Adaptive Register-Allocation with a Linear Number of Registers

Carole Delporte-Gallet[1], Hugues Fauconnier[1], Eli Gafni[2], and Leslie Lamport[3]

[1] U. Paris Diderot, France
[2] Computer Science Department, UCLA, USA
[3] Microsoft Research

Abstract. We give an adaptive algorithm in which processes use multi-writer multi-reader registers to acquire exclusive write access to their own single-writer, multi-reader registers. It is the first such algorithm that uses a number of registers linear in the number of participating processes. Previous adaptive algorithms require at least $\Theta(n^{3/2})$ registers.

Keywords: shared memory, read/write registers, distributed algorithms, wait-free, space complexity, renaming.

1 Introduction

One way to implement multiprocess synchronization is by providing each process with a single-writer, multi-reader atomic register (SWMR) that it can write and other processes can read. We present an adaptive algorithm to implement such a system of registers with an array of multi-writer multi-reader atomic (MWMR) registers whose length is linear in the number of participating processes. The algorithm is non-blocking unless an unbounded number of processes initiate operations.

An adaptive algorithm, also called a uniform algorithm [13], is one that does not know the number of potentially participating processes. Equivalently, it is an algorithm whose cost is a function not of the total number of processes but of the number of processes that actually participate in the algorithm. For the SWMR registers, this is the number of processes that actually perform a read or write operation. Our goal is to minimize the number of MWMR registers, and our algorithm uses a number that is linear in the number of participants. No *a priori* bound on this number is assumed.

Why do we find this algorithm interesting? There are simpler algorithms that assume stronger communication primitives—for example, test and set registers—but MWMR registers are the weakest ones for which we know that an adaptive algorithm is possible. More efficient randomized algorithms are possible, but our algorithm is always correct, not just correct with high probability. There is a trivial way to implement a collection of SWMR registers with an array C of MWMR registers. The i^{th} process simply uses $C[i]$ as its register. Of course, this algorithm uses an unbounded number of registers. The obvious way to make the number of registers linear in the number of participating processes is by having the processes first execute an adaptive renaming algorithm [7, 10] in which each participating process is assigned a unique number from 0 to M for some M that depends linearly on the number of participants. A process assigned the number j then uses $C[j]$ as its register. However, we know of few renaming algorithms that do not assume a collection of SWMR registers already allocated to processes [6, 8, 19]. Those algorithms are all based on the grid-network of "splitters" proposed by of Anderson and Moir [19]. Of these, the more space-efficient is an improvement of Aspnes [6] that requires $\Theta(k^{3/2})$ MWMR registers for k participating processes. Even though the

renaming algorithm is used only to determine the assignment of processes to elements of the array C, the values in those $\Theta(k^{3/2})$ registers must be maintained forever because additional processes may enter the system at any time. (Reclaiming the space requires knowing an *a priori* bound on the number of processors that might participate.) Thus, our algorithm is the first that implements a collection of SWMR registers with $O(k)$ MWMR registers.

Almost all previous methods for making an algorithm adaptive start by using one of several renaming algorithms [2–4, 7, 10]. It has generally been assumed that this is the only way to implement an adaptive algorithm [9]. Based on an idea in [11], our implementation avoids the use of a renaming algorithm to begin reliable communication. Instead, participating processes first announce their presence by using a non-blocking one-shot limited-snapshot algorithm that we call the GFX (Generalized Fast eXclusion) protocol, which can be viewed as generalizing [16] from 1-concurrency to k-concurrency. The snapshot is limited to having the property that two snapshots of the same size coincide. It need not ensure that snapshots of different sizes are related by containment. To perform a read or write operation to a register, a process first reads the posted snapshots to find the number n of participants that have announced their presence, and it executes an algorithm [11] that assumes at most n processes. It then reads the number of participants again, finishing the operation if that number still equals n. Otherwise, the process repeats the n-process algorithm for the new value of n. While we use this approach to implement renaming, it can be used to provide an adaptive implementation of any task.

