Long-lived Adaptive Collect with Applications
(Extended Abstract)

Yehuda Afek
Computer Science Dep.,
Tel-Aviv University,
Israel 69978.
afek@math.tau.ac.il.

Gideon Stupp
Computer Science Dep.,
Tel-Aviv University,
Israel 69978.
afek@math.tau.ac.il.

Dan Touitou
Interdisciplinary Center,
Herzliya,
Israel 46150.
dant@idc.ac.il.

Abstract

A distributed algorithm is adaptive if the worst case step complexity of its operations is bounded by a function of the number of processes that are concurrently active during the operation (rather than a function of N, the total number of processes, which is usually much larger). In this paper we present long-lived and adaptive algorithms for collect in the read/write shared-memory model. Replacing the reads and writes in long-lived shared memory algorithms with our adaptive collect results in many cases in a corresponding long-lived algorithm which is adaptive. Examples of such applications, which are discussed in the paper are atomic-snapshots, and l-exclusion.

Following the long-lived and adaptive collect we present a more pragmatic version of collect, called active set. *This algorithm is slightly weaker than the collect but has several advantages. We employ this algorithm to transform algorithms, such as the Bakery algorithm, into their corresponding adaptive long-lived version, which is more efficient than the version that was obtained with the collect.*

Previously, long-lived and adaptive algorithms in this model were presented only for the renaming problem [2]. A one-shot and adaptive collect algorithm was presented in [10].

1. Introduction

Traditionally, the efficiency of a distributed algorithm is measured as a function of N, the total number of processes that may participate in the algorithm. However, recently it has been observed that the worst case complexity of distributed algorithms could perhaps be made adaptive, that is, bounded by a function of a significantly smaller quantity, the number of concurrently participating, or actually active processes [4]. For example, Lamport's fast mutual exclusion algorithm [18] takes a constant number of steps if one processor runs alone and a linear in N number of steps if two or more processes run concurrently. Other adaptive (sometimes called *fast*) algorithms have been since designed [18, 4, 2, 19, 10, 6, 9, 7].

Long-lived and adaptive algorithms in the read/write shared memory model have been previously presented only for the renaming problem [11, 2, 3]. General methodologies for long-lived adaptive algorithms have been presented only in a model that uses strong synchronization primitives such as read-modify-write, or load-linked and store-conditional [4]. This paper presents building blocks and tools with which many long-lived algorithms in the read/write model can be transformed into their corresponding adaptive versions.

The strongest form of adaptiveness in the read/write shared memory model has been defined and achieved in the recently presented long-lived renaming algorithms [11, 2, 3]. In these specially tailored algorithms the complexity of an operation is a function of the *point contention* of the operation, defined as the maximum number of processes executing concurrently at some point during the operation's interval. However, these renaming algorithms, as they are, turned out to be useless in transforming other long-lived algorithms into their adaptive variants.

Methods for converting algorithms into their adaptive versions have been presented in the read/write model but only for particular *one-shot* algorithms [10, 14]. The basic idea in the transformation suggested in [14] assumes that one-shot single-writer-multi-reader algorithms take the simple form in which each process p_i alternates between writing its own variable C_i and collecting the values of all the variables $C_1, \ldots, C_N$. The transformation then replaces each of the collects with the one-shot adaptive collect procedure of Attiya and Fouren [10]. For many one-shot single-writer-multi-reader algorithms, this transformation removes any dependence of the step complexity on N, resulting in one-shot adaptive multi-writer-multi-reader algorithms.

The piece whose absence prohibits the application of a similar transformation for *long-lived* algorithms in the read/write model, is a long-lived adaptive collect algorithm. In this paper we provide this piece by presenting such an algorithm and demonstrating its applications. First we define two variants of adaptive and long-lived collect, the standard *collect* and *active-set*. Given these two variants the contributions of the paper are as follows:

Basic building blocks:

1. A long-lived and adaptive to point-contention implementation of the standard collect is given and proved.
2. A long-lived and adaptive implementation of the active-set is presented and proved. Although this algorithm satisfies slightly different properties then the standard adaptive collect it has several advantages as discussed below.

Applications of the building blocks:

1. The long-lived and adaptive collect is employed to transform the snapshot algorithm presented in [1] and the l-exclusion algorithm presented in [5]. The resulting algorithms are long-lived and adaptive to the interval contention (a weaker form of adaptiveness, defined below).
2. We specially tailor the transformed atomic-snapshot algorithm to get adaptiveness to the point-contention.
3. The application of the long-lived adaptive active-set algorithm to transform the Bakery mutual exclusion algorithm into a corresponding long-lived and adaptive algorithm is presented. We also discuss similar applications to the l-exclusion, and the l-assignment algorithm of Burns and Peterson [12].

The first building block, which is simply called *collect*, is non-sequentially specified just like the traditional collect: There are N single-writer-multi-reader registers $C_1, \ldots, C_N$, one for each process. To write in the collect process p_i writes its value in C_i. To perform collect a concurrent process reads the N registers, one at a time in an arbitrary order, and returns the vector of values read. Since our implementation of the above specification is adaptive to the (point) contention, it uses multi-writer-multi-reader registers. Moreover, because each operation returns a vector of size N there can be no *adaptive* collect that uses registers of size smaller than N (consider the case that one process runs alone and has to return a vector of N values in $O(1)$ primitive operations). Therefore, our implementation uses registers that can hold N values (as is also the case with the atomic snapshot algorithm of [1]). Furthermore, in our collect algorithm, only in terms of shared memory operations the complexity is adaptive. The number of operations each process performs on its *local* memory is linear in N (due to local operations on the size N registers). However, first notice that remote (shared memory) accesses are considerably more expensive than local memory accesses, secondly, in part we overcome this problem as discussed below.

