Help when needed, but no more: Efficient Read/Write Partial Snapshot

Damien Imbs, Michel Raynal

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N^{o} 1907

HELP WHEN NEEDED, BUT NO MORE:
EFFICIENT READ/WRITE PARTIAL SNAPSHOT

DAMIEN IMBS MICHEL RAYNAL

ISSN 1166-8687

IRISA

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Help when needed, but no more: Efficient Read/Write Partial Snapshot

Damien Imbs[*] Michel Raynal[**]

Systèmes communicants
Projet ASAP

Publication interne n ° 1907 — Novembre 2008 — 24 pages

Abstract: An atomic snapshot object is an object that can be concurrently accessed by asynchronous processes prone to crash. It is made of m components (base atomic registers) and is defined by two operations: an update operation that allows a process to atomically assign a new value to a component and a snapshot operation that atomically reads and returns the values of all the components. To cope with the net effect of concurrency, asynchrony and failures, the algorithm implementing the update operation has to help concurrent snapshot operations in order they can always terminate.

This paper is on *partial snapshot* objects. Such an object provides a snapshot operation that takes a (dynamically defined) subset of the components as input parameter, and atomically reads and returns the values of this subset of components. The paper has two contributions. The first is the introduction of two properties for partial snapshot object algorithms, called *help-locality* and *uptodateness*. Help-locality requires that an update operation helps only the concurrent partial snapshot operations that read the component it writes. When an update of a component r helps a partial snapshot, uptodateness requires that the update provides the partial snapshot with a value of the component r that is at least as recent as the value it writes into that component. (No snapshot algorithm proposed so far satisfies these properties.) The second contribution consists of an update and a partial snapshot algorithms that are wait-free, linearizable and satisfy the previous efficiency properties. Interestingly, the principle that underlies the proposed algorithms is different from the one used so far, namely, it is based on the “write first, and help later” strategy. An improvement of the previous algorithms is also presented. Based on LL/SC atomic registers (instead of read/write registers) this improvement decreases the number of base registers from $O(n^2)$ to $O(n)$. This shows an interesting tradeoff relating the synchronization power of the base operations and the number of base atomic registers when using the “write first, and help later” strategy.

Key-words: Adaptive algorithm, Asynchronous shared memory system, Asynchrony, Atomicity, Atomic snapshot, Efficiency, Concurrency, Linearizability, LL/SC atomic registers, Locality, Partial snapshot, Process crash, Read/Write atomic register, Wait-free algorithm.

(Résumé : tsvp)

[*] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France damien.imbs@irisa.fr
[**] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

N’aider que si c’est nćessaire : Capture efficace d’instantanés partiels

Résumé : Ce rapport présente un algorithme efficace pour la lecture atomique d’un ensemble de variables partagées par des processus concurrents sujets à la défaillance par crash.

Mots clés : Atomicité, Contrôle de la concurrence, Crash de processus, Mémoire partagée asynchrone, Capture d’instantané, Linéarizabilité, Registre atomique, Opérations LL/SC, Localité, Synchronisation sans attente.

1 Introduction

1.1 Context of the study: snapshot objects

Shared memory snapshot objects Snapshot objects have been introduced in [1]. Considering a shared memory system made up of base atomic read/write registers, that can be concurrently accessed by asynchronous processes prone to crash, a snapshot object is an object that (1) consists of m components (each component being a base atomic register that can contain an arbitrary value), and (2) provides the processes with two operations, denoted update() and snapshot(). The update() operation allows the invoking process to atomically store a new value in an individual component. Differently, the snapshot() operation returns the values of all the components as if they had been read simultaneously.

From an execution point of view, a snapshot object has to satisfy the safety property called *linearizability*: the update and snapshot operations have to appear as if they had been executed one after the other, each being instantaneously executed at some point of the time line comprised between its start event and its end event [19]. From a liveness point of view, each update or snapshot operation has to terminate if the invoking process does not crash. This liveness property is called *wait-freedom* [16]. It means that an operation issued by a correct process has to terminate whatever the behavior of the other processes (the fact that some processes crash or are very slow cannot prevent an operation from terminating, as long as the issuing process does not crash). Wait-freedom is starvation-freedom despite asynchrony and process failures. In order to implement the wait-freedom property, a process that issues an update() operation can be required to help terminate the processes that have concurrently issued a snapshot() operation (preventing them from looping forever). This helping mechanism is required to ensure that all the snapshot() operations (issued by processes that do not crash) do always terminate [1].

The snapshot abstraction The snapshot object has proved to be a very useful abstraction for solving many other problems in asynchronous shared memory systems prone to process crashes, such as approximate agreement, randomized consensus, concurrent data structures, etc. A snapshot object hides the "implementation details" that are difficult to cope with in presence of the net effect of concurrency, asynchrony and failures. It is important to notice that, from a computational point of view, a snapshot object is not more powerful than the base atomic read/write objects it is built from. It only provides a higher abstraction level.

Shared memory snapshot vs message-passing snapshot The values returned by a snapshot() operation is a value of the part of the shared memory that is encapsulated in the corresponding snapshot object. It follows from the linearizability property satisfied by a snapshot object that there is a time instant at which the values returned by a snapshot() operation were simultaneously present in the shared memory, this time instant belonging to the time interval associated with that snapshot() operation.

The previous observation is in contrast with the notion of *distributed snapshot* used to capture consistent global states in asynchronous message-passing systems [11] where two distributed snapshots obtained by two processes can be consistent but incomparable in the sense that they cannot be linearized. The set of all the distributed snapshots that can be obtained from a message-passing distributed execution has only a lattice structure (basically, they can be partially ordered but not totally ordered). In that sense, the abstraction level provided by a shared memory snapshot object is a higher abstraction level than the one offered by message-passing distributed snapshots. They hide more asynchrony.

Types of snapshot objects Two types of snapshot objects have been investigated: single-writer and multi-writer snapshot objects. A single-writer snapshot object has one component per process, and the component associated with a process can be written only by that process. The number of components (m) is then the same as the number of processes (n). The base registers from which a single-writer snapshot object is built are then single-writer/multi-reader atomic registers. Wait-free algorithms implementing single-writer snapshot objects for n processes are described in [1]. Their costs is $O(n^2)$ (when counting the number of shared memory accesses). An algorithm whose cost is $O(n \log(n))$ is described in [9]. An implementation suited to systems with a possibly infinite number of processes (but where finitely many processes can take steps in each finite time interval) is described in [3]. An implementation that is adaptive to total contention (i.e., adaptive to the actual number $k \in [1..n]$ of processes that access the snapshot object during an entire execution [6]) is described in [7]. Its cost is $O(k \log(k))$.

A multi-writer snapshot object is a snapshot object of which each component can be written by any process. So, the base read/write registers on which its implementation relies are multi-writer/multi-reader atomic registers. Wait-free algorithms implementing multi-writer snapshot objects made up of m base components are described in [4, 21, 22]. The algorithm described in [21] has a linear cost $O(n)$. A short survey of algorithms that implement single-writer and multi-writer snapshot objects is presented in [12].

The notion of partial snapshot Usually, when a process invokes the $\mathsf{snapshot}()$ operation, it is not interested in obtaining the values of all the components, but in the values of a given subset of the components. A *partial snapshot* operation (denoted $\mathsf{p\_snapshot}()$) is a generalization of the base $\mathsf{snapshot}()$ operation. It takes a sequence $R =< r_1, \cdots, r_x >$ of component indices as input parameter, and returns a sequence $< v_1, \cdots, v_x >$ of values such that the value v_ℓ is the value of the component whose index is r_ℓ (an invocation of $\mathsf{p\_snapshot}()$ that considers all the components is actually a $\mathsf{snapshot}()$ invocation). As before, the invocations of $\mathsf{p\_snapshot}()$ and $\mathsf{update}()$ have to be linearizable, and their implementation has to be wait-free. The notion of partial snapshot object has first been introduced and investigated in [8].

1.2 Content of the paper and related work

Related work This paper is on the efficient wait-free implementation of multi-writer/multi-reader partial snapshot objects in the base read/write shared memory model augmented with an underlying *active set* object [2]. Such an object offers three operations: $\mathsf{Join}()$, $\mathsf{Leave}()$ and $\mathsf{GetSet}()$. Basically, $\mathsf{Join}()$ adds the invoking process to the active set, while $\mathsf{Leave}()$ suppresses it from this set; $\mathsf{GetSet}()$ returns the current value of the active set. (There are efficient adaptive read/write implementations of an active set, i.e., implementations whose number of read/write shared memory accesses depends only on the number of processes that invoke $\mathsf{Join}()$ and $\mathsf{Leave}()$ [2]. So, the base model used in this paper is the read/write atomic register model.

An algorithm based on read/write atomic registers and an active set object, that implements a partial snapshot object is described in [8] (as far as we know, it is the only such algorithm proposed so far) [1]. That algorithm extends the basic full snapshot algorithm described in [1]. It is based on the following principle. Each invocation of $\mathsf{p\_snapshot}(R)$ first makes public the list $R =< r_1, \cdots, r_x >$ of indices of the components it wants to read. Then, it sequentially invokes $\mathsf{Join}()$, an internal $\mathsf{embedded\_snapshot}(R)$ operation, and finally $\mathsf{Leave}()$. The update operation works as follows. When a process p_i invokes $\mathsf{update}(r, v, i)$ (where v is the value it wants to assign to the component whose index is r), it first invokes $\mathsf{GetSet}()$ to have a view of all the processes that are concurrently executing a $\mathsf{p\_snapshot}()$ operation. To guarantee the wait-freedom property (any process that does not crash has to terminate its operations), p_i helps terminate all the concurrent $\mathsf{p\_snapshot}()$ operations. To that end, it executes an $\mathsf{embedded\_snapshot}()$ operation whose input includes all the components read by the processes in the active set (whose value has been obtained by

[1] That paper presents also another algorithm implementing a partial snapshot object, that is based on read/write atomic registers, and more sophisticated registers that support Fetch&Add and Compare&Swap atomic operations. These more sophisticated registers are mainly used to obtain an efficient implementation of the underlying active set object. Here we consider the pure base read/write atomic register model.

the $\mathsf{GetSet}()$ invocation). In that way, if p_i does not crash, a concurrent $\mathsf{p\_snapshot}()$ operation can retrieve the values it is interested in from the values returned by the $\mathsf{embedded\_snapshot}()$ issued by p_i.