By using our adaptive algorithm for implementing a collection of SWMR registers, we can solve any task under the assumption of finite arrival [14]. In particular, using existing algorithms, we can implement adaptive renaming with a linear range [7, 10]. This in turn allows us to allocate unique registers to processes with a number of registers linear in the number of participants. With register allocation, we can implement a collection of SWMR registers with wait-free read and write operations rather than just non-blocking ones. For many tasks of high read-write complexity, doing renaming first may reduce the step complexity of an adaptive algorithm.

We ignore time complexity—the number of steps taken by the algorithm. Our algorithm is executed just once, to assign SWMR registers to processes; it adds nothing to the cost of using those registers. Since space used by an adaptive algorithm cannot be reclaimed, it is perhaps more important than time complexity. Still, optimal time complexity is an interesting problem that remains unsolved.

In the non-adaptive case, it has been shown that at least n registers are required to implement n SWMR registers [11], so the linear number of registers used by our algorithms is optimal up to a constant factor. We originally believed that adaptive algorithms required more than a linear number of registers, and we tried to derive such a lower bound on the number of registers, independent of their size. When the difficulty is caused by processes stepping on each other because of the lack of *a priori* coordination, size of the registers is not a factor. (See the lower bound for consensus [12].) We were therefore surprised to discover our algorithm.

Section 2 describes our implementation and sketches an informal proof of its correctness. (Some might call this sketch a proof.) In Section 3, the two key algorithms used in the implementation are precisely described in the PlusCal algorithm language [18]. The section also describes formal TLA$^+$ correctness proofs of the safety properties of these algorithms. The complete mechanically-checked proofs are available on the Web [15].

2 An Informal Proof of the Algorithm

A sequence of SWMR registers is easily implemented using an algorithm we call *SnapShot*. We obtain this algorithm via two intermediate algorithms: the Leaky Repository Protocol and Algorithm *GFX*. We give here informal proofs of these algorithms; formal proofs of algorithms *GFX* and *SnapShot* are described in Section 3.

2.1 Preliminaries

Our algorithms assumes a small constant number of infinite arrays of MRMW registers, indexed by natural numbers, all registers containing the same initial value that we take to be $\{\}$ (the empty set). The algorithms write into only the first k elements of the arrays, where k is a linear function of the number of participating processes. Hence, they can be implemented by finite arrays, given a bound on the number of possible participants.

Since we are interested only in space complexity, for simplicity we never read a single array entry; we always atomically read the entire array, using the double scan method of [2]. To allow scanning an infinite array A, we use an auxiliary infinite array $\overline{A}$, where a process writes $A[i]$ by first writing some value other than $\{\}$ into $\overline{A}[0], \ldots, \overline{A}[i]$. A scan of A can assume $A[i] = \{\}$ for all $i \geq j$ if $\overline{A}[j] = \{\}$.

2.2 The Leaky Repository

The Leaky Repository Protocol maintains a repository of facts using an infinite array A of MWMR registers, where the value of a register can be any finite set of facts. At any time, the contents of the repository is the set $A[0] \cup A[1] \cup \ldots$ of facts, which can be obtained by atomically reading the array A. The repository is leaky because facts stored in it may be lost. We would like a process to be able to add a facts to the repository and have them remain there forever, but that is hard to do. Instead, we describe a protocol that tries to do this. It doesn't succeed, but it does provide a property that makes it a useful building block for the *GFX* and *SnapShot* algorithms.

Here is how process p tries to add a set F of facts to the repository. To try to avoid destroying previously added facts, p writes to a register only by performing a read-then-write operation that first atomically reads the entire array A and then writes the facts in F together with all the other facts it has ever read or written. To try to keep the facts in F from being overwritten by other processes, p performs such read-then-write operations to put the facts in F into multiple registers. To use as little of the array A as possible, p writes into the first n registers of A, for some n that it hopes is large enough.