In an attempt to overcome the two drawbacks above namely, size of registers and the complexity in terms of the number of local operations, we notice that there are algorithms (e.g., Bakery algorithm) in which not all the values returned in a collect operation are important. Rather, only the values of currently participating processes are relevant. For such cases we can redefine a variant of collect, the *active set*, which can be implemented in shared memory and local memory adaptive step complexities. Furthermore, we are able to implement the active set with standard size registers ($O(\log N)$ bits each).

In the active set algorithm we deal with a more basic problem of collecting the processes for which an active flag is set. In this problem there are N single-writer-multi-reader boolean registers, $F_1, \ldots, F_N$, one for each process. Three concurrent operations are supported by the *active set*: join, leave and get_set. In join process p_i writes true in its flag F_i. In leave it writes false to its flag F_i. In a get_set operation a process returns the set of processes that are currently active. That is, the get_set must return any process whose flag is true from before the get_set starts until after it finishes, and it must not return a process whose flag was false during the entire same period. It may or may not return any other process. Using the active-set we implement a weak-collect by first performing get_set and then reading the private single-writer-multi-reader registers of these active processes.

To get some sense of the difficulties in designing long-lived and adaptive collect we briefly review here the one-shot version. The one-shot collect algorithm of Attiya and Fouren [10] uses a large binary tree with a splitter[20] object in each node. Abstractly, the splitter object is a shared device that returns a special value to a process that accesses the splitter alone and before any other process has accessed the splitter, such a process is said to have won or captured the splitter. All the processes that access the splitter concurrently with others or after it has been already touched, are split into two sets such that not all the processes accessing the splitter are placed in the same set. Notice that it is possible for no process to return the special value when accessing a splitter (i.e., it is possible that no process wins a splitter). To perform an update in the collect object a process starts at the root of the binary tree and traverses down until it wins a splitter, marking all the splitters it has touched as it goes down. Then, the process writes its value in a register associated with the splitter it has captured. To perform a collect a processor performs a DFS traversal of the marked nodes of the tree collecting all the values it observes during the traversal. The difficulties in making this algorithm long-lived, are first, how to un-mark nodes when processes finish their update without the concurrently traversing (collecting) processes being confused, and second, where would the values written by processes that have left be found. That is, a process doing collect when the contention is very low is not allowed to perform a DFS to reach deep enough locations in the tree. One of the major achievements of this paper is an algorithm that overcomes these difficulties.

Notice that in our mechanical transformations, even though we use a long-lived and adaptive to point-contention collect, the resulting algorithms are not necessarily adaptive to the point-contention. Sometimes the resulting algorithms are adaptive to the interval-contention. We speculate that in many cases there is no automatic transformation that would result in a point-contention variant of an algorithm, while such a transformation into an interval-contention variant exists.

2. Preliminaries

We assume a standard asynchronous shared-memory model of computation, following, e.g., [15]. A system consists of N processes, $p_1, \ldots, p_N$, communicating by reading and writing to shared registers; we assume that each process can read from and write to any register (*multi-writer multi-reader* registers).

The following is a non-sequential specification of long-lived collect: There are N single-writer-multi-reader registers $C_1, \ldots, C_N$, one for each process. To write process p_i simply writes the value in C_i. To perform collect a concurrent process reads the N registers in an arbitrary order and returns the vector of values read.

Consider some execution α of a long-lived algorithm A; below, α' is some finite prefix of α.

Process p_i is *participating* at the end of α', if α' includes an invocation of some operation of A by process p_i without the matching response. The *active processes* at the end of α', denoted Cont(α'), is the set of processes participating at the end of α'. Given a subsequence β of α, let $\alpha'\beta$ be the shortest prefix of α that contains β, we define the *interval contention* of β and the *point contention* of β, denoted IntCont(β) and PntCont(β) respectively, as follows:

$$\text{IntCont}(\beta) = \Big|\bigcup_{\alpha'\beta' \text{ prefix of } \alpha'\beta} \text{Cont}(\alpha'\beta')\Big|$$

$$\text{PntCont}(\beta) = \max_{\alpha'\beta' \text{ prefix of } \alpha'\beta} |\text{Cont}(\alpha'\beta')|$$

Intuitively, the interval contention of a subsequence β is the number of different processes that were active, (i.e., participating) during β while the point contention is the maximum number of process active at any point of time during β. Clearly, for any subsequence β, PntCont$(\beta) \leq$ IntCont(β).

Given an operation *op*, we define the *execution interval* of *op*, denoted $\beta(op)$ as the subsequence of α starting at the invocation of *op* and ending at the completion of *op*. The *interval contention* and *point contention* of an operation op are defined as IntCont$(\beta(op))$ and PntCont$(\beta(op))$ respectively. In the rest of the paper, k denotes the PntCont$(\beta(op))$ of some operation op, unless it is specifically said to represent IntCont$(\beta(op))$.

The step complexity of an algorithm is *adaptive to interval contention* if there is a bounded function S, such that the number of steps performed by any process p_i in any execution interval of an operation op_i of A is at most S$($IntCont$(\beta(op_i)))$. Similarly, the step complexity of an algorithm is *adaptive to point contention* if there is a bounded function S, such that the number of steps performed by any process p_i in any execution interval of an operation op_i of A is at most S$($PntCont$(\beta(op_i)))$.

Clearly, the contention of an execution interval is bounded by n, the total number of processes. Therefore, in an adaptive algorithm, op_i terminates within a bounded number of steps of p_i, regardless of the behavior of other processes. Thus, an algorithm with adaptive step complexity is necessarily *wait-free*.