Features of the proposed algorithm The update and partial snapshot algorithms proposed here have several noteworthy features that make them different from all the previous full/partial snapshot algorithms (as far as we know). These features are the "write first, help later" strategy, and the cheap way helping is realized. They result from the additional help-locality and uptodateness properties the update and snapshot algorithms are required to satisfy.

Uptodateness The aim of the *uptodateness* property is to oblige an update operation that helps a snapshot operation to provide that snapshot with values as recent as possible. More precisely, let $\mathsf{up} = \mathsf{update}(r, v, i)$ be an update invoked by the process p_i to write the value v in the component r of the partial snapshot object, and $\mathsf{psp} = \mathsf{p\_snapshot}(R)$ be a concurrent snapshot invocation such that $r \in R$. Moreover, let us suppose that psp is helped by up, i.e., the values returned by psp have been provided by up. Uptodateness requires that the value returned for the component r be v or a more recent value (as each component is an atomic register, the notion of "more recent" is well defined)[2].

To obtain that property, the update algorithm proposed in the paper uses the "first write, help later" strategy (differently, the previous algorithms are based on the "help first, then write" strategy). As it allows providing a snapshot with more uptodate values, the uptodateness property shows that the unusual "write first, help later" strategy is really interesting and should maybe deserve more investigation.

Help-locality This property aims at obtaining more efficient update operations. To that end, it reduces the help provided to the partial snapshot operations by the update operations. More explicitly, let $\mathsf{update}(r, -, -)$ be an update operation that is concurrent with a partial snapshot operation $\mathsf{p\_snapshot}(R)$. The *help-locality* property demands the update not to help the partial snapshot if they do not conflict, i.e., if $r \notin R$. This means that, when $\mathsf{update}(r, -, -)$ is concurrent with $\mathsf{p\_snapshot}(R_1), \ldots, \mathsf{p\_snapshot}(R_z)$, it has to help only the ones such that $r \in R_\ell$ $(1 \leq \ell \leq z)$. This favors disjoint access parallelism.

As for uptodateness, the help-locality property is not ensured by the algorithms proposed so far to implement the update operation. The snapshot algorithm presented in [1] is very conservative: each update operation is required to compute one helping full snapshot value even when there is no concurrent snapshot. The partial snapshot algorithm described in [8] is a little bit less conservative: an $\mathsf{update}(r, -, -)$ operation concurrent with no snapshot operation is not required to help, but an $\mathsf{update}(r, -, -)$ operation concurrent with one or more $\mathsf{p\_snapshot}(R_\ell)$ operations $(1 \leq \ell \leq z)$ has to help each of them, whatever the sets R_ℓ, i.e., even the $\mathsf{p\_snapshot}(R_\ell)$ operations such that $r \notin R_\ell$.

An additional asynchrony feature An additional feature of the proposed update algorithm lies in its asynchrony and in the size of the helping snapshot values it computes. Previous (partial or full) snapshot implementations use one base atomic register $REG[r]$ per component r. These registers have to be large. They are made up of several fields, including a field for the last value written, a field storing a snapshot value used to help snapshot operations, and a few other fields containing control data. A snapshot value is made up of one value per component in the case of a full snapshot object, and one value for a subset of the components in the case of a partial snapshot object. Then, each $\mathsf{update}(r, v, -)$ operation atomically writes

[2] It is important to notice that neither the full snapshot algorithms proposed so far (e.g., [1]), nor the partial snapshot algorithm presented in [8], satisfies the uptodateness property. They all provide the snapshot with a component r value that is strictly older than v.

into $REG[r]$ both the new value v and a snapshot value. This means that the implementation of this atomic write can be time expensive.

Differently, thanks to the "write first, help later (and individually)" strategy, the proposed $\mathsf{update}(r, v, -)$ algorithm separates the write of the value v into $REG[r]$ and the individual writes of helping snapshot values, one for each concurrent $\mathsf{p\_snapshot}(R_\ell)$ operation such that $r \in R_\ell$. The fact that an $\mathsf{update}(r, v, -)$ operation first writes v, and helps, only later and individually, each concurrent partial snapshot that reads the component r, (1) allows those to obtain a value for the component r that is at least as recent as v, and (2) allows the use of several independent helping atomic registers that are written individually (thereby allowing more efficient atomic write operations). Moreover, the size of these atomic "array-like" registers can be smaller than m [3].

Motivation As the work described in [8], our aim is to better understand synchronization in presence of failures. From a more practical point of view, a $\mathsf{p\_snapshot}(R)$ operation can be seen as the reading part of a transaction that needs to obtain mutually consistent and uptodate values from the base objects specified in R. Such a study can help better understand the underlying foundations of software transactional memories [5, 13, 14, 18, 17, 20, 25].

Roadmap The paper is composed of 7 sections. Section 2 presents the base asynchronous read/write shared memory prone to process crashes, equipped with an active set object. Section 3 defines the atomic partial snapshot object and the help-locality and uptodateness properties. Then, Section 4 presents algorithms implementing the update and partial snapshot operations. These algorithms satisfy the previous properties. They are proved correct in Section 5. Section 6 discusses the proposed algorithms and presents a version of them based on LL/SC atomic registers (instead of read/write atomic registers). This improvement, that satisfies the help-locality and uptodateness properties, is more efficient than the base algorithm from a memory size point of view, namely it requires $O(n)$ LL/SC atomic registers instead of $O(n^2)$ read/write atomic registers. Finally, Section 7 provides a few concluding remarks.

2 Underlying shared memory model

2.1 Asynchronous shared memory model

The system is made up of n processes $p_1, \ldots, p_n$. The identity of p_i is i. These processes communicate through multi-writer/multi-reader atomic registers. Atomic means that each read or write operation on a register appears as if it has been executed sequentially at some point of the time line comprised between its start and end event. The registers are assumed to be reliable (this assumption is without loss of generality -from a computability point of view- as it is possible to build atomic reliable registers on top of crash prone atomic registers [10, 15, 17, 23]).

There is no assumption on the speed of processes: they are asynchronous. Moreover, up to $(n-1)$ processes may crash. Before it crashes (if it ever crashes), a process executes correctly its algorithm. A crash is a premature halt: after it has crashed, a process executes no more step. Given a run, a process that does not crash is *correct* in that run, otherwise it is *faulty* in that run.

[3] If each partial snapshot by a process p_i is on at most k components, the atomic "array-like" registers used to help p_i need to have only k entries. If $k << m$, the writes into such atomic k-size registers can generate less contention that writes into atomic m-size registers.

2.2 Active set object

We assume that the processes can access an *active set* object. Such an object, first proposed in [2], can be used to solve adaptive synchronization problems (e.g., [26]). As already indicated, its aim is to allow the processes to have a view of which of them are concurrently executing operations. To that end, an active set object provides the processes with three operations, $\mathsf{Join}()$, $\mathsf{Leave}()$ and $\mathsf{GetSet}()$ (informally described in the Introduction). These operations are not required to be atomic. (So, the definition of an operation cannot assume that the concurrent executions of other operations are both instantaneous and one at a time, they have to explicitly take into account the fact that their execution spans a finite period of time.)

Let S be an active set object. Initially, the predicate $i \notin S$ is true for any process p_i (S is empty). A process p_i executes the following sequence (expressed using the regular language notation) of operation invocations (from which the object identifier S is omitted):

$$\left[\mathsf{GetSet}()^*\left(\mathsf{Join}()\mathsf{Leave}()\right)^*\right]^*.$$

- Let us consider two consecutive operations $\mathsf{Join}()$ and $\mathsf{Leave}()$ (if any) issued by a process p_i. (This means that (1) $\mathsf{Leave}()$ follows immediately $\mathsf{Join}()$, or (2) $\mathsf{Join}()$ follows $\mathsf{Leave}()$ and only $\mathsf{GetSet}()^*$ can occur between them.)
  - From the end event of $\mathsf{Join}()$ until the start event of $\mathsf{Leave}()$, the predicate $i \in S$ is true.
  - From the end event of $\mathsf{Leave}()$ until the start event of $\mathsf{Join}()$, the predicate $i \in S$ is false.
  - During the execution of $\mathsf{Join}()$ or $\mathsf{Leave}()$, the predicate can be true or false.
- A $\mathsf{GetSet}()$ invocation returns a set of process ids including:
  - Each j such that the predicate $j \in S$ was continuously true during the execution of $\mathsf{GetSet}()$.
  - No j such that the predicate $j \in S$ was continuously false during the execution of $\mathsf{GetSet}()$.
  - Possibly some js such that the predicate $j \in S$ was not continuously true during $\mathsf{GetSet}()$.

As an example let us consider the $\mathsf{GetSet}()$ invocation depicted in Figure 1 (the length of a box indicates the time duration of the corresponding operation). That $\mathsf{GetSet}()$ invocation returns a set that (1) does contain k and does not contain j, and (2) can possibly contain ℓ or m.

