
   for all c, $p_{bc} = \mathbf{not}(q_{cb})$
   $Ord_{bb} = localOrder$
   $QTAG_b = qtag_b$

Fig. 5. traceable-write procedure.

Scan() procedure, code for processor b // steps 1-3 collect Time-Lapse snapshot [4]
1. **Produce-tag**()
2. For all c, atomically Read from r_c: // copy X_c, $QTAG_c[b]$, Ord_{cc} into local var
   $value_b[c] = X_c$
   $qtag_b[c] = QTAG_c[b]$
   $order_b[c] = Ord_{cc}$
3. For all c
   If $qtag_b[c] = ptag_b[c]$
   // if some processors c have updated X_c, read values set aside during traceable-write
   Read $value_b[c] = Scan{-}Aside_{cb}$
   Read $order_b[c] = Ord_{cb}$
4. The labels and respective orders are $value_b$, $order_b$. // determine real-time order
   To decide the order between the labels $value_b[i][1 \ldots\ldots n]$ and $value_b[j][1 \ldots\ldots n]$:
   For $i = 1$ to n
   For $j = 1$ to n
   Let k be the most significant index in which
   $value_b[i][1 \ldots k \ldots n]$ differs from $value_b[j][1 \ldots k \ldots n]$
   Determine the relative order using $order_b[k]$

Fig. 6. Scan() procedure.

down from each processor, at the same time as it collects the labels, the ordering information necessary for it to determine the order. Each processor uses two pools – a private value pool and a tag pool. Private values are used by labelers for making up labels. Tags are used by scanners to collect a snapshot of labels in the system. As in DW, **Label()** invokes **traceable-collect**, but in **Scan()**, a different approach is used. The scanner collects two things from each processor – its label and the ordering information of its private value pool. To get a

Produce-tag() procedure, code for processor b // produces new tag for every proc
1. Read $ptag_b = PTAG_b$
2. For all $c \neq b$, $x[c] =$ **garbage collect**($PTAG_b[c]$, $QTAG_c[b]$) //garbage collect tags
3. For all $c \neq b$, choose from Tag_{bc}, the local pool of tags for c, a tag $ptag_b[c] \notin x[c]$
4. Write $PTAG_b = ptag_b$ // this is new set of tags produced for other processor

Fig. 7. Produce-tag() procedure.

consistent view of the system, the scanner captures a snapshot, i.e., it collects labels and orders as they existed at a single point in time during the interval of the **Scan()**'s execution. The ordering information collected in the snapshot is used to determine the relative order among the labels. Consider the scenario described in Section 2. In the DW algorithm, all the labels collected by c during **Scan()** – totally $O(n^2)$ private values, $O(n)$ values for each of the n processors – are recorded in shared variables. Labeler b cannot recycle any of these private values since c may refer to those values again when it wakes up. In fact, b can reuse these values only after c performs a subsequent **Scan()**, writing new values in shared memory, in place of the earlier values. Hence, every processor needs to have a minimum pool size of $O(n^2)$ private values for the $O(n)$ scanners. In our algorithm, the scanner collects the labels and corresponding ordering information as they existed at a single point in time, without recording the private values in shared memory. Only the labeler, when it updates, sets aside values in shared memory, for those processors from which it noticed a concurrent **Scan()**. Hence, every processor needs to maintain a pool of only $O(n)$ private values. The scanner can determine the order among the labels by referring to the ordering information it has already noted down from each processor. Conceptually, the algorithm invokes **traceable-collect** from **Label()** and the Time-Lapse snapshot [4] from **Scan()**. This approach is comparable to [7,4] where atomic snapshot and Time-Lapse snapshot respectively are invoked from **Scan()**. While the precedence graph forms the backbone recycling mechanism in [7,4], our algorithm uses the Traceable Use mechanism for recycling private values at the back-end and is hence simpler.