3. Adaptive Collect

3.1. Algorithm overview

The algorithm uses a $2N^2$ entries array in which processes store their values. To perform an update a process temporarily captures an entry in the array, in exclusion, as close to the beginning of the array as possible. It records the value it writes in the multi-writer-multi-reader register associated with this entry of the array and releases the entry. At this point the process has successfully stored its value in the system, later updates will not overwrite it. However, a subsequent collect will have to scan the array to find the new value. If this collect happens with low contention (e.g., in solo) such a scan would make it not adaptive. Therefore, before finishing the update the process has to bubble up its value to the beginning of the array. Future collects then start at the beginning of the array, and in the absence of contention find the necessary values at the beginning of the array. In bubbling up, a process iterates on the entries of the array from the entry it has captured up to the top (beginning) of the array. In each such entry it recursively per-

Algorithm 1 Code for adaptive collect for process p.

```
    Type:
        $pid$ = process id, $1 \ldots N$ and $\perp$;
        $item$ = $\langle pid, val, timestamp \rangle$;
        $itemSet$ = Set of $item$;
    Shared:
        $A[1..2N^2]$: atomic MRMW registers of type
                                                  $itemSet$;
        $last[1..2N^2]$: atomic MRMW registers of type $pid$;
        $C[1..N][1..2N^2]$: atomic SWMR registers of type
            $itemSet$ each initialized to $\emptyset$;
    Local registers global to the program:
        $index$;
        $timestamp$;

    procedure update($val$)
1       refresh($\{\langle p, val, timestamp \rangle\}$);
2       $timestamp$:=$timestamp$+1;

    function collect() returns $ItemSet$
3       $s$:=gather(1);
4       refresh($s$);
5       return $s$;

    function gather($t$) returns $ItemSet$
6       $q$:=$last[t]$;
7       $tmp$:=$C[q][t]$;
8       if ( $tmp = \perp$) then
9           $tmp$:=merge(gather($t$+1),$A[t]$);
10      return $tmp$;

    procedure refresh($itemSet\ S$)
11      $index$ := $2k^2$-rename($p$);                // from [3]
12      $A[index]$:= merge($A[index]$, $S$);
13      $2k^2$-release-name($index$);
14      for ( $t = index$ down to 1) do
15          $C[p][t]$:=$\perp$;
16          $last[t]$:=$p$;
17          $C[p][t]$:=gather($t$);
        od;

    function merge($ItemSet\ S$, $ItemSet\ T$)
18      The function merges the two sets,
        leaving for each process only the
        most recent entry according to the timestamps.
```

forms a sub-collect of the values recorded in the part of the array which is below this entry, and records this collect in a *private* single-writer-multi-reader register that is associated with this entry.

To perform a collect a process starts at the top (beginning) of the array collecting values as it goes down. A key point is that the number of entries the collecting process scans depends on the point-contention it encounters. In the absence of contention it finds the necessary values at the first entry of the array. As described, the collect is regular, i.e., the first of two sequential collects that overlap a write of value "new" may return "new" while the later might return "old". To prevent this we require each collect to write the values it has read (perform an update()), thus ensuring that any later collect returns the same or later values.

To capture an entry in the array in exclusion we use the long-lived and adaptive $2k^2$-renaming algorithm of [2, 3] whose step complexity is $O(k^3)$ adaptive to point-contention (Line 11 in Algorithm 1). After acquiring a name $index$ the process adds its value to a register associated with entry $index$ in the array (Line 12 in Algorithm 1). It then releases the acquired name (Line 13) and starts the bubbling up process.

In the bubbling up process the updater goes through the entries of the array from the index it had captured to the top. In each entry it performs a sub-collect of the bottom part of the array (Function gather) and records it with this entry (Line 17). To do that we associate with each entry a pointer, called $last$, that points to the single-writer-multi-reader register of the last process that has recorded a sub-collect with this entry. The process performs the bubbling up through an entry in a particular order: First it writes $\perp$ in its single-writer-multi-reader register associated with this entry (called $C[p][i]$ for process p in entry i), secondly it writes its name into $last$, and finally performs the sub-collect and records its result in $C[p][i]$ (Lines 15 to 17).

This particular order of recording collected information in each entry guarantees the following property: If a process q read $last[i] = p$ and subsequently it reads $C[p][i] = vector$ then any update that has recorded information below entry i in the array and has terminated before q read $last[i]$ is included in the $vector$. Furthermore, if q observes $C[p][i] = \perp$ then it knows process p is concurrent with its operation and by the adaptiveness it may now perform a few more operations, in particular recursively gathering the information at entry $i + 1$ (Line 9 in the code).

Since the name process p gets from the renaming algorithm is at most $2k^2$, and the gather procedure takes at most $O(k^2)$ the total step complexity of the update() and of collect is at most $O(k^4)$.

To be able to distinguish new values from older values we add to every value written by every process a sequence number. When process p adds its information to entry $A[i]$

it erases the last value it wrote there (if any). When two sets of values are merged (recursive collect operation and content of $A[]$ of some entry, Lines 9 and 12), the value with the larger sequence number is taken for each process.

The proof of correctness and that the complexity of either the collect or update is $O(k^4)$ are given in Sections 3.2 and 3.3, where it is proved that the adaptive collect satisfies the following two properties (these are the properties used to prove the correctness of the algorithms that use the adaptive collect):

1. For each process the collect() operation returns a value that was written either by the last update operation to end before the collect() has started or by a later update operation which is concurrent with the collect().

2. In two sequential collects, $C1$ and $C2$, such that $C2$ starts after $C1$ ends, for each process j, $C2[j]$ is a value that was writen not before value $C1[j]$ was.