Process p hopes that, if an atomic read of A shows the facts in F in each of the first n registers of A, then that ensures they will remain in the repository forever. Of course, it doesn't—the repository is leaky. Here's what can go wrong. Suppose that there are n processes other than p, each of which has performed the read of a read-then-write operation to a different one of the first n registers and is about to do the write. Process p can then perform read-then-write operations to the first n registers and read A to find that those registers all contain the facts in F. The n other processes can then perform their writes, destroying all traces of the facts in F. Before the n^{th} process writes, the contents of the repository satisfies:

R1. It contains all the facts in F.

This property is falsified by the n^{th} process's write. Each register i then contains a set F_i of facts written by a different process p_i. Moreover, each of the those n read-then-write operations was begun before p's final read of A. Therefore, the contents of the repository at that moment satisfies the following property, where R is the read of A by p that found the facts of F in all those n registers.

R2. It contains all the facts in $F_0 \cup \ldots \cup F_{n-1}$, for sets F_i such that there are n distinct processes $p_0, \ldots, p_{n-1}$ different from p, where each p_i wrote F_i with a read-then-write operation that began before R.

We now generalize from this scenario. Note that R1 and R2 assert properties (that may be true or false) of an arbitrary read R of the repository by a process p that obtains a set F

of facts. R2 asserts the existence of some sets F_i and processes p_i, not the ones from any particular scenario. Note also that if a set S of facts satisfies R1 or R2, then any superset of S also satisfies R1 or R2. We will prove the following:

> **Property R** If a read R of A by process p finds $F \subseteq A[i]$ for $i = 0, \ldots, n-1$, then at all times after that read, the contents of the array A satisfies R1 or R2.

Property R allows R2 to be satisfied with different sets F_i and processes p_i at different times, and it allows R1 to become true again after it has become false. If there are at most n participants when p performs R, then R2 can never be true, so the facts in F must remain in the repository forever.

Property R is true of any protocol in which a process writes to a register of A using only a read-then-write operation that first reads A and then writes all the facts it has ever read from or written to A (perhaps writing additional facts too). We say that any such algorithm obeys the Leaky Repository Protocol for repository A.

We could use the Leaky Repository Protocol in an obvious way to implement an *add* F operation that always satisfies R1 or R2 after it has completed. However, we instead implement an *add* & *read* f operation that adds a single fact f to the repository and returns a set F such that F is the contents of the repository when the operation completes and thereafter always satisfies R1 or R2, for some "suitable" n. What n is suitable varies with the application, and it may depends on the *add* & *read* operation and on F. To perform an *add* & *read* f operation, process p executes the Leaky Repository Protocol to keep writing f in registers. The operation completes and returns the set F of facts when a read of A finds that F is the contents of the repository and $A[0] = \cdots = A[n-1] = F$ for some suitable n.

The *add* & *read* operation is used by Algorithm *SnapShot* with the "suitable" value of n being the number of participants. In that case, R2 cannot be true, so the set of facts returned by the *add* & *read* remain in the repository forever. To determine the number of participants, *SnapShot* uses Algorithm *GFX*, which uses *add* & *read* operations in which the "suitable" value of n is one plus the number of facts in the repository. Property R then implies that if the facts that a process read from the repository are no longer all there, then facts added by $n+1$ other processes are.

A process p's *add* & *read* operation need never complete. It can forever keep doing read-then-write operations if other processes keep performing *add* & *read* operations that add new facts. However, with a bounded number of participating processes and a bound on the number of registers that each operation writes, the entire collection of *add* & *read* operations is non-blocking—meaning that if some process is performing an *add* & *read* operation then some *add* & *read* operation will eventually complete. To prove this, we suppose that some set of processes is forever trying to perform *add* & *read* operations, none of which complete, and we obtain a contradiction. Since each process writes non-decreasing sets of facts and there are only a finite number of facts being added, eventually each process p forever reads only a fixed set F_p of facts and keeps writing F_p. If all the sets F_p are the same, every process will write only that set. Since there is a bound on the number of registers that an operation writes, this implies that all the operations will finish. If all the F_p are not the same, choose a minimal set F_q. Since q's operation doesn't finish, it must eventually read a set F_r different from F_q. Minimality of F_q implies that F_r contains a fact not in F_q, contradicting the assumption that q reads only facts in F_q. Hence, the algorithm is non-blocking.