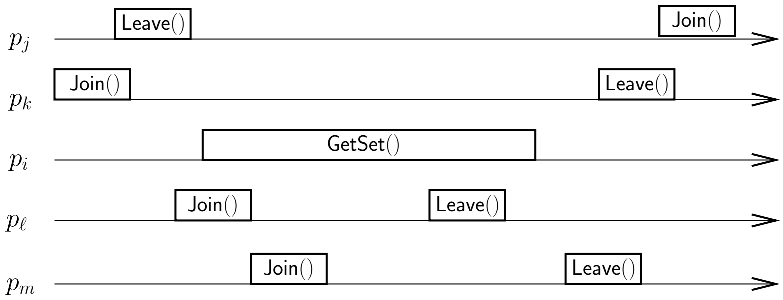

Figure 1: An example of active set object

An adaptive wait-free implementation of an active set object is described in [2]. *Adaptive* means that the cost of each operation (measured by the number of shared memory accesses) depends on the number of processes that have invoked $\mathsf{Join}()$ and have not yet terminated the corresponding $\mathsf{Leave}()$. As already indicated, *wait-free* means that any operation invocation issued by a correct process always terminates [16].

3 Definitions

3.1 Partial snapshot object

As already said in the Introduction, a multi-writer/multi-reader partial snapshot object is made up of m components (each being a multi-writer/multi-reader atomic register) that provides the processes with two operations $\mathsf{update}()$ and $\mathsf{snapshot}()$ such that:

- $\mathsf{update}(r, v, i)$ is invoked by p_i to write the value v in the component r ($1 \leq r \leq m$) of the snapshot object. That operation returns the control value ok.
- $\mathsf{p\_snapshot}(R)$, where R is a sequence $< r_1, \cdots, r_x >$ of component indexes, allows a process to obtain the value of each component in R. It returns a corresponding sequence of values $< v_1, \cdots, v_x >$.

A partial snapshot object is defined by the following properties.

- Termination. Every invocation of $\mathsf{update}()$ or $\mathsf{p\_snapshot}()$ issued by a correct process terminates.
- Consistency. The operations issued by the processes (except possibly the last operation issued by a faulty process[4]) appear as if they have been executed one after the other, each one being executed at some point of the time line between its start event and its end event.

The termination property is wait-freedom [16] (starvation-freedom despite concurrency and process crashes). The consistency property is linearizability [19] (here, it means that a $\mathsf{p\_snapshot}()$ operation always returns component values that were simultaneously present in the shared memory, and are uptodate).

3.2 Additional properties related to the implementation

These properties, that have been informally presented in the Introduction, do not concern the definition of the partial snapshot problem, but the way it is solved by the algorithms that implement its operations.

Definition 1 *The algorithms implementing the update and partial snapshot operations satisfy the* help-locality *property if, for any pair (r, R) such that $r \notin R$, an* $\mathsf{update}(r, -, -)$ *invocation never helps a* $\mathsf{p\_snapshot}(R)$ *operation.*

This property, related to efficiency, follows from the observation that an $\mathsf{update}(r, -, -)$ operation and a $\mathsf{p\_snapshot}(R)$ operation that are concurrent and such that $r \notin R$, are actually independent operations. (This is similar to a read on a register X and a write on a register $Y \neq X$ that are concurrent.) Intuitively, help-locality requires that the implementation does only what is necessary and sufficient.

Let $\mathsf{p\_snapshot}(< r_1, \ldots, r_n >)$ be a snapshot operation that returns a snapshot value $< v_1, \ldots, v_n >$ that has been computed by an update operation, say $\mathsf{update}(r, v, -)$. Hence, $r \in \{r_1, \ldots, r_n\}$ and the update *helps* the snapshot.

Definition 2 *The algorithms implementing the update and partial snapshot operations satisfy the* uptodateness *property if, for a snapshot operation* $\mathsf{p\_snapshot}(R)$ *that is helped by an* $\mathsf{update}(r, v, -)$ *operation, the value returned for the component r is at least as recent as v.*

The aim of this property is to provide the partial snapshot operations with values "as fresh as possible". As noticed in the Introduction, (to our knowledge) no pair of update/snapshot algorithms proposed so far satisfies help-locality or uptodateness.

[4] If such an operation does not appear in the sequence, it is as if it has not been invoked.

4 An efficient partial snapshot construction

This section presents a construction (Figures 2 and 3) of a partial snapshot object that satisfies the help-locality and uptodateness properties previously defined.

4.1 The underlying shared objects

The algorithms implementing the $\mathsf{p\_snapshot}()$ and $\mathsf{update}()$ operations use the following shared variables.

- An array, denoted $REG[1..m]$, of multi-writer/multi-reader atomic registers. The register $REG[r]$ is associated with the component r of the snapshot object. It is composed of three fields $< value, pid, sn >$, whose meaning is the following. $REG[r].value$ contains the current value of the component r; $REG[r].pid$ and $REG[r].sn$ are control data associated with that value. $REG[r].pid$ contains the id of the process that issued the corresponding $\mathsf{update}()$ operation, while $REG[r].sn$ contains its sequence number among all the $\mathsf{update}()$ operations issued by that process.
- An array, denoted $ANNOUNCE[1..n]$, of single-writer/multi-reader atomic registers. The register $ANNOUNCE[i]$ can be written only by p_i. This occurs when p_i invokes $\mathsf{p\_snapshot}(R)$: it then stores R in $ANNOUNCE[i]$ (the indexes $r_1, \cdots, r_x$ of the components it wants to read). In that way, if a process p_j has to help p_i to terminate its $\mathsf{p\_snapshot}()$ operation, it only has to read $ANNOUNCE[i]$ to know the components p_i is interested in.
- An array, denoted $HELPSNAP[1..n, 1..n]$, of single-writer/multi-reader atomic registers. The register $HELPSNAP[i, j]$ can be written only by p_i. When, while executing an $\mathsf{update}()$ operation, p_i is required to help p_j terminate its current $\mathsf{p\_snapshot}(< r_1, \cdots, r_x >)$ operation, it does so by depositing in $HELPSNAP[i, j]$ a sequence of values $< v_1, \cdots, v_x >$ that can be used by p_i as the result of its $\mathsf{p\_snapshot}(< r_1, \cdots, r_x >)$ operation.

The shared variables are denoted with upper case letters. Differently, the local variables are denoted with lower case letters (those are introduced in the algorithm description).

4.2 The $\mathsf{p\_snapshot}()$ operation

The algorithm that implements this operation is described in Figure 2. Similarly to [8], it borrows its underlying principle from [1]. More precisely, it first uses a "sequential double scan" to try to terminate by itself. If it cannot terminate by itself, it looks for a process that could help it terminate (namely, a process that has issued two updates on a component it wants to read).

Startup When it invokes $\mathsf{p\_snapshot}(R)$, a process p_i first announces the components it wants to read (line 01) and invokes $\mathsf{Join}()$ (line 02). This is in order to allow the processes that concurrently update a component of R to help it.

Sequential double scan Then, the process p_i enters a loop (line 04-18). During each execution of the loop body, it uses a pair of scans of the registers $REG[r]$ for the components it is interested in, namely $\{r_1, \dots, r_x\}$. It is important to notice that these are sequential [1]: the second scan always starts after the previous one has terminated. The values obtained by the first scan are kept in the array aa (line 03 for the first loop, and then line 05 followed by line 17), while the values obtained by the second scan are kept in the array bb (line 05).

```
operation p_snapshot($< r_1, \cdots, r_x >$):    % (code for $p_i$) %
(01) $ANNOUNCE[i] \leftarrow < r_1, \cdots, r_x >$;
(02) $can\_help\_me_i \leftarrow \emptyset$; Join();
(03) for each $r \in \{r_1, \cdots, r_x\}$ do $aa[r] \leftarrow REG[r]$ end for;
(04) while $true$ do % Lin point if return at line 08 %
(05)     for each $r \in \{r_1, \cdots, r_x\}$ do $bb[r] \leftarrow REG[r]$ end for;
(06)     if $(\forall r \in \{r_1, \cdots, r_x\} : aa[r] = bb[r])$ then
(07)         Leave();
(08)         return($< bb[r_1].value, \cdots, bb[r_x].value >$)
(09)     end if;
(10)     for each $r \in \{r_1, \cdots, r_x\}$ such that $(aa[r] \neq bb[r])$ do
(11)         $can\_help\_me_i \leftarrow can\_help\_me_i \cup \{< w, sn >\}$ where $< -, w, sn >= bb[r]$
(12)     end for;
(13)     if $\big(\exists < w, sn1 >, < w, sn2 > \in can\_help\_me_i$ such that $sn1 \neq sn2\big)$ then
(14)         Leave();
(15)         return($HELPSNAP[w, i]$)
(16)     end if;
(17)     $aa \leftarrow bb$
(18) end while.
```

Figure 2: An algorithm for the p_snapshot() operation

Try first to terminate without help If, for each $r \in \{r_1, \ldots, r_x\}$, it observes no change in $REG[r]$ (test of line 06), p_i can conclude that at any point of the time line between the end of the first scan and the beginning of the second one, no $REG[r]$, $r \in \{r_1, \ldots, r_x\}$, has been modified. This is called a *successful double scan*. Hence, the values read in bb were simultaneously present in the snapshot object: they can be returned as the result of the p_snapshot($< r_1, \cdots, r_x >$) invocation (line 08). In that case, before terminating, p_i invokes Leave() to announce that it does no longer need help (line 07).

Otherwise, try to benefit from the helping mechanism While until that point, the statements previously described are the same as the ones used in [1, 8], the statements that follow are different. This difference is mainly due to the "write first, then help" strategy, and its impact on the way it is exploited by the algorithm.

If the test of line 06 is not satisfied, p_i uses the helping mechanism that (from its side) works as follows. As the test is false, there is at least one component $r \in \{r_1, \cdots, r_x\}$ that has been updated between the two scans. For each such component r, p_i considers the identity of the last write, namely the pair $< w, sn >$ extracted from $bb[r] =< -, w, sn >$ (the last writer of $REG[r]$ is p_w and sn is the increasing sequence number it has associated with the corresponding update); p_i adds this pair to a local set $can\_help\_me_i$ where it stores the processes that could help it (lines 10-12).