3.2. Correctness of Adaptive Collect (sketch of proof)

Any execution r of procedure refresh has a set s_r of triplets that is passed to the procedure as a parameter. Given a set R of procedure refresh executions, we say that some triplet $t = \langle p, val, ts\rangle$ is in R, denoted $t \in R$ if there is some $r \in R$ such that $\langle p, val, ts\rangle \in s_r$. Given an execution g (c) of gather (collect), we denote by s_g (s_c) the set of triplets returned by g (c).

The following lemma states that any value returned by gather was once added by some refresh call. This lemma is easily proved by induction on the prefixes of the adaptive collect execution.

Lemma 3.1 *Let e be an execution of the adaptive collect algorithm, then for any prefix pre of e if R is the set of* refresh *operations that started during pre, then any register in C and in A contains triplets that are in R. Moreover, if g is an execution of* gather *completely included in pre then any triplet returned by g is in R.*

Given two triplets $t = \langle p, val, ts\rangle$ and $t' = \langle p, val', ts'\rangle$ we say that t' is *more recent* than t (denoted $t' \geq t$) if $ts' \geq ts$. Given a set R of refresh executions and a process p we define the *most recent* triplet of p in R, denoted $Recent_{p,R}$ to be $\langle p, val, ts\rangle \in R$, s.t., for every $t = \langle p, *, *\rangle \in R$, $Recent_{(p,R)} \geq t$. If R does not contain any triplet $\langle p, val, ts\rangle$ then $Recent_{p,R}$ is undefined.

Given an execution r of refresh, we define the *entry slot* of r to be the index returned by the $2k^2$-rename algorithm in Line 11. Given an execution r of refresh with entry slot x, we say that r *crossed* slot $x' \leq x$ if the process executing r has already changed $last[x']$ to its id (Line 6).

Lemma 3.2 *Let e be an execution of the adaptive collect algorithm, and let g be an execution of* gather(x) *contained in e. Assume that R_1 is the set of all* refresh *operations that started before the end of* gather(x) *and R_2 is the set of all* refresh *operations that crossed x before the beginning of* gather(x), *then for every process p, if $Recent_{p,R_2}$ is defined then s_g contains a triplet $t = \langle p, val, ts\rangle$ s.t., $t \in R_1$ and $t \geq Recent_{p,R_2}$.*

Proof: We show that the claim holds for every prefix of e by induction on the length of the prefixes. Let g be a call to gather(x) and let p be some process and assume that $Recent_{p,R_2}$ is defined. Let r be an execution of refresh such that $Recent_{p,R_2} \in s_r$. According to the algorithm, a process q returns from gather(x) either **(1)** after reading some process id q' in $last$[x] and then reading a non-$\perp$ value c[q'][x]= s (lines 7,8) or **(2)** after merging the result of gather$(x+1)$ with the content of A[x] (Line 9).

Case **1**. Since process q' assigned $\perp$ to c[q'][x] before writing its id into $last$[x] (Line 15) we may deduce that the value read in $C[q']$[x] is the result of a gather(x) call performed by q' after q' wrote its id into $last$[x]. If r crossed x before q' executed gather(x), according to the induction hypothesis the set returned by the execution of gather(x) by q' contains a triplet $t = \langle p, val, ts\rangle$ which is more recent than $Recent_{p,R_2}$. If r crossed x after the beginning of q''s execution of collect(x), the process executing r must have overwritten q' in $last[x]$ with its own id before it was read by q- a contradiction.

Case **2**. If the entry slot of r is greater than x, then according to the induction hypothesis, the call to gather$(x+1)$, which is completely included in the call to gather(x), returns a triplet $t = \langle p, val, ts\rangle$ which is more recent than $Recent_{p,R_2}$. Now, since during the call to merge, only updates with higher timestamps than those returned from gather$(x+1)$ are chosen we are done. Assume that the entry slot of r is x itself, by Line 12 and by Lemma 3.1 when q reads $A[x]$, it must contain a triplet $t = \langle p, val, ts\rangle$ which is more recent than $Recent_{p,R_2}$. ■

The following lemma follows immediately from the fact that the result of a collect is gather(1).

Lemma 3.3 *For every execution c of* collect *and every process p if s_c contains a triplet of the form $t = \langle p, val, ts\rangle$ then t is more recent than all the triplets updated by p before the beginning of c.*

Since every collect performs a refresh(s) where s is the collect result, due to the previous lemma we may state:

Lemma 3.4 *Let c and c' be two executions of* collect *and assume that c' starts completely after the end of c then for every triplet t in $s(c)$, there is a triplet t' in $s(c')$ which is more recent than t.*

3.3. Complexity (Sketch of analysis)

Assume that p is a process executing gather(x), we say that p *skips* over a slot x if it reads $last[x] = p'$ and C[p'][x]$= \perp$ for some process p', and consequently has to perform a recursive call gather($x+1$). In such a case we also say that *p skips over p'.*

By the algorithm, a process skips a slot when it encounters another process p' that traverses this slot during its bubbling up procedure (lines 15 – 17). Since process p' writes a value different than $\perp$, into $C[p'][x]$ before climbing up to slot $x-1$, there is at most one entry $C[p'][*]$ at any point of time which is $= \perp$. Hence, p may skip over p' at most once while p' executes the bubbling up procedure and p a gather procedure.