We now prove Property R. We must show that R1 or R2 holds forever after the read R of p finds $F \subseteq A[i]$ for all $i < n$. Define $W(i)$ to be the set (whose elements are sets of facts) that contains every set of facts that some process is about to write into $A[i]$, having completed the read of A in a read-then-write operation. Let $W0(i)$ be the value of $W(i)$ when p performs

R. We show that the following invariant is true upon completion of p's read R and is left true by every further step of the algorithm:

> For all $i < n$, the value of $A[i]$ and every element of $W(i)$ contains (as a subset) either F or an element of $W0(i)$.

The invariant is true upon completion of R because then $W0(i) = W(i)$ and $F \subseteq A[i]$ for all $i < n$. A step that writes a value from $W(i)$ into $A[i]$ obviously cannot falsify the invariant. A step that adds a value to $W(i)$ cannot falsify the invariant because the value being added to $W(i)$ contains all the facts obtained by reading the repository after read R, which includes the value of $A[i]$. This completes the proof of invariance. The invariant implies that the contents of the repository satisfies R1 or R2, since either (i) some $A[i]$ contains F, so R1 holds, or else (ii) each $A[i]$ with $i < n$ contains an element of $W0(i)$, which by definition of $W0(i)$ implies that the union of the $A[i]$ satisfies R2. This proves Property R.

2.3 Algorithm *GFX*

Algorithm *GFX* is a one-shot algorithm, meaning that it is executed at most once by any process. It solves the following weaker version of the snapshot task [2]: A process p that executes the algorithm must return a set F_p of participants such that

- $p \in F_p$ for any p.
- $|F_p| = |F_q|$ implies $F_p = F_q$ for any p and q, where $|F|$ is the cardinality of the set F.

To implement the algorithm, we use the Leaky Repository Protocol with a single infinite array $A1$, where the repository's facts are (names of) processes. A process p executes the *GFX* algorithm by executing an *add* & *read* p operation that completes and returns a set of facts/processes F until it reads $A1[0] = \cdots = A1[|F|] = F$. Thus, the suitable n for this *add* & *read* operation is $1 + |F|$, where F is the set of facts being returned.

Now suppose a process p's execution of the *GFX* algorithm completes and returns the value F. Every write by a process q writes the fact/process q. Property R therefore implies that after the read by p that completes its execution of the *GFX* algorithm, the repository $A1$ forever contains either (by R1) all the processes in F or (by R2) $|F| + 1$ distinct processes. Any execution of the *GFX* algorithm that then completes cannot return a set $G \neq F$ of facts with $|G| = |F|$. This proves that the *GFX* algorithm satisfies its required properties.

Each execution of the *GFX* algorithm is an execution of an *add* & *read* operation for the leaky repository that writes a number of registers at most one greater than the total number of participants. The algorithm is therefore non-blocking if there is a bounded set of participants. In a non-blocking one-shot algorithm with a finite set of participants, every execution of the algorithm by a participant completes.

2.4 Algorithm *SnapShot*

Algorithm *SnapShot* implements a non-leaky repository that provides an add-and-read operation we call *snap* f that atomically adds the single fact f and returns the new contents of the repository. More precisely, in addition to the obvious properties that *snap* f adds fact f and returns only facts that have been added, the algorithm satisfies the property that if a *snap* operation op_p by process p returns set F_p and a *snap* operation op_q by process q returns F_q, then:

- $F_p \subseteq F_q$ or $F_q \subseteq F_p$.
- If op_p finishes before op_q starts, then $F_p \subseteq F_q$.