Then, p_i checks if it can terminate thanks to the helping mechanism (lines 13-16). The helping termination predicate is as follows: "p_i has observed that there is a process p_w that has issued two different updates (on any pair of components)". From an operational point of view, this is captured by the fact that p_w appears twice in $can\_help\_me_i$ (line 13). As we will see in the proof, the fact that this predicate is true means that p_w has determined a set of values $< v_1, \cdots, v_x >$ (kept in $HELPSNAP[w, i]$) that p_i can use as the result of its p_snapshot($< r_1, \cdots, r_x >$) operation. In that case, p_i invokes Leave() to indicate it does no longer need help and returns the content of $HELPSNAP[w, i]$ (lines 14-15).

Finally, if the helping predicate is false, p_i cannot terminate and consequently enters again the loop body (after having shifted the array bb in the array aa, line 17).

4.3 The update() operation

The algorithm for the update() operation is described in Figure 3. The invoking process p_i first writes the new value (line 01, where nbw_i is a local sequence number generator), and then (lines 02-30) asynchronously helps the other processes. As indicated in the Introduction, the principles that underlie this mechanism differ from the ones used in previous snapshot/update algorithms. Let update(r, v, i) be an update invocation. The helping mechanism works as follows.

```
operation update$(r,v,i)$:     % (code for $p_i$) %
(01) $nbw_i \leftarrow nbw_i + 1$; $REG[r] \leftarrow < v, i, nbw_i >$; % Lin Point %
(02) $readers_i \leftarrow$ GetSet(); $to\_help_i \leftarrow \emptyset$;
(03) for each $j \in readers_i$ do
(04)     $announce_i[j] \leftarrow ANNOUNCE[j]$;
(05)     if $(r \in announce_i[j])$ then $to\_help_i \leftarrow to\_help_i \cup \{j\}$ end if
(06) end for;
(07) if $(to\_help_i = \emptyset)$ then return$(ok)$ end if;
(08) $to\_read_i \leftarrow \big(\bigcup_{j \in to\_help_i} announce_i[j]\big)$ expressed as a sequence $< rr_1, \dots, rr_y >$;
(09) for each $j \in to\_help_i$ do $can\_help_i[j] \leftarrow \emptyset$ end for;
(10) for each $rr \in to\_read_i$ do $aa[rr] \leftarrow REG[rr]$ end for;
(11) while $(to\_help_i \neq \emptyset)$ do
(12)     for each $rr \in to\_read_i$ do $bb[rr] \leftarrow REG[rr]$ end for;
(13)     $still\_to\_help_i \leftarrow \emptyset$;
(14)     for each $rr \in to\_read_i$ such that $aa[rr] \neq bb[rr]$ do
(15)         for each $j \in to\_help_i$ such that $rr \in announce_i[j]$ do
(16)             $still\_to\_help_i \leftarrow still\_to\_help_i \cup \{j\}$;
(17)             $can\_help_i[j] \leftarrow can\_help_i[j] \cup \{< w, sn >\}$ where $< -, w, sn >= bb[rr]$
(18)         end for
(19)     end for;
(20)     for each $j \in to\_help_i \backslash still\_to\_help_i$ do
(21)         $HELPSNAP[i,j] \leftarrow < bb[r_1].value, \dots, bb[r_x].value >$ where $< r_1, \dots, r_x >= announce_i[j]$
(22)     end for;
(23)     for each $j \in still\_to\_help_i$ do
(24)         if $(\exists < w, sn1 >, < w, sn2 > \in can\_help_i[j]$ such that $sn1 \neq sn2)$ then
(25)             $HELPSNAP[i,j] \leftarrow HELPSNAP[w,j]$; $still\_to\_help_i \leftarrow still\_to\_help_i \backslash \{j\}$
(26)         end if
(27)     end for;
(28)     $to\_help_i \leftarrow still\_to\_help_i$; $to\_read_i \leftarrow \big(\bigcup_{j \in to\_help_i} announce_i[j]\big)$; $aa \leftarrow bb$
(29) end while;
(30) return$(ok)$
```

Figure 3: An algorithm for the update() operation

Are there processes to help? As in [8], a process p_i first invokes GetSet() to learn the set of processes that have concurrently invoked a p_snapshot(R) operation (line 02). It then computes the set to_help_i of the conflicting processes, i.e., the ones such that $r \in R$. To that end, it uses the array $ANNOUNCE$ (lines 03-06). If there is no conflicting process ($to\_help_i = \emptyset$), p_i does not have to help (help-locality property), and terminates accordingly (line 07).

If $to\_help_i \neq \emptyset$, p_i has to possibly help the processes in to_help_i. To that end, it first computes the set to_read_i of the components it has to read to help these processes (line 08). It also initializes a local array can_help_i to $\emptyset$ (line 09). The entry $can\_help_i[j]$ contains the processes p_w that (to p_i's knowledge) could also help the conflicting process p_j.

How a process helps individually another process Each process p_j in to_help_i is helped individually by p_i. This is done in the loop (line 11-29), that terminates when the set to_help_i becomes empty.

In each loop iteration, similarly to what is done in the $\mathsf{p\_snapshot}()$ operation, p_i first executes a double scan (whose values are kept in the local arrays aa and bb) and does the following.

- Part 1: lines 13-19. For each component rr such that $aa[rr] \neq bb[rr]$, let $bb[rr] =< -, w, sn >$, which means that (to p_i's knowledge) p_w is the last process that wrote the component rr (lines 14 and 17). Moreover, this write occurred between the double scan. According to that observation, p_i keeps track of the fact that such a process p_w could help every p_j such that $rr \in announce_i[j]$. This is done by adding the pair $< w, sn >$ to the set $can\_help_i[j]$ (lines 15-17). Additionally, p_i adds j to the set $still\_to\_help_i$ (lines 13 and 16).
- Part 2: lines 20-22. Then, p_i looks for the processes that can be helped directly. Those are the processes p_j such that $j \in to\_help_i \setminus still\_to\_help_i$. The components rr they want to read are such that $aa[rr] = bb[rr]$, which means that the pair (aa, bb) constitutes a successful double scan. Accordingly, p_i writes in $HELPSNAP[i, j]$ a snapshot value that p_j can use if its partial snapshot operation is still pending. The value of this helping snapshot value is $< bb[r_1].value, \dots , bb[r_x].value >$ where $< r_1, \dots , r_x >= announce_i[j]$ (line 21).
- Part 3: lines 23-27. For each process p_j that it has not previously helped (line 23), p_i looks if there is a process p_w that can help p_j. The helping termination predicate (line 24) is the same as the one used in the $\mathsf{p\_snapshot}()$ algorithm (line 13 in Figure 2): there are two writes issued by a process p_w that appear in $can\_help_i[j]$. If the predicate is true, the helping value provided to p_j by p_w is borrowed by p_i to help p_j (line 25).
- Part 4: line 28. Finally, p_i updates to_help_i and to_read_i before entering again the loop. If $to\_help_i = \emptyset$, the loop terminates.

5 Proof of the algorithm

5.1 Preliminary definitions

Definition 3 *If a process terminates a partial snapshot operation at line 08 (Figure 2), or executes line 21 of an update operation (Figure 3), the last pair of readings of the array REG constitutes a* successful double scan. *(The first scan is the reading whose values are kept in the array $aa[\,]$, while the second scan is the reading whose values are kept in the array $bb[\,]$.)*

Definition 4 *Let* psp *be a partial snapshot operation issued by a process p_i.*

- psp *is* 0-helped *if it terminates with a successful double scan (Figure 2, line 08).*
- psp *is* 1-helped *if it terminates by returning $HELPSNAP[w1, i]$ (Figure 2, line 15), and the values in $HELPSNAP[w1, i]$ come from a successful double scan by p_{w1} (i.e., the values in $HELPSNAP[w1, i]$ have been computed at line 21 of Figure 3 by an update issued by p_{w1}).*
- psp *is* 2-helped *if it terminates by returning $HELPSNAP[w1, i]$ (Figure 2, line 15), and the values in $HELPSNAP[w1, i]$ come from $HELPSNAP[w2, i]$ (p_{w1} executed line 25, Figure 3) which in turn come from a successful double scan by p_{w2} (i.e., the helping update by p_{w2} computed these values at line 21).*
- *For the next values of h, the h-*helped *notion is defined similarly.*

The aim of the following definitions is to help prove that the values returned are "consistent", i.e., they are from the appropriate registers, were simultaneously present in the snapshot object and are recent[5].

[5] Always returning the initial values would provides well-defined and mutually consistent values, but those would not be fresh values.

Definition 5 *The values* $< v_1, \cdots, v_y >$ *returned by a* p_snapshot$(< r_1, \dots, r_x >)$ *operation are* well defined *if* $x = y$ *and for each* ℓ*,* $1 \leq \ell \leq x$*, the value* v_ℓ *has been read from* $REG[r_\ell]$*.*

Definition 6 *The values returned by a* p_snapshot$(< r_1, \dots, r_x >)$ *operation are* mutually consistent *if there is a time at which they were simultaneously present in the snapshot object.*

Definition 7 *The values returned by a* p_snapshot$(< r_1, \dots, r_x >)$ *operation are* fresh *if, for each* ℓ*,* $1 \leq \ell \leq x$*, the value* v_ℓ *returned for* r_ℓ *is not older than the last value written into* $REG[r_\ell]$ *before the partial snapshot invocation*[6]*.*

5.2 The values returned are well-defined, mutually consistent and fresh

Lemma 1 *The values returned by a* 0*-helped* p_snapshot$()$ *operation are well-defined, mutually consistent and fresh.*

Proof Let p_snapshot$(< r_1, \dots, r_x >)$ be a 0-helped snapshot operation. As it terminates at line 08, it follows that the value returned for each component $r \in \{r_1, \dots, r_x\}$ has been obtained from $bb[r].value$. The well-definition of these values follows directly from the observation that, for each $r \in \{r_1, \cdots, r_x\}$, the value currently in $bb[r]$ has been obtained at line 05 from the corresponding register $REG[r]$.