Lemma 3.5 *The largest slot index that can be reached during an execution of procedure* gather *is $g(k)+k$ where k is the point contention of the* gather *execution interval and g(k) is the name complexity of the renaming algorithm used.*

Proof: Let g be the execution of procedure gather(x) by process p. By the algorithm, p performs recursive calls on larger slots until it reaches slot x' such that, $last[x'] = p'$ and $C[p'][x'] \neq \perp$. We separate the slots skipped by p into two sets: all the slots in which the processes crossed by p started their call to refresh *after* p started the call to gather and all the slots in which the processes crossed by p started their last call to refresh *before* p started the call to gather. If some process p' started a call to refresh after p started its call to gather the point contention it encountered during the renaming algorithm is at most k. Consequently, the entry slot of p' is bounded by $g(k)$ and in such a case p may cross it only while skipping over slots $1 \ldots g(k)$. If some process p' started its last call to refresh before p started its call to gather, it may be crossed by p at most once, while bubbling up during refresh (lines 15– 17). ■

Since any execution of gather goes down at most to the $g(k)+k$ slots, and since $g(k) = 2k^2$ we may conclude:

Theorem 3.6 *The shared memory step complexity of* update *and* collect *is $O(k^4)$*

In Section 8 we show how to reduce the complexity of procedure gather to $O(k)$, which leads to an overall complexity of $O(k^3)$ for the update and collect operations.

4. Transformations With Adaptive Collect

Most long-lived algorithms in the read/write shared memory model can be writen in the natural form in which each process alternates between writing its private single-writer-multi-reader register and reading all the registers. Following [14] we transform any such algorithm by replacing the writes with the updates from Algorithm 1, and the read-all with the collect. The transformed algorithm is in many cases a long-lived and adaptive version of the original algorithm. Examples of such algorithms are the snapshot algorithm from [1] and the l-exclusion algorithm of [5].

Notice that unlike the adaptive collect procedure (Algorithm 1) that is used in the transformation, the resulting algorithms are not necessarily adaptive to the point-contention but rather they might be adaptive to the interval-contention encountered. As in the collect procedure the local step complexity of the transformed algorithms is not adaptive. To overcome these disadvantages we present in the next section a long-lived snapshot algorithm that is adaptive to the point-contention, and in Section 6 we present a long-lived and adaptive active set. Using the active set we can transform the Bakery algorithm, the l-exclusion algorithm [5] and the l-assignment algorithm [12] into corresponding long-lived algorithms that are adaptive also in the number of local operations performed.

5. Point Contention Adaptive snapshot

The snapshot algorithm, like the collect algorithm, supports two operations: an snapshot_update() operation with which any process updates its value and a scan() operation in which any process p collects the values written by all the processes. The difference between a snapshot and a collect is that the scans can be linearized with respect to the update operations.

The snapshot algorithm in [1] is based on the idea that in every update operation by process p, p first performs a scan and then writes the scan result with the new update value. To perform a scan, process p performs two collect operations. If the values of all the processes are the same in the two collects (to make this check robust every new value is tagged with a sequentially increasing label) the collect is a valid snapshot. If not, the process is iterated and another two collects are performed. This procedure continues until p observes some process, q that has changed its value twice. Since q made a scan, $scan_q$, between its updates, $scan_q$ was performed during p's scan, $scan_p$, and is thus a valid returned value for $scan_p$. From this algorithm it follows that a scan operation performs at most $2k$ collect operations, or that its step complexity is at most $O(kN)$, where k here is the interval contention of the operation, i.e., IntCont($\beta(scan_p)$).

As we mentioned before, simple substitution of the collect operation with an adaptive collect creates an adaptive with respect to interval contention algorithm. Consider for example the following scenario: during a snapshot, k processes update their values one after the other with no overlap in their executions. Still the scanning process fails to obtain two consecutive identical collects k times. However, in this scenario the point contention is 2 but the algorithm makes $O(k)$ collect operations.

The key idea in the modified snapshot algorithm, Algorithm 2, is as follows: If $scan_p$ observed q with two different values, v_1 and v_2, and later p observes any other process r reporting a snapshot in which q appears with value v_2, then p may return this other snapshot for its scan operation. Since this scan by r is linearized after process q wrote v_2 it's linearization point is contained in the interval of $scan_p$. Moreover, if no such r is observed but some process s did change its variable, then the update operation of process s was concurrent with the second update of q and hence at the same time concurrent with $scan_p$. Furthermore, s's update is concurrent with any other process that is observed to change its variable.

Algorithm 2 Code for adaptive with respect to point contention snapshot for process p.

```
function scan()
    $moved\_id$:=$\perp$;
    while true do
        $a[1..N]$:=collect();              // from Algorithm 1
        $b[1..N]$:=collect();
        if ($\forall j \in \{1 \ldots N\}$ $a[j].seq = b[j].seq$) then
            return ($\langle b[1].data$, $b[1].seq\rangle$,
                                  $\ldots, \langle b[N].data, b[N].seq\rangle$);
        else-if $moved\_id = \perp$
            then for ($j$=1 to $N$) do          // local access
                if($a[j].seq \neq b[j].seq$) then
                    $moved\_id$:=$j$;
                    $moved\_seq$:=$b[j].seq$;
                    break;
            od;
            else                              // $moved\_id \neq \perp$
                for $j$ = 1 to $N$ do             // local access
                    if ($b[j].view[moved\_id].seq \geq moved\_seq$)
                    then return $b[j].view$;
    od;  od;

procedure snapshot_update($data$)
    $s[1..N]$:=scan;
    $seq$ := $seq$ + 1;
    update($\langle data, seq, s[1..N]\rangle$); // from Algorithm 1
```

5.1. Correctness proof (sketch), and complexity

In order to prove that our implementation is a correct atomic snapshot, we show that our implementation is *linearizable*[17]: for every execution of the snapshot algorithm, there exists a total order among the scan and snapshot_update operations, consistent with the partial order induced by the execution, such that, the sequential history[17] induced by this total order is *correct*. A sequential history is correct if every scan returns for every process p the value written by the last snapshot_update of p that is serialized before the scan operation[1].