The idea of the *SnapShot* algorithm is to use the Leaky Repository Protocol on an array $A3$, and to implement a *snap* f operation by an *add* & *read* f operation to the repository, where the "sufficient" number n of registers is greater than the total number of participants. Property R then implies that if the *add* & *read* f operation succeeds, the value returned remains forever in the repository (because R2 cannot hold).

Let's suppose that that there is a *count* operation that a process p can call to learn the number of participants that can be executing a *snap* operation. To perform a *snap* f operation, a process p first executes *count* to obtain a bound n on the number of participants. It then executes the Leaky Repository Protocol to add f to the repository, writing in the first n registers of $A3$. If a read of the repository obtains a value F such that $A[0] = \cdots = A[n-1] = F$, process p executes the *count* operation again. If that execution returns the same number n of participants, then the *snap* f operation completes and returns the value F. Otherwise, the process continues the procedure, replacing n with the new value returned by *count*.

If a *snap* f operation by process p completes and returns the set F of facts, Property R holds for the final read of the repository that obtains F. Since F was in n registers and the read occurred when there were at most n participants, R2 cannot hold. Hence R1 holds forever, so F remains forever in the repository. Every *snap* operation that completes after p's *snap* f operation sees the facts in F and therefore returns a set G with $F \subseteq G$. This implies that the *SnapShot* algorithm satisfies its requirement.

We still have to implement the *count* operation. We do that by using algorithm *GFX* and a second array $A2$ of registers. When a participant p arrives, before performing any *snap* operation it (i) executes *GFX* to obtain a set S of participants, which includes itself, and (ii) writes (the processes in) S in $A2[|S|-1]$. The correctness property of *GFX* implies that no other value can ever be written in $A2[|S|-1]$. Since the processes written in $A2$ are all participants and every participant is written in $A2$, the set of all processes in $A2$ includes all participants that can write to $A3$. The *count* operation is then performed by reading $A2$ and counting the number of (distinct) processes read.

A *snap* operation executes a leaky repository's *add* & *read* operations that write a number of registers at most equal to the number of participating processes. Therefore, if there are a bounded number of participants, then the *SnapShot* algorithm is non-blocking.

2.5 Implementing the SWMR Registers

Using algorithm *SnapShot*, the collection of SWMR registers is implemented as follows. To write x as the i^{th} write to its (simulated) SWMR register, a process p performs the operation *snap* $\langle p, i, x\rangle$, ignoring the value returned by the *snap* operation. To atomically read all processes' SWMR registers, a process executes a *snap* $\perp$ operation for a special fact $\perp$. (Algorithm *SnapShot* allows multiple *snap* f operations with the same fact f.) The current value of process q's register is the value x in the triple $\langle q, i, x\rangle$ with the largest value of i in the set returned by the *snap* operation. If no such triple exists, then q has not yet written to its SWMR register. It follows easily from the properties of the *SnapShot* algorithm that this implements a collection of SWMR registers with an atomic operation that reads all the registers.

3 The Formal Proofs

We believe that our implementation of SWMR registers from algorithm *SnapShot* is obvious enough that a precise description of it and a formal proof of its correctness are not necessary. However, algorithms *GFX* and *SnapShot* are subtle. In this section, we precisely describe these algorithms in the PlusCal algorithm language [18]. PlusCal constructs whose meanings may not be obvious are briefly explained as they are introduced. A PlusCal expression can

be any TLA$^+$ formula [17], and a PlusCal algorithm is automatically translated to a TLA$^+$ specification that defines the algorithm's formal meaning.

We have written formal, mechanically-checked TLA$^+$ correctness proofs of the safety properties of the *GFX* and *SnapShot* algorithms. Those proofs are sketched here; the complete proofs are available on the Web [15]. Unlike the informal proofs of Section 2, which use behavioral reasoning, the formal proofs use purely assertional reasoning. They are therefore not a direct formalization of the informal proofs.