Figure 1 illustrates the main differences between our construction and the Dwork-Waarts construction [5] at the implementation level. X_b is the label of processor b. The boolean p_{b*}, q_{b*} and tog_b registers implement the RWRWR handshake mechanism to coordinate between the **traceable-collect** of one labeler and the **traceable-write** of another labeler. $Aside_{bi}$ is written by b in **traceable-write** if b notices a concurrent **traceable-collect** operation by processor i. The boolean $PTAG_b$ and $QTAG_b$ registers enable b to detect scans from other processors, concurrent with its Label's **traceable-write**. $Scan{-}Aside_{bj}$ and Ord_{bj} are written by b in **traceable-write** if b notices a concurrent scan from processor j. The $Copy_{bi}$ registers, written by the corresponding processor i, enable i to make it known to b which of b's private values it read from label X_b in **traceable-collect**. Ord_{bb}, which is a field of r_b, is updated atomically with X_b by b. The local pool of tags used by b (tags Tag_{b1}, ..., Tag_{bn}) is a collection of $(n-1)$ pools, one pool of three tags for every processor. Since a

scanner b needs to produce a new tag for every processor i, it needs to choose a tag other than the ones which i and b are currently holding and therefore three tags suffice for every processor. Hence the size of the pool of tags is bounded by $3n-3$. From Figure 1, it should be clear that private values of processor b that may be in use at any time are recorded in registers $Scan{-}Aside_{b*}[b]$, $Aside_{b*}$, $Copy_{b*}$ and $X_b[b]$. Hence the private pool size is bounded by $3(n-1)+1+1 = 3n-1$. It is also clear from the figure that since there is only one long SWMR atomic register of length $O(n \log n)$ bits and the remaining long registers r_{b*} are all SWSR regular, the total shared space associated with b is $O(n^2 \log n)$ SWSR safe bits. Hence the total shared space of the system is bound by $O(n^3 \log n)$ SWSR safe bits. The size of the two pools being $O(n)$ integers, the local memory is bounded by $O(n \log n)$ bits.

3.2 Algorithm Description

The pseudo-code of the proposed algorithm is given in Figure 3 to Figure 7. Local variables such as $localorder$, $y[]$, v_b, $new{-}l$, $value_b$, $qtag_b$, $ptag_b$, $order_b[]$, Y_c, etc. used in the routines are assumed to be persistent.

In **Label()**, line 1 invokes the **traceable-collect** routine and collects private values of other processors. Line 2 corresponds to the **garbage collect** routine of [5]. In our algorithm, garbage collection is embedded in **Label()**. Private values of b which could be in use in the system, are collected in $y[]$ by scanning the $Scan{-}Aside$, $Aside$, $Copy$ & X_b registers. After choosing a new private value $v_b \in y[]$ from $localOrder$, b writes the new order in $localOrder$. The new label is written in X_b during **traceable-write** in line 4.

The **Scan()** operation collects a snapshot of $label_c$ and $order_c$ from all c, as they existed at a single point in time. In line 1, b invokes **Produce-tag()** to choose a new tag for each processor from its local pool of tags. Tags are used by a scanner i to determine if a processor d updated itself after i began its **Scan()**. If so, i discards the label and order collected from d in favor of the $Scan{-}Aside$ & Ord_{db} values, which d set aside for it. Line 2 does an atomic collect of the label and order from every processor. Updates that may have occurred during the interval of **Scan()** by some processors are determined by checking $qtag$ in line 3; for each such updater, the values set aside for it by the updater are taken. The total order on the collected labels consistent with the real-time order of their corresponding labeling operations is determined in line 4.