Let s and s' be two scan results. We say that s' is *more recent* than s, denoted $s' \geq s$ if for every process p, if s contains a triplet $t = \langle p, val, ts\rangle$ then s' contains a triplet t' such that t' is more recent than t. The next lemma states that every two scan results are comparable:

Lemma 5.1 *For every two scan results, s and s', either $s \geq s'$ or $s' \geq s$.*

Proof: Directly from Lemma 3.4. ∎

Given an execution of the snapshot algorithm we define the total order, $\rightarrow$, among scan and snapshot_update operations in the following way: Let S and S' be two scan executions with results s and s' respectively and let U and U' be two snapshot_update operations performed by p_U and $p_{U'}$ with time-stamps ts_U and $ts_{U'}$ respectively. We say that $S \rightarrow S'$ iff s' is more recent than s, $U \rightarrow S$ iff s contains a triplet $\langle p_U, val, ts\rangle$ such that $ts \geq ts_{U'}$. Finally if there is a scan operation S such that $U \rightarrow S$ and $S \rightarrow U'$, we say that $U \rightarrow U'$. If there is no such scan operation we order U and U' according to their respective ending points. The correctness of the following lemmas follows from lemmas 3.3 and 3.4.

Lemma 5.2 *The $\rightarrow$ total order is consistent with the partial order induced by the execution.*

Lemma 5.3 *The sequential history induced by $\rightarrow$ is correct.*

Now we conclude:

Theorem 5.4 *The snapshot algorithm is a linearizable implementation of the snapshot object.*

5.2. Complexity

Lemma 5.5 *If process p performs the while loop k times during an execution of the scan function* without terminating *then, the point contention of this scan execution is at least k.*

Sketch of proof: Assume that p performed k iterations of the while loop during the execution of scan. For iteration i there is at least one process, denoted p_i, such that during the i-th iteration p has read $a[p_i].seq \neq b[p_i].seq$. Let x_i be the content of $b[p_i]$ during the i-th iteration. Clearly, for every $i = 1 \ldots k$ p_i updated x_i into the collect object after p started the scan execution. According to the algorithm for every $i, j \in \{1 \ldots k\}$, $i \neq j$, $p_i \neq p_j$, otherwise the scan operation would have terminated by the condition in Line 33. Moreover, for every $i = 2 \ldots k$, $x_i.view[p_1].seq < x_1.view[p_1].seq$. Therefore, for every $i = 2 \ldots k$, p_i has started the update operation during which it updated x_i into the collect object *before* p_1 updated x_1 into the collect object and therefore, $p_2 \ldots p_k$ are all active at that point of time. ■

Algorithm 3 Code for active set for process p.

```
   Type:
      $pid$ = process id, $1..N$ and $\perp$;
      $pidSet$ = Set of $pid$ types;
   Shared:
      $A[1..2N^2]$, atomic MRMW registers of type $pid$;
      $last[1..2N^2]$, atomic MRMW registers of type $pid$;
      $C[1..N][1..2N^2]$, atomic SWMR registers of type
         $pidSet$, each initialized to $\emptyset$;
   Local registers global to the program:
      $index$;

   procedure join()
38    $index$ := $2k^2$-rename($p$);
39    $A[index]$:=$\{p\}$;
40    bubble_up();

   procedure leave()
41    $A[index]$:=$\emptyset$;
42    $2k^2$-release-name($index$);
43    bubble_up();

   procedure get_set() returns $pidSet$.
      return gather(1)

   procedure bubble_up()
44    for ( $t = index$ down to 1) do
45       $C[p][t]$:=$\perp$;                    // different than $\emptyset$
46       $last[t]$:=$p$;
47       $C[p][t]$:=gather($t$);
      od;

   function gather($t$) returns $pidSet$.
48    $q$:=$last[t]$;
49    $temp$:=C[q][t];      // try to get the set from the last
                              process that updated this entry.
50    if ( $temp = \perp$) then // some other process is active
                       therefore more operations are allowed.
51       $tmp$:=gather($t$+1)$\cup A[t]$;
52    return $tmp$;
```

Theorem 5.6 *The complexity of a* scan *or* snapshot_update *operation is* $O(k^4)$

6. Active set

Although the adaptive collect presented in Section 3 enables an automatic substitution of a standard collect operation it has two major drawbacks; the size of the registers, which is unavoidable, and the non-adaptive local step complexity. To deal with these drawbacks we suggest the *active set*. A new object which, in many cases, can substitute for collect:

Definition 6.1 *An* active set *object supports the following operations:*

1. join(): *with which a process joins the set.*
2. leave(): *with which a process leaves the set.*
3. get_set(): *which returns the current set of active processes. More formally,* get_set() *must return all the processes that have finished* join() *before* get_set() *has started and did not start* leave() *during* get_set(). *It also must* not *return any process that has finished* leave() *before* get_set() *started and has not started* join() *during* get_set().

Our implementation of an active set is such that the step complexity of the operations depends both on the contention and on the number of processes that are currently in the set. I.e., if there are M processes in the set then get_set() can take $O(M)$ steps, even if all the processes have finished the join() and have not yet started leave(). Therefore, for this problem we redefine (from Section 2): *Process p is participating at the end of α'*, if α' includes an invocation of join by p without the response of the matching leave by p. Fortunately, many non adaptive long-lived algorithms may be transformed into their adaptive versions by substituting the standard collects with a collect that is based on the active set. In these algorithms only the values of processes that are in certain region of the code are relevant (e.g., outside the remainder section, in mutual exclusion algorithms). Typically, in such an algorithm process p starts with a join() and ends with a leave(). Whenever a process needs to collect information from the other processes, it collects it only from the active processes as follows: it first performs a get_set() and then collects the data only from the processes in the set. If, for example, the number of collects in the algorithm is constant, then the new transformed algorithm *is* adaptive with respect to point contention (as originally defined) since process p is actually active between the join() and leave(). For example, see Algorithm 4.