Algorithms *GFX* and *SnapShot* are written in terms of the set *Proc* of all processes that eventually participate. We assume that this set is finite (otherwise the algorithms would not be non-blocking). Processes that perform no actions are not represented in our specifications. Since processes do not use the value of *Proc*, our algorithm does not assume any *a priori* knowledge of the number of participating processes.

3.1 Algorithm *GFX*

The Specification

The specification of what algorithm *GFX* is supposed to do is given by algorithm *GFXSpec* of Figure 1. The **variable** statement declares the global variable *result* and initializes it to be

```
--algorithm GFXSpec
{ variable result = [p ∈ Proc ↦ {}]
  process(Pr ∈ Proc)
    { A: with (P ∈ {Q ∈ SUBSET Proc :
                      ∧ self ∈ Q
                      ∧ ∀p ∈ Proc \ {self} :
                          ∨ Cardinality(result[p]) ≠ Cardinality(Q)
                          ∨ Q = result[p]
                   } )
           {result[self] := P}
    }
}
```

Fig. 1. Specification of Algorithm *GFX*.

an array indexed by the set *Proc* of processes, with *result*[*p*] initially the empty set {} for each process *p*. The **process** statement declares there to be one process for each element of *Proc*, the statement's body giving the code for process *self*. The statement **with** $(x \in S)\{\Sigma\}$ executes Σ with an arbitrary element of S substituted for x. The expression SUBSET *Proc* denotes the set of all subsets of *Proc*. TLA$^+$ allows conjunctions and disjunctions to be represented as lists of formulas bulleted with $\wedge$ or $\vee$, using indentation to eliminate parentheses. (This notation makes large formulas easier to read.)

In PlusCal, an atomic action is the execution of code from one label to the next, where there is an implicit label *Done* at the end. Thus, the entire body of the process is executed as a single atomic action *A* (named by the label). The **with** statement sets *result*[*self*] to *P*, which is an arbitrarily chosen element *Q* in the set of subsets of *Proc* such that (i) *self* is in *Q* and (ii) for each other process *p*, either the cardinality of *Q* is unequal to the cardinality of *result*[*p*], or else *Q* equals *result*[*p*]. Thus, a process *p* that does not execute its *A* action has *result*[*p*] always equal to the empty set. A process *p* that executes its *A* action terminates with *result*[*p*] equal to a set of processes containing *p* such that for any other process *q*, either *result*[*p*] and *result*[*q*] have different cardinalities, or *result*[*p*] = *result*[*q*]. The TLA$^+$ translation of the

algorithm introduces a variable pc, where $pc[p]$ equals the label at which control is in process p, so $pc[p]$ equals either the string "A" or the string "Done".

The Algorithm

Algorithm *GFX* is described in Figure 2. The variables *known* and *notKnown* are local to

```
--algorithm GFX
{ variables A1 = [i ∈ Nat ↦ {}], result = [p ∈ Proc ↦ {}] ;
  process (Pr ∈ Proc)
    variables known = {self}, notKnown = {} ;
    { a: known := known ∪ NUnion(A1) ;
         notKnown := {i ∈ 0 .. (Cardinality(known)) : known ≠ A1[i]} ;
         if (notKnown ≠ {})
           { b : with (i ∈ notKnown) {A1[i] := known} ;
                 goto a
           }
         else {result[self] := known} ;
    }
}
```

Fig. 2. Algorithm *GFX*.

self (the current process) and cannot be read or written by other processes. Variable *known* stores the set of processes known to process *self*, and *unKnown* stores a set of array indices (natural numbers). In the TLA$^+$ translation, their values are arrays indexed by the set *Proc*. The other new notation used in this algorithm is: *Nat* is the set of natural numbers, $i \,..\, j$ is the set of integers k with $i \leq k \leq j$, and the operator *NUnion* is defined (in the TLA$^+$ module containing the algorithm) by

$$NUnion(A) \triangleq \text{UNION}\{A[i] : i \in Nat\}$$

where the UNION expression is commonly written by mathematicians as $\bigcup_{i \in Nat} A[i]$. Evaluation of that expression is implemented by atomically reading the array A. Observe that although *result* is a global variable, $result[p]$ is accessed only by process p.