For mutual consistency and freshness, let us notice that the termination of the p_snapshot$()$ operation is due to a successful double scan. None of the values read has been modified between these two scans (otherwise the predicate of line 06 would be false). As these scans are sequential (the second one is started only after the first one has terminated) it immediately follows that, at any time between these two scans, the values that are returned were simultaneously present in the shared memory and, for each v_ℓ, no write into $REG[r_\ell]$ occurred between the write of $< v_\ell, -, - >$ into $REG[r_\ell]$ and its read whose value has been kept in $bb[r_\ell]$. It follows that the values returned are both mutually consistent and fresh, which proves the lemma. $\Box_{Lemma\ 1}$

Lemma 2 *The values returned by a* 1*-helped* p_snapshot$()$ *operation are well-defined, mutually consistent and fresh.*

Proof Let psp $=$ p_snapshot$(< r_1, \dots, r_x >)$ be the 1-helped snapshot operation and p_i the process that invoked it. As that operation returns $HELPSNAP[w1, i]$ at line 15, it follows that the predicate evaluated at line 13 is true: there is a process p_{w1} such that there are two distinct pairs $< w1, sn1 >, < w1, sn2 > \in can\_help\_me_i$. Without loss of generality let $sn1 < sn2$. So, $w1$ has been added twice to $can\_help\_me_i$ (line 11). The proof follows from the following sequence of observations. The time instants defined and used in the proof are depicted in Figure 4. Except when explicitly indicated, the line numbers refer to Figure 2.

1. As $w1$ appears twice in $can\_help\_me_i$, it follows that the process p_{w1} has issued two distinct updates, say update$(r1, v, w1)$ and update$(r2, v', w1)$, that entailed two writes (to $REG[r1]$ and $REG[r2]$) such that $r1, r2 \in \{r_1, \dots, r_x\}$ (let notice that it is possible that $r1 = r2$).
2. It follows from the addition of $< w1, sn1 >$ to $can\_help\_me_i$ (that is due to the component $r1$), that there are two time instants t_1 and t_2, and two different values $aa1$ and $bb1$, such that:
   - There is a process p_u that has written $aa1$ into $REG[r1]$, and that value has then been read by p_i and stored in $aa[r1]$ at time t_1 (line 03 or lines 05 and 17).

[6] Let us recall that, as each $REG[r_\ell]$ is an atomic register, its read and write operations can be totally ordered in a consistent way. The word “last” is used with respect to this total order.

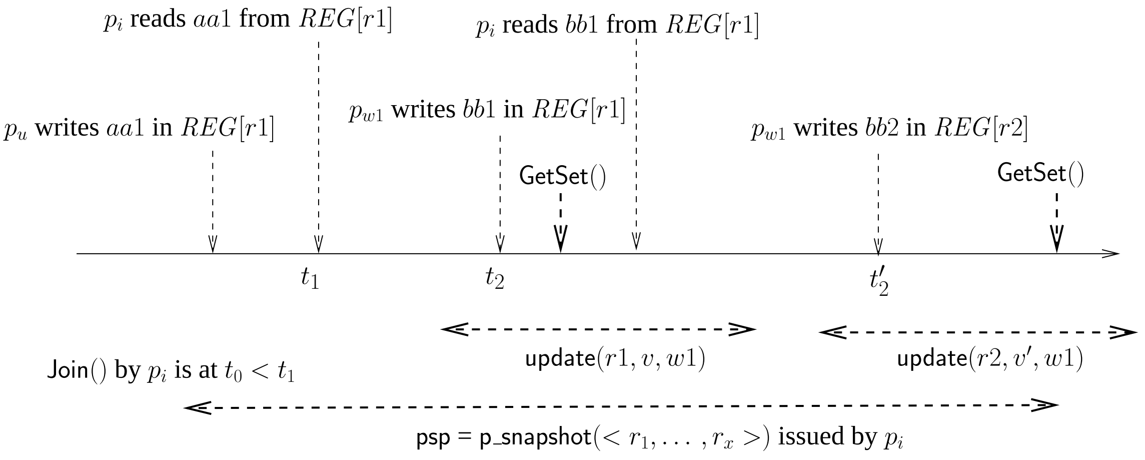
p_i reads $aa1$ from $REG[r1]$
p_i reads $bb1$ from $REG[r1]$
p_u writes $aa1$ in $REG[r1]$
p_{w1} writes $bb1$ in $REG[r1]$
p_{w1} writes $bb2$ in $REG[r2]$
GetSet()
GetSet()
t_1
t_2
t'_2
update$(r1, v, w1)$
update$(r2, v', w1)$
Join() by p_i is at $t_0 < t_1$
psp = p_snapshot($< r_1, \dots, r_x >$) issued by p_i

Figure 4: Order on operations on base objects

- p_{w1} has written $bb1 =< v, w1, sn1 >$ into $REG[r1]$ at time t_2 (line 01 of Figure 3), and that value has later been read by p_i and stored in $bb[r1]$ (line 05).
- As p_i reads from the atomic register $REG[r1]$ first $aa1$ and then $bb1 \neq aa1$, it follows that $t_1 < t_2$.

3. Similarly to the previous item, it follows from the addition of $< w1, sn2 >$ to $can\_help\_me_i$ (due to $r2$) that there are two time instants t'_1 and t'_2, and two different values $aa2$ and $bb2$, such that:
   - There is a process p_s that has written $aa2$ into $REG[r2]$, and that value has then been read by p_i and stored in $aa[r2]$ at time t'_1 (line 03 or lines 05 and 17).
   - p_{w1} has written $bb2 =< v', w1, sn2 >$ into $REG[r2]$ at time t'_2 such that $t'_1 < t'_2$ (line 01 of Figure 3), and that value has later been read by p_i and stored in $bb[r2]$ (line 05).
4. As p_{w1} is sequential and $sn1 < sn2$, it follows that p_{w1} has first executed update$(r1, v, w1)$ (that entailed the write of $REG[r1]$) before executing update$(r2, v', w1)$ (that entailed the write $REG[r2]$). It follows from that observation that $t_2 < t'_2$.
5. Let t_0 be the time at which p_i started its Join() invocation (line 02). As p_i executes Join() before reading entries of REG, we have $t_0 < t_1$, and consequently $t_0 < t_2 < t'_2$.
6. When p_{w1} executed update$(r1, v, w1)$, it first wrote $REG[r1]$, and only then invoked GetSet() (lines 01-02 of Figure 3). As $t_0 < t_2$, it follows from the specification of the underlying active set object that i belongs to the set $readers^{r1}_{w1}$ returned by the GetSet() invocation issued by update$(r1, v, w1)$.
7. The values returned by psp have been deposited in $HELPSNAP[w1, i]$ either by the first update update$(r1, v, w1)$ or the second update update$(r2, v', w1)$. Let up1 be this update. If they have been deposited by the second update, it follows from the lines 02-06 of Figure 3 that we necessarily have $i \in readers^{r2}_{w1}$.

   The next item shows that, whatever the update that deposited in $HELPSNAP[w1, i]$ the values that are read by p_i, those are well defined, mutually consistent and fresh.
8. It follows from the following facts:
   - up1 is on a component $r1$ (or $r2$) that p_i wants to read (Item 1),
   - The predicate $i \in readers_{w1}$ is true when evaluated in up1,
   - p_i updates $ANNOUNCE[i]$ to $< r_1, \dots, r_x >$ before invoking Join(),

- up1 invokes GetSet() (and obtains $readers_{w1}$) before reading the array $ANNOUNCE$,

that up1 sets $announce_{w1}[i]$ to $< r_1, \dots, r_x >$ and includes i in to_help_{w1} (lines 04-05 of Figure 3).

9. As psp is a 1-helped snapshot operation (assumption), the values read by psp from $HELPSNAP[w1, i]$ have been deposited by up1 at line 21 (Figure 3). As they are the values of the components in $announce_{w1}[i]$, they are well defined. Moreover, as they are from a successful double scan, they are mutually consistent. As the double scan started after the beginning of psp, they are fresh.

$\Box_{Lemma\ 2}$

Lemma 3 *For $h \geq 2$, the values returned by a h-helped partial snapshot operation are well-defined, mutually consistent and fresh.*

Proof We state the proof for $h = 2$. The proof of the cases $h \geq 3$ is obtained from a simple induction on h. Let psp $=$ p_snapshot($< r_1, \dots, r_x >$) be a 2-helped partial snapshot operation that, invoked by a process p_i, returns the values in $HELPSNAP[w1, i]$. These values have been written by p_{w1} in $HELPSNAP[w1, i]$ at line 25 of an update operation (denoted upw1). Moreover, that write assigned to $HELPSNAP[w1, i]$ the current value of $HELPSNAP[w2, i]$, that has been computed at line 21 of an update (denoted upw2) issued by p_{w2}. We have the following.

- The Items 1-8 stated in the proof of the Lemma 2 are still valid when considering psp and upw1. (This follows from the observation that only the Item 9 uses the fact that psp be a 1-helped partial snapshot operation.) It follows that $i \in readers_{w1}$ and $announce_{w1}[i] =< r_1, \dots, r_x >$.
- *Claim.* upw2 is such that $announce_{w2}[i] =< r_1, \dots, r_x >$.
  Proof of the claim. As p_{w1} executes $HELPSNAP[w1, i] \leftarrow HELPSNAP[w2, i]$ (line 25, Figure 3), there are two components $r1$ and $r2$ such that (1) $r1, r2 \in announce_{w1}[i] =< r_1, \dots, r_x >$. and (2) p_{w2} issued two updates involving $REG[r1]$ and $REG[r2]$, and (3) one of these updates is upw2.