Algorithm 3 implements an active set in $O(k'^4)$, where k' is the point-contention of operation op, according to the new definition of *participating* given above. In the full paper an active set algorithm similar to Algorithm 5 with $O(k'^3)$ step complexity is presented.

During join() process p uses renaming() to acquire a unique entry. This entry is assigned to p until it performs leave(). After getting an entry, p writes its name in that entry and then propagates its name and the names of all the processes it sees up to the beginning of the array (similar to the adaptive collect algorithm). In the leave() procedure process p erases its name from the entry, releases the name it got, and again propagates this information up the array. The get_set() operation invokes the gather operation which works like in the adaptive collect algorithm. Either the last process to propagate up from entry i has finished, in which case a correct return value can be read from its dedicated register, or it is active and so the gather() procedure can continue to the next entry. Notice that here, there is only one value in each entry and merging the $O(k'^2)$ entries takes $O(k')$ time, even in local steps. In Section 6.1 we show how the algorithm is implemented using registers of size $O(\log N)$ bits.

The proof of correctness is very similar to the proof of the adaptive collect (Section 3.2) and is given in the full paper.

6.1. Adaptive Active Set: Reducing the Register Size

As the reader may notice, the only data structure that uses large registers is C, where processes store the result of a completed gather. One possible way to avoid using large registers is to replace every entry in C with a linked list of process ids. In such case, a situation in which some process p executing gather(x) reads $C[p'][x].list$ while p' is updating $C[p'][x].list$ may occur. To overcome this problem we borrow a technique from [16] and we wrap every list in C with two counters $start$ and $done$. Whenever p' decides to alter $C[p'][x].list$, it first increments $C[p'][x].start$ changes $C[p'][x].list$ and then increments $C[p'][x].done$. A process p reading $C[p'][x].list$, first reads $C[p'][x].done$, collects the content of $C[p'][x].list$ and then reads $C[p'][x].start$. If the values read in $C[p'][x].done$ and $C[p'][x].start$ differ, that means that p' was concurrently updating the list, and so p cannot return the list but rather has to merge $A[x]$ with a the result of a recursive call to gather($x+1$). Notice, that one can bound the number of values used in $C[p'][x].start$ and $C[p'][x].done$ using techniques from [8].

7. Local Adaptive Mutual Exclusion

Here (Algorithm 4) we demonstrate the usefulness of the long-lived and adaptive active set in transforming the Bakery mutual exclusion algorithm into an adaptive mutual exclusion algorithm. Since in mutual exclusion a hungry process must busy wait while other processes are in the critical section the step complexity is redefined in this case as follows: each spin-lock on a variable while it does not change is counted as one operation (in other words we count only the number of remote, un-cashed, accesses). To the best of our knowledge no adaptive mutual exclusion algorithm was previously presented (Choy and Singh have presented a mutual exlusion algorithm whose system response time is adaptive, but not its step complexity [13]). Algorithm 4 presents the code of the transformed adaptive Bakery mutual exclusion algorithm. The correctness proof is given in Section 7.1.

Algorithm 4 Code for adaptive Bakery for process p.

```
Initially $Number[i]$=0 and
$Choosing[i]$=false, for $i$, $1 \le i \le N$.

(entry)
53    join();
54    $Choosing[p]$:=**true**;
55    $S$:=get_set();
56    $Number[p]$:= $\max_{j \in S}(Number[j])$+1;
57    $Choosing[p]$:=**false**;
58    $S$:=get_set();
59    once for every $j \in S$ do
60        wait until $Choosing[j]$=**false**;
61        wait until $Number[j]$=0 or
                      $(Number[j], j) > (Number[p], p)$;
(Critical Section)
(Exit):
62    $Number[p]$:=0;
63    leave();
(Remainder)
```

7.1. Correctness of the adaptive Bakery Algorithm

Lemma 7.1 *The adaptive bakery algorithm provides mutual exclusion.*

Proof: Assume by way of contradiction that two processes, p and q, are in the critical section at the same time. According to the algorithm, $Number[p]$ and $Number[q]$ remain unchanged from the moment p and q finish Line 56 until they leave the critical section. Let $label_p$ and $label_q$ be the content of $Number[p]$ and $Number[q]$ respectively during that period of time. Assume w.l.o.g that $(label_p, p) > (label_q, q)$. If the returned set S that p gets from the second execution of get_set at Line 58 does not contain q, then by Property 3 of Definition 6.1, q does not complete the join execution at Line 53 before p starts the get_set at Line 58. Therefore q executes lines 55– 56 after p executes Line 56 and consequently reads $label_p$ in $Number[p]$ while executing Line 56. In such case $(label_p, p) < (label_q, q)$ which is a contradiction to the assumption that $(label_p, p) > (label_q, q)$. Therefore $q \in S$. Assume that p completes the execution of Line 61 with $j = q$, by reading 0 in $Number[q]$. In order to pass the wait statement in Line 60 with $j = q$, p had to read **false** in $Choosing[q]$. This may happen either if (**1**) before q executes Line 54 or (**2**) after q executes Line 60. In case (**1**), q sees $label_p$ while executing Line 56 and $(label_p, p) < (label_q, q)$. In case (**2**) p cannot read 0 in $Number[q]$. Therefore p must read $label_q$ in $Number[q]$ at least until q leaves the critical section and for that reason p cannot enter the critical section at least until q leaves it. ∎

The proof that the algorithm is lockout-free is the same as in the original algorithm.