The **traceable-collect** routine invoked by **Label**() is identical to the one in [5]. Over lines (1,3,5), the **traceable-write** sets aside in $Aside$ its own component of label if **traceable-collect** of another labeler is detected to overlap using the RWRWR mechanism. Over lines (2,4,5), the **traceable-write** sets aside the label with the associated order in $Scan{-}Aside$ and Ord registers if **Scan()** of a scanner is detected to overlap using the $PTAG$ and $QTAG$ registers. The **Produce-tag**() routine used by **Scan**() produces a new set of tags for every other processor. Line 2 performs garbage collection to determine a new, unused tag that the scanner can produce again. The pools of tags used in our construction correspond to the pools of colors used in [4].

4 Remarks

We presented a simple, linear-time algorithm for constructing a Bounded Concurrent Timestamp System (BCTS) by giving new, modified designs for the **Label**() and **Scan**() operations. The time complexity of our algorithm matches the best know BCTS algorithms [4,5,8] while the total shared memory required in terms of the fundamental building blocks (SRSW safe registers) is less than that of [5] by two orders of magnitude. No expensive garbage collection is necessary. The pool of private values used by each processor is smaller than [5,8] so that the local memory overhead on each processor is correspondingly smaller. A full and formal version of the paper, including the proof of correctness of the construction, is given in [15]. The main drawback of our construction, and also of [5,8], is that we use long registers of size $O(n \log n)$ bits.

References

1. K. Abrahamson, On achieving consensus using a shared memory, *Proc. 7th ACM Symposium on Principles of Distributed Computing*, 291–302, 1988.
2. H. Attiya and J. Welch, Distributed computing: Fundamentals, simulations and advanced topics, *McGraw-Hill Publishing Company, London, UK*, 1998.
3. D. Dolev and N. Shavit, Bounded concurrent time-stamping, *SIAM Journal of Computing*, 26(2): 418–455, 1997.
4. C. Dwork, M. Herlihy, S. Plotkin and O. Waarts, Time-lapse snapshots, *SIAM Journal of Computing*, 28(5): 1848–1874, 1999.
5. C. Dwork and O. Waarts, Simple and efficient bounded concurrent timestamping and the traceable use abstraction, *Journal of the ACM*, Vol. 46, pp. 633–666, 1999.
6. M. Fischer, N. Lynch, J. Burns and A. Borodin, Distributed Fifo allocation of identical resources using small shared space, *ACM Transactions on Programming Language Systems*, 11(1): 90–114, 1989.
7. R. Gawlick, N. Lynch, N. Shavit, Concurrent timestamping made simple, *Proc. Israeli Symp. on Computing and Systems,* 171–183, LNCS 601, Springer, 1992.
8. S. Haldar and P. Vitanyi, Bounded concurrent timestamp systems using vector clocks, *Journal of the ACM*, Vol. 49, pp. 101–126, 2002.
9. A. Israeli and M. Li, Bounded timestamps, *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pp. 371–382, 1987.
10. A. Israeli and M. Pinhasov, A concurrent timestamp scheme which is linear in time and space, *Proc. Workshop on Distributed Algorithms*, 95–109, LNCS 647, Springer-Verlag, 1992.
11. L. Lamport, A new solution to Dijkstra's concurrent programming problem, *Communications of the ACM*, 17, 1974.
12. M. Li, J. Tromp and P. Vitanyi, How to share concurrent wait-free variables, *Journal of the ACM*, 43(4): 723–746, 1996.
13. G. Peterson, Concurrent reading while writing, *ACM Transactions on Programming Language Systems*, 5(1): 46–55, 1983.
14. T. Petrov, A. Pogosyants, S. Garland, V. Luchangco and N. Lynch, Computer-assisted verification of an algorithm for concurrent timestamps, *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE/PSTV'96*, IFIP Procs., pp. 29–44, 1996.

15. V. Shikaripura, A. Kshemkalyani A simple memory-efficient bounded concurrent timestamping algorithm, Technical Report UIC-CS-02-04, June 2002.
16. P. Vitanyi and B. Awerbuch, Shared register access by asynchronous hardware, *Proc. 27th IEEE Symp. on Foundations of Computer Science*, pp. 233–243, 1986.