  As upw2 started after the beginning of upw1, that in turn started after the Join() invoked by psp, it follows that the set returned by the GetSet() invocation by upw2 includes i, and consequently, $announce_{w2}[i] =< r_1, \dots, r_x >$ (lines 02-06 and lines 20-22 of Figure 3, executed by upw2). *End of the proof of the claim.*
- If follows from the previous items and the fact that the values in $HELPSNAP[w2, i]$ are determined from a successful double scan at line 21 by upw2, that they are well-defined, mutually consistent and fresh.

$\Box_{Lemma\ 3}$

5.3 Uptodateness and help-locality

Lemma 4 *The* update() *and* p_snapshot() *algorithms satisfy the uptodateness and help-locality properties.*

Proof The help-locality follows immediately from line 05 of the update algorithm. The uptodateness property follows from the fact that an update($r, v, -$) operation first writes v into $REG[r]$, and only then reads the value in $REG[r]$ if it is needed for helping some processes. $\Box_{Lemma\ 4}$

5.4 Wait-freedom

The next lemma shows that the algorithms that construct a partial snapshot object are wait-free, i.e., terminate despite the crash of any number of processes [16].

Lemma 5 *When executed by a correct process, every* update() *and* p_snapshot() *operation terminates.*

Proof Let us recall that we assume a wait-free implementation of the underlying active set object. The proof is similar to the proof described in [1] designed for for a snapshot object whose components are single-writer/multi-reader registers.

Let us first consider a p_snapshot() invocation, issued by a correct process p_i. If, while p_i executes line 06, the test is true, the p_snapshot() operation terminates. So, let us assume that this test is never satisfied. We have to show that the predicate of line 13 eventually becomes true. As the predicate of line 06 is never satisfied, each time p_i executes the loop body, there is a component r such that $aa[r] \neq bb[r]$. The process p_k that has modified $REG[k]$ between the two readings by p_i entails the addition of the pair $< k, snk >$ to $can\_help\_me_i$ (where $< k, snk >$ is extracted from $bb[r]$). In the worst case, $n-1$ pairs (one associated with each process, but p_i because it cannot execute an update operation while it executes a snapshot operation) can be added to $can\_help\_me_i$ while the predicate of line 13 remains false. But once $can\_help\_me_i$ contains one pair per process (but p_i), the next pair that is added is necessarily due to a process p_w such that $can\_help\_me_i$ already contains a pair $< w, sn1 >$. Consequently, after line 11 has been executed due to that process p_w, a second pair $< w, sn2 >$ is added to $can\_help\_me_i$. Then, the test of line 13 becomes satisfied, which proves the lemma.

Let us now consider an update() invocation, issued by a correct process p_i. If, when computed at line 05, to_help_i remains empty, p_i terminates at line 07. So, let us suppose that $to\_help_i \neq \emptyset$, and p_i executes the **while** loop (lines 11-29).

We have to show that the set to_help_i eventually becomes empty. The processes that are not added to the set $still\_to\_help_i$ (line 16) are suppressed from to_help_i (line 28). So, let us assume (by contradiction) that there is a non empty set $to\_help\_forever_i \subseteq to\_help_i$ of processes that are added to $still\_to\_help_i$ each time p_i executes the body of the **while** loop (line 16). Let p_j be one of these processes. As $still\_to\_help_i$ is reset to $\emptyset$ each time p_i executes the loop body and p_j remains forever in $to\_help\_forever_i$, it follows that there is a pair $< w, sn >$ that is added to $can\_help_i[j]$ at each execution of the loop body. The reasoning is now the same as for proving the termination of the p_snapshot() algorithm. One new pair is added to $can\_help_i[j]$ each time p_i executes the body of the loop and there are $n-1$ processes different from p_i. Consequently, after at most n executions of the loop body, the new pair $< w', sn' >$ that is added to $can\_help_i[j]$ is such that $w = w'$ and $sn \neq sn'$. Then, the test of line 24 becomes satisfied and p_j is suppressed from $still\_to\_help_i$, contradicting the initial assumption, which completes the proof of the lemma. $\square_{Lemma\ 5}$

5.5 Linearizability

As stated in the Introduction and Section 3.1, the consistency property of a snapshot object is linearizability. This means that all the update() and p_snapshot() operations issued by the processes during a run (except possibly the last operation issued by faulty processes), have to appear as if they were executed one after the other, each one being executed at some point of the time line between its start event and its end event.

Lemma 6 *Every run of a partial snapshot object whose* update() *and* p_snapshot() *are implemented with the algorithms described in the Figures 2 and 3, is linearizable.*

Proof The proof consists in associating with each operation op a single point of the time line (denoted $lp(\mathsf{op})$ and called its *linearization point*), such that

- $lp(\mathsf{op})$ is between the beginning (start event) of op and its end (end event),
- No two operations have the same linearization point,
- The sequence of the operations defined by their linearization points is a sequential execution of the snapshot object.

So the proof consists in an appropriate definition of the linearization points. The linearization point of each operation (except possibly the last operation of faulty processes) is defined as follows:

- An $\mathsf{update}(r, -, -)$ operation is linearized at the time of its embedded write of $REG[r]$ (line 01).
- The linearization of a snapshot operation $\mathsf{psp} = \mathsf{p\_snapshot}(< r_1, \cdots, r_x >)$ depends on the line at which its $\mathsf{return}()$ statement is executed.
  - Case 1: psp returns at line 08 due a successful double scan (psp is 0-helped). Its linearization point is any point of the time line between the first scan and the second scan of that successful double scan.
  - Case 2: psp returns at line 15, (psp is h-helped with $h \geq 1$). In that case, the values returned by psp have been computed by some update operation at line 21 or line 25. Moreover, whatever the case, they have been computed by a successful double scan executed by some process p_z. When considering this successful double scan, $lp(\mathsf{psp})$ is placed between the end of the first scan and the beginning of the second one.

It follows from the previous definition of the linearization points that each operation is linearized between its beginning and its end, and no two operations are linearized at the same point[7]. Moreover, it follows from the Lemmas 1, 2 and 3 concerning the well-definition, mutual consistency and freshness of the values returned by the partial snapshot operations, that the update and snapshot operations appear as if they had been executed according to the total order defined by their linearization points, which completes the proof of the lemma. $\Box_{Lemma\ 6}$

5.6 The partial snapshot object is correct, wait-free and efficient

Theorem 1 *The algorithms described in Figure 2 and Figure 3 satisfy the termination and consistency properties (stated in Section 3.1) that defines a partial snapshot object. Moreover, they satisfy the uptodateness and help-locality properties.*

Proof The proof is an immediate consequence of the lemmas 4, 5 and 6. $\Box_{Theorem\ 1}$

6 Discussion

6.1 On the cost of the $\mathsf{p\_snapshot}()$ and $\mathsf{update}()$ operations

Evaluating the “real” cost of the operations is difficult. This is mainly due to two reasons. The first one is the difficulty to figure out realistic patterns that address both (1) the concurrency among the $\mathsf{p\_snapshot}(R)$ and

[7] If two operations are linearized at the same point, they can be tie-broken according to the id of the processes that issued these operations.

$\mathsf{update}(r, -, -)$ operations, and, for each operation, (2) the content of its set R or component r. The second one is the difficulty to find a "good" measure. Evaluating only the number of shared memory accesses is a poor measure, as it takes into account neither the size of the values that are atomically read or written (that defines an "atomicity grain" of the corresponding read and write operations), nor the restrictions on asynchrony imposed by that "atomicity grain"[8].

Such a "real cost" analysis is beyond the scope of this paper (whose main concern is the investigation of new properties and new techniques for implementing snapshot objects). The following analysis, that only counts the number of shared memory accesses, is consequently limited and should not be considered as an ultimate cost criterion. It has to be enriched with other measures to become meaningful.

Cost of the p_snapshot() operation Let us first look at the number of accesses to the array REG. It follows from the proof of Lemma 5 that, in the worst case, a p_snapshot() operation executes n times the body of its **while** loop (lines 04-18). Consequently, it reads (at lines 03 and 05, Figure 2) $n+1$ times the x components $r_1, \cdots, r_x$ it is interested in. Hence, as far the array REG is concerned, there are at most $(n+1)x$ shared memory accesses. This is particularly interesting when $n < m$, i.e., when the number of processes is much smaller than the number of components (a case that occurs when a lot of critical variables are kept in the same snapshot object).

In the best case, a p_snapshot() operation reads only twice the x entries of the array REG in which it is interested. In these cases there are only $2x$ accesses of the array REG. Interestingly, this can appear in two distinct scenarios.

- The first scenario is when the p_snapshot() terminates due to a successful double scan that occurs during the first execution of the loop.
- The second scenario is when, during the first execution of the loop, the predicate of line 13 is true (while the one of line 06 is false). This occurs when a process p_w has updated two different components $r1$ and $r2$ such that $r1, r2 \in \{r_1, \cdots, r_x\}$, and these updates occurred between the scans of REG in aa and bb, respectively.

In addition to the accesses to the array REG, a p_snapshot() operation accesses once the array $ANNOUNCE$, and at most once the array $HELPSNAP$. It also invokes once Join() and Leave().

Cost of the update() operation An update() operation writes a single register $REG[r]$ and invokes once GetSet(). Then its number of shared memory accesses depends on the value $readers_i$ returned by GetSet() to the invoking process p_i.