8. Adaptive collect, an O(k) gather procedure

The basic idea of the $O(k)$ improvement of function gather, is as follows: if the point contention of a gather execution is k, and process p which executes gather has already crossed ("skipped") over other processes more than k times, there is at least one process crossed by p that already completed its own execution of gather. Therefore, every process executing gather mantains a set of processes it has crossed since the beginning of the gather execution. From time to time, process p checks if one of the crossed processes has completed, and if so, p "returns back" to the entry in which the crossed process was crossed. We add for every process p and every entry i a SWMR register $Clast[p][i]$ that contains the result of the last gather(i) execution p has performed during refresh. $Clast[p][i]$ as oppose to $C[p][i]$ is never set to $\perp$. In addition every process keeps a tag/counter it increments each time it writes a new gather result in C (Line 85). That tag value will be written together with the process id, in $last$ before the gather execution (Line 86), and together with the result of the gather in $Clast$ (Line 88). In order to ensure a $O(k)$ collect complexity each time it crosses some other process p', process p that executes gather adds to a special set, $crossed$, in its private memory p''s id together with the associated tag value and the slot number in which p and p' have met (Line 74). Every time p executing gather accesses a new slot in A it search its set in order to find if one of the processes in the set has completed the gather execution it was doing while p crossed it. This is done, by comparing the tag associated to p''s id p has read in $last[i]$ while crossing p' at entry i, with the tag associated with the gather result now mantained in

Algorithm 5 Code for $O(k)$ gather

```
   Updated Data Structures
      $last[1..2N^2]$: atomic MRMW registers
         of type $pid \times$ Integer;
   Additional data structures
   Shared:
      $Clast[1...N][1...2N^2]$, $2N^3$ atomic SWMR
      registers of type $Set \times$ Integer for each process,
   Local registers global to the program:
      $TS$ an integer initialized to 0;
         each initialized to $\emptyset$;
   function gather($t$) returns $ItemSet$.
      $crossed := \emptyset; t' := t$;
      $pwr := 1; ctr := 0$;
64    while (true) do
65       $\langle pr, ts\rangle := last[t']$;
66       $tmp := C[pr][t']$;
67       if $tmp \neq \perp$ then
68          break;
69       if $ctr = pwr$ then
70          $pwr := 2 * pwr$;
71          if $crossed$ contains $\langle q, t'', ts'\rangle$ for some
            process $q$ entry $t''$ and timestamp $ts'$
            s.t., the timestamp in Clast[q][t"] $\geq ts'$ then
72             $t' := t''$;
73             $\langle tmp, ts\rangle := Clast[q][t'']$;
               break;
74       $crossed := crossed \cup \langle pr, t', ts\rangle$;
75       $t' := t' + 1$;
76       $ctr := ctr + 1$;
      od;
77    while ($t' \geq t$) do
78       $t' := t' - 1$;
79       $tmp :=$ merge(tmp, $A[t']$);
      od;
      return $tmp$;
   procedure refresh($itemSet\ S$)
80    $index := 2k^2$-rename($p$);
81    $A[index] :=$ merge($A[index]$, $S$);
82    $2k^2$-release-name($index$);
83    for ( $t = index$ down to 1) do// Bubble up process
84       $C[p][t] := \perp$;                    // different than $\emptyset$.
85       $TS := TS + 1$;
86       $last[t] := \langle p, TS\rangle$;
87       $c :=$ gather($t$);
88       $Clast[p][t] := \langle c, TS\rangle$;
89       $C[p][t] := c$;
90       $t := t - 1$;
      od;
```

$Clast[p'][i]$ (Line 71). If this is the case p "jumps" back to entry i, uses the content of $Clast[p'][i]$ as its own result of gather on entry i and in fact behaves as if it never crossed over p' (Line 77). In the full paper we show that if the point contention is k, after scanning k entries, a process executing gather finds at least one process that was crossed by p and has terminated its ongoing collect and stored it in $Clast$. Note that if a process would have to scan the set after each entry in A the complexity would be $O(k^2)$. For that reason a process does not scan the set for every entry, but only every 2^j entries for $j = 0, 1 \ldots$ (Line 69). In this way, if the point contention is k, the terminating process is discovered after at most $2k$ slots, but the total work on the set is still $O(k)$. The correctness and complexity of the new algorithm are shown in the full paper.

Theorem 8.1 *There is an implementation of Adaptive collect in which the complexity of* update *and* collect *are* $O(k^3)$

Acknowledgements: We would like to thank Hagit Attiya and Arye Fouren for many helpful discussions.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, pages 91–103, May 1999.

[3] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive Long-Lived Renaming Using Bounded Memory Submitted to DISC99, April 1999. ftp://ftp.math.tau.ac.il/pub/stupp/PAPERS/name99.ps.gz

[4] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, pages 538–547, May 1995.

[5] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Trans. on Programming Languages and Systems*, 16(3):939–953, May 1994.

[6] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, pages 105–112, May 1999.

[7] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proc. 16th Annual ACM Symp. on Principles of Distributed Computing*, pages 111–120, August 1997.

[8] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM, Aug. 1995.

[9] H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th Annual ACM Symp. on Principles of Distributed Computing*, pages 223–232, May 1996. Extended version available as Technion Computer Science Department Technical Report #0931, April 1998.

[10] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, pages 277–286, June 1998. Extended version available as Technion Computer Science Department Technical Report #0931, April 1998.

[11] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 0956, Faculty of Computer Science, Technion, Haifa, 1999. http://www.cs.technion.ac.il/∼hagit/pubs/tr0956.ps.gz.

[12] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, Edmonton, Alberta, Canada, August 1989.

[13] M. Choy and A. K. Singh. Adaptive solutions to the mutual exclusion problem. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 183–194, August 1993.

[14] E. Gafni. Public communication. 17th Annual ACM Symp. on Principles of Distributed Computing, July 1998.

[15] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[16] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, November 1993.

[17] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[18] L. Lamport. A fast mutual exclusion algorithms. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[19] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[20] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995. Also in Proc. 8th Int. Workshop on Distributed Algorithms, September 1994, 141-155.