If the set $readers_i$ is empty, there are no concurrent p_snapshot() operations, and update() terminates. Otherwise, let $\alpha = |readers_i|$. Then, p_i accesses α times the shared memory to read appropriate entries of the array $ANNOUNCE$. Let $\beta = |to\_help_i|$. If $\beta = 0$, there is no more shared memory accesses. If $\beta \neq 0$, the update() operation accesses β times the vector $HELPSNAP[i, -]$. The number of accesses to the array REG depends on the size of to_help_i. In the worst case there are $n(|to\_read_i|)$ accesses to the array REG.

6.2 Balancing the load: an active set per component

The p_snapshot() and update() algorithms presented in Figure 2 and Figure 3, respectively, use a single active set object. A direct consequence is the fact that, when a process p_i that executes the $\mathsf{update}(r, -, -)$

[8] Using large atomic registers that contain large values can be more costly than using several atomic registers of smaller size (e.g., $HELPSNAP[i, j]$ registers).

algorithm invokes $\mathsf{GetSet}()$, it obtains a set $readers_i$ made up of the ids of all the processes that are concurrently executing a $\mathsf{p\_snapshot}(R_\ell)$ operation, independently of the fact that r belongs or not to the corresponding sets R_ℓ. The process p_i has then to read all the registers $ANNOUNCE[j]$ for $j \in \mathsf{GetSet}()$ in order to learn which are the ones such that $r \in R_\ell$ (lines 03-06 in Figure 3). This places on each update operation a load (in number of shared memory accesses) that is due to the set of components read by the partial snapshot operations.

There is a simple way to balance this load by using an active set per component of the partial snapshot object. Let $AS[1..m]$ be the corresponding array of active set objects. We have the following modifications.

- The lines 01-03 of Figure 2 are replaced by the following lines:

```
$ANNOUNCE[i] \leftarrow < r_1, \cdots, r_x >$; $can\_help\_me_i \leftarrow \emptyset$;
for each $r \in \{r_1, \cdots, r_x\}$ do $AS[r].\mathsf{Join}()$; $aa[r] \leftarrow REG[r]$ end for.
```

- The lines 07 and 14 of Figure 2 are replaced by the following statement:

```
for each $r \in \{r_1, \cdots, r_x\}$ do $AS[r].\mathsf{Leave}()$ end for.
```

- The lines 02-06 of Figure 3 are replaced by the following ones:

```
$to\_help_i \leftarrow AS[r].\mathsf{GetSet}()$;
for each $j \in to\_help_i$ do $announce_i[j] \leftarrow ANNOUNCE[j]$ end for.
```

6.3 Using LL/SC registers instead of read/write atomic registers

An array of LL/SC registers This section shows that using LL/SC registers instead of the atomic read/write registers as underlying base registers reduces the number of base registers from $O(n^2)$ to $O(n)$. More precisely, the array $ANNOUNCE[1..n]$ and the matrix $HELPSNAP[1..n, 1..n]$ both made up of atomic read/write registers can be replaced by a single array $ANNHELP[1..n]$ such that each of its registers is accessed by the pair of LL/SC operations.

The LL/SC pair of operations An LL/SC register is an atomic register that provides the processes with two operations denoted $\mathsf{LL}()$ (Linked Load) and $\mathsf{SC}()$ (Store Conditional). Considering an LL/SC register X, $X.\mathsf{LL}()$ returns the current value of X. A conditional store $X.\mathsf{SC}()$ issued by a process p_i returns $true$ (the write succeeded) or $false$ (the write failed). Its success depends on the fact that, since the previous $X.\mathsf{LL}()$ issued by p_i, other processes have or have not updated X. It succeeds if and only if, since its last reading (whose value has been obtained by $X.\mathsf{LL}()$), X has not been written by another process p_j (whatever the value written by p_j, that value being possibly the same as the current value of X). If $X.\mathsf{SC}()$ is successful, p_i knows that X has not been updated since its last reading of X.

The input/output behavior of $X.\mathsf{LL}()$ and $X.\mathsf{SC}()$ can be precisely described by the following statements executed atomically (this description is from [24]). An array $valid_X[1..n]$ (initialized to $[false, \ldots, false]$) is associated with each LL/SC object X; $valid_X[i]$ is a flag set up by p_i when it issued $X.\mathsf{LL}()$ and set down by any process p_j when its write succeeds.

```
operation $X.\mathsf{LL}()$ issued by $p_i$: [$valid_X[i] \leftarrow true$; return($X$)].

operation $X.\mathsf{SC}(v)$ issued by $p_i$: [if $valid_X[i]$ then $X \leftarrow v$;
                                                  $\forall j : valid_X[j] \leftarrow false$;
                                                  return($true$)
                                             else return($false$)
                                 end if].
```

Weak variants of LL/SC registers are proposed in some architectures such as Alpha AXP (under the name ldl_l/stl_c), IBM PowerPC (under the name lwarx/stwcx), or ARM (under the name ldrex/strex) [17]. In these architectures, the entities using these base operations are the processors (and not the processes).

The computational power of the pair LL/SC operations in presence of process crash failures is the same as the one of the Compare&Swap operation, namely it has a consensus number equal to $+\infty$ [16].

The array $ANNHELP[1..n]$ The entry $ANNHELP[i]$ is used both by p_i and by any other process $p_j \neq p_i$ to pass information from one to the other (in both directions). It can contain three types of values, as described below.

- When it invokes p_snapshot(R), the process p_i sets $ANNHELP[i]$ to $< req, R >$ to announce that it wants to read atomically the components of R.
- When it returns from p_snapshot(R) without being helped (successful double scan), the process p_i sets $ANNHELP[i]$ to $\perp$ to prevent future help from any other process.
- When it helps p_i, a process p_j writes into $ANNHELP[i]$ the values corresponding to the components R that p_i wants to read.

So, $< req, R >$, $\perp$ and $< v_1, \dots, v_x >$ are the three types of values that $ANNHELP[i]$ can contain. Its initial value is $\perp$.

The LL/SC-based update() operation The LL/SC-based update() operation is described in Figure 5. It is exactly the same as Figure 3 except for 4 lines that are modified (their line number is postfixed with "M"). More precisely, we have the following. Let p_i be the process that executes update(r, v, i).

- Lines 04.M and 05.M. The process p_i now invokes $ANNHELP[j]$.LL() to learn the announce of each concurrent reader p_j. If $ANNHELP[j]$.LL() $\neq < req, - >$, then p_i does not have to help p_j. This is because p_j has already been helped by another process p_k (in that case $ANNHELP[j]$ contains a list of values) written by p_k at line 21.M), or p_j has terminated its p_snapshot(R) operation without being helped by another process (in that case, as we will see in the algorithm of the p_snapshot() operation, $ANNHELP[j] = \perp$).
- Line 21.M. As just indicated, this line is executed when p_j helps p_i. To that end, p_i executes $ANNHELP[j]$.SC$(< bb[r_1].value, \dots, bb[r_x].value >)$ to store into $ANNHELP[j]$ the help it gives to p_j. Due to the semantics of the pair LL/SC of each register $ANNHELP[j]$, the invocation $ANNHELP[j]$.SC() issued by p_i at line 21.M is successful only if (1) no other process has helped p_j since the last invocation $ANNHELP[j]$.LL() issued by p_i, and (2) p_j has not yet terminated its partial snapshot. This ensures that p_i does not overwrite a new announce by p_j.
- Line 25.M. Differently from the algorithm based on atomic read/write registers, where the help of p_i to p_j is individual and is consequently written in the entry (i, j) of the matrix $HELPSNAP$ (line 25 in Figure 3), the help of p_i to p_j is now written directly in $ANNHELP[j]$ at line 21.M. Consequently, if the test of line 24 is satisfied, another process p_w has already helped p_j, and p_i does not have to relay this help anymore. Hence, line 25.M is line 25 without the relay of the help to p_j.

The LL/SC-based p_snapshot() operation The LL/SC-based p_snapshot(R) operation is described in Figure 5. It is the exactly same as Figure 2 except for the two lines that are modified (their line number is postfixed with "M") and a new line that is added (its line number is postfixed by "A"). More precisely, we have the following. Let p_i be the process that executes p_snapshot(R). In order to make the presentation easier to follow, the line 15.M is first explained, then the line 07.A, and finally the line 01.M.

```
operation update($r, v, i$):    % (code for $p_i$) %
01    $nbw_i \leftarrow nbw_i + 1$; $REG[r] \leftarrow < v, i, nbw_i >$; % Lin Point %
02    $readers_i \leftarrow$ GetSet(); $to\_help_i \leftarrow \emptyset$;
03    for each $j \in readers_i$ do
04.M      $annhelp_i[j] \leftarrow ANNHELP[j]$.LL();
05.M      if $(annhelp_i[j] =< req, ann >) \wedge (r \in ann)$ then $announce_i[j] \leftarrow ann$; $to\_help_i \leftarrow to\_help_i \cup \{j\}$ end if
06    end for;
07    if $(to\_help_i = \emptyset)$ then return($ok$) end if;
08    $to\_read_i \leftarrow \left(\bigcup_{j \in to\_help_i} announce_i[j]\right)$ expressed as a sequence $< rr_1, \dots, rr_y >$;
09    for each $j \in to\_help_i$ do $can\_help_i[j] \leftarrow \emptyset$ end for;
10    for each $rr \in to\_read_i$ do $aa[rr] \leftarrow REG[rr]$ end for;
11    while $(to\_help_i \neq \emptyset)$ do
12        for each $rr \in to\_read_i$ do $bb[rr] \leftarrow REG[rr]$ end for;
13        $still\_to\_help_i \leftarrow \emptyset$;
14        for each $rr \in to\_read_i$ such that $aa[rr] \neq bb[rr]$ do
15            for each $j \in to\_help_i$ such that $rr \in announce_i[j]$ do
16                $still\_to\_help_i \leftarrow still\_to\_help_i \cup \{j\}$;
17                $can\_help_i[j] \leftarrow can\_help_i[j] \cup \{< w, sn >\}$ where $< -, w, sn >= bb[rr]$
18            end for
19        end for;
20        for each $j \in to\_help_i \backslash still\_to\_help_i$ do
21.M          $ANNHELP[j]$.SC($< bb[r_1].value, \dots, bb[r_x].value >$) where $< r_1, \dots, r_x >= announce_i[j]$
22        end for;
23        for each $j \in still\_to\_help_i$ do
24            if $(\exists < w, sn1 >, < w, sn2 > \in can\_help_i[j]$ such that $sn1 \neq sn2)$ then
25.M              $still\_to\_help_i \leftarrow still\_to\_help_i \backslash \{j\}$
26            end if
27        end for;
28        $to\_help_i \leftarrow still\_to\_help_i$; $to\_read_i \leftarrow \left(\bigcup_{j \in to\_help_i} announce_i[j]\right)$; $aa \leftarrow bb$
29    end while;
30    return($ok$)
```

Figure 5: A LL/SC-based update() operation

- Line 15.M. If p_i is helped by another process p_w, that process has deposited the helping values for R in $ANNHELP[i]$. Consequently, p_i returns these values for the components it has specified in R.
- Line 07.A. If p_i terminates its partial snapshot operation without being helped by another process (successful double scan), it strives to write $\perp$ into $ANNHELP[i]$. Let us observe that this write is successful if no process has deposited a helping value in $ANNHELP[i]$.

  In that way, when p_i terminates its partial snapshot operation (being helped at line 15.M, or without help at line 08), we necessarily have $ANNHELP[i] \neq < req, - >$ whatever the failure and asynchrony pattern. This means that, between the end of the current p_snapshot() invocation issued by p_i and the beginning of its next p_snapshot() invocation, no process p_i can help it (as for p_j to help p_i, p_j has to find $ANNHELP[i]$.LL() $=< req, - >$, as described at line 05.M in Figure 5).
- Line 01.M. When p_i calls p_snapshot(R), it first invokes $ANNHELP[i]$.SC($< req, R >$) to announce the components it wants to read. Let us observe that, due to the observation done in the previous item, the sequence LL/SC issued at line 01.M ensures that this conditional store is always successful.

```
operation p_snapshot($< r_1, \cdots, r_x >$):     % (code for $p_i$) %
01.M $ANNHELP[i]$.LL(); $ANNHELP[i]$.SC($< req, < r_1, \cdots, r_x >>$);
02    $can\_help\_me_i \leftarrow \emptyset$; Join();
03    for each $r \in \{r_1, \cdots, r_x\}$ do $aa[r] \leftarrow REG[r]$ end for;
04    while $true$ do % Lin point if return at line 08 %
05       for each $r \in \{r_1, \cdots, r_x\}$ do $bb[r] \leftarrow REG[r]$ end for;
06       if ($\forall r \in \{r_1, \cdots, r_x\}$ : $aa[r] = bb[r]$) then
07          Leave();
07.A        if $ANNHELP[i]$.LL() $=< req, - >$) then $ANNHELP[i]$.SC($\perp$) end if;
08          return($< bb[r_1].value, \cdots, bb[r_x].value >$)
09       end if;
10       for each $r \in \{r_1, \cdots, r_x\}$ such that $(aa[r] \neq bb[r])$ do
11          $can\_help\_me_i \leftarrow can\_help\_me_i \cup \{< w, sn >\}$ where $< -, w, sn >= bb[r]$
12       end for;
13       if $\big(\exists < w, sn1 >, < w, sn2 > \in can\_help\_me_i$ such that $sn1 \neq sn2\big)$ then
14          Leave();
15.M        return($ANNHELP[i]$)
16       end if;
17       $aa \leftarrow bb$
18    end while.
```

Figure 6: A LL/SC-based p_snapshot() operation

7 Conclusion

The concept of shared memory snapshot object has first been proposed in [1]. The notion of partial snapshot object has then been introduced in [8]. The present paper has first proposed two efficiency properties related to the implementation of partial snapshot objects. It has then addressed the design of a partial snapshot object whose implementation meets these properties. To attain this goal, the proposed implementation takes into account the current concurrency pattern and strives to be as efficient as possible. Its main features are the following[9].

1. The proposed algorithm is the first that (to our knowledge) relies on the "write first, help later and help individually" strategy.
2. An update operation helps a snapshot operation only if needed, and no more. More explicitly, an update helps only the concurrent snapshots that read the component it writes. This new property, called *help-locality* states "no help when no conflict".
3. Differently from the update algorithms proposed so far, an update provides the processes with *more asynchrony*: the write of a new value and the writes of helping snapshot values are separated writes.
4. The update operation satisfies the following *uptodateness* property: the helping values (if any) supplied by an update$(r, v, -)$ operation includes, for the component r, the value v (written by that update) or a more recent value.
5. The number of underlying base atomic registers can be reduced from $O(n^2)$ to $O(n)$ when these registers can be accessed by the LL/SC pair of operations instead of the weaker read/write pair of base operations.

[9] These features are also the main differences with the partial snapshot algorithm (based on read/write atomic registers) presented in [8], and the algorithms (based on read/write atomic registers) that implement a classical snapshot object (e.g., [1]).

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Afek Y., Stupp G., Touitou D., Long-lived Adaptive Collect with Applications. *Proc. 40th IEEE Symposium on Foundations of Computer Science Computing (FOCS'99)*, IEEE Computer Press, pp. 262-272, 1999.

[3] Aguilera M.K., A Pleasant Stroll Through the Land of Infintely Many Creatures. *ACM SIGACT-NEWS*, 35(2):36-59, 2004.

[4] Anderson J., Multi-writer Composite Registers. *Distributed Computing*, 7(4):175-195, 1994.

[5] Attiya H., Needed: Foundations for Transactional Memory. *ACM SIGACT News*, 39(1):59-61, 2008.

[6] Attiya H. and Fouren A., Algorithms Adapting to Contention Point. *Journal of the ACM*, 50(4):444-468, 2003.

[7] Attiya H., Fouren A. and Gafni E., An Adaptive Collect Algorithm with Applications. *Distributed Computing*, 15:87-96, 2002.

[8] Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, pp. 336-343, 2008.

[9] Attiya H. and Rachman O., Atomic Snapshot in $O(n \log n)$ Operations. *SIAM Journal of Computing*, 27(2):319-340, 1998.

[10] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, (2d Edition), Wiley-Interscience, 414 pages, 2004.

[11] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[12] Ellen F., How Hard Is It to Take a Snapshot? *Proc. 31th Conference on Current Trends in Theory and Practice of Computer Science (SOFTSEM'05)*, Springer-Verlag #3381, pp. 28-37, 2005.

[13] Felber P., Fetzer C., Guerraoui R. and Harris T., Transactions Are Back–But Are They the Same? *ACM SIGACT News*, 39(1): 47-58, 2008.

[14] Guerraoui R. and Kapałka M., On the Correctness of Transactional Memory. *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, ACM Press, pp. 175-184, 2008.

[15] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of Atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS #4671, pp. 47-61, 2007.

[16] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[17] Herlihy M.P. and Shavit N., The Art of Mutiprocessor Programming. *Morgan Kaufmann Pub.*, San Francisco (CA), 508 pages, 2008.

[18] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News*, 39(1): 62-72, 2008.

[19] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[20] Imbs D. and Raynal M., A Lock-based STM Protocol that Satisfies Opacity and Progressiveness. *Proc. 12th Int'l Conference On Principles Of Distributed Systems (OPODIS'08)*, Springer-Verlag LNCS #5401, pp. 226-245, 2008.

[21] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear Time Snapshots Using Multi-writer Multi-reader Registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer-Verlag #857, pp. 130-140, 1994.

[22] Jayanti P., An Optimal Multiwriter Snapshot Algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOCS'05)*, ACM Press, pp. 723-732, 2005.

[23] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[24] Moir M., Practical Implementation of Non-Blocking Synchronization Primitives. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 219-228, 1997.

[25] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.

[26] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

A Full snapshot based on the "write first, help later" strategy

The snapshot() and update() algorithms described in Figure 7 consider the case where each snapshot operation reads all the components. Moreover, an update systematically computes a helping snapshot value (i.e., whatever the concurrency pattern). These algorithms provide a basic implementation of a multi-writer/multi-reader snapshot object. They can be seen as a simplified (and not efficient) version of the algorithms presented in the body of the paper (Figures 2 and 3), that is the "write first, help later" counterpart of the classical snapshot algorithms presented in [1].

```
operation snapshot():   % (code for $p_i$) %
  $can\_help_i \leftarrow \emptyset$;
  for each $r \in \{1, \cdots, m\}$ do $aa[r] \leftarrow REG[r]$ end for;
  while $(true)$ do
      for each $r \in \{1, \cdots, m\}$ do $bb[r] \leftarrow REG[r]$ end for;
      if $(\forall r \in \{1, \cdots, m\} : aa[r] = bb[r])$ then return$(bb[1].value, \cdots, bb[m].value)$ end if;
      for each $r \in \{1, \cdots, m\}$ such that $bb[r] \neq aa[r]$ do
          $can\_help_i \leftarrow can\_help_i \cup \{< w, sn >\}$ where $< -, w, sn >= bb[r]$
      end for;
      if $(\exists < w, sn1 >, < w, sn2 > \in can\_help_i$ such that $sn1 \neq sn2)$ then
          return$(HELPSNAP[w])$
      end if;
      $aa \leftarrow bb$
  end while.
=================================================================
operation update$(r, v)$:   % (code for $p_i$) %
  $sn_i \leftarrow sn_i + 1$; $REG[r] \leftarrow < v, i, sn_i >$;
  $HELPSNAP[i] \leftarrow$ snapshot();
  return$(ok)$.
```

Figure 7: Algorithms for update and full snapshot