There are two atomic actions that a process p can execute. Action a sets $known[p]$ and $notKnown[p]$, executes the **if** test, and then either goes to label b or else executes the **else** clause, setting $result[p]$, and terminates. Action b writes to one element of $A1$ and goes to label a.

Safety The safety property satisfied by the *GFX* algorithm is that it implements algorithm *GFXSpec* under the refinement mapping [1] that substitutes expressions of *GFX*'s variables for the variables of *GFXSpec* as follows:

$$\begin{aligned} &result \leftarrow result \\ &pc \quad\; \leftarrow [p \in Proc \mapsto \text{IF } pc[p] = \text{"Done"} \text{ THEN "Done" ELSE "A"}] \end{aligned}$$

Implementation under this refinement mapping means that in any execution of algorithm *GFX*, the sequence of values assumed by the substituting expressions is ones that algorithm *GFXSpec* allows for its variables.

This safety property is a fairly direct consequence of the invariance of the assertion *GFXCorrect* defined as follows. Let two sets of processes be *compatible* iff they are either

equal or have different cardinality. We define *GFXCorrect* to assert that, for any processes p and q of *Proc*, if p and q have terminated then $result[p]$ and $result[q]$ are compatible.

To understand why *GFXCorrect* is an invariant of algorithm *GFX*, observe that process p terminates and sets $result[p]$ to $known[p]$ after using the *GFX* protocol to write $known[p]$ into registers $A1[0], \ldots, A1[Cardinality(known[p])]$. Any process q that reads $known[p]$ will set $known[q]$ to be a superset of that value, so $known[p]$ and $known[q]$ are compatible because the definition of compatibility implies that two sets are compatible if one is a superset of the other. If no process reads the value $known[p]$, then $Cardinality(known[p]) + 1$ processes must have written their *known* values into $A1$. Since $known[r]$ contains r, for each process r, the union of all $A1[i]$ therefore has cardinality greater than that of $known[p]$, and any process q that then terminates will do so with $result[q]$ having cardinality greater than $known[p]$.

To make this reasoning completely rigorous requires an inductive invariance proof [5]. Define $PA1$ to be the set of *potential* values of the array $A1$, meaning the values that $A1$ could have after some subset of the processes at control location b execute their b action. The key part of the inductive invariant is:

$$\begin{array}{l} \forall p \in Proc,\, P \in PA1 : \\ \quad \lor\ Cardinality(known[p]) < Cardinality(NUnion(P)) \\ \quad \lor\ known[p] \subseteq NUnion(P) \end{array}$$

A machine-checked formal proof of safety is available on the Web [15].

Liveness The algorithm is non-blocking, meaning that if some process in *Proc* keeps taking steps, then some process in *Proc* eventually terminates—even if other processes stop before they terminate. The proof that *GFX* is non-blocking is by contradiction. Assume a non-empty set Π of nonterminating processes in *Proc* that keep taking steps. For each process p in *Proc*, elements are never removed from the set $known[p]$, and $known[p]$ is a subset of the finite set *Proc*. Hence, eventually the values of $known[p]$ remain unchanged for all p in *Proc*. We consider the execution from that point on. Choose p in Π so that $known[p]$ is minimal, meaning that $known[q]$ is not a proper subset of $known[p]$ for any q in Π. Since p has not terminated and keeps taking steps, it must perform an infinite number of writes to registers $A[i]$ with $0 \leq i \leq Cardinality(known[p])$. Hence, it must perform an infinite number of writes to $A[j]$ for some j. To do that, p must infinitely often read $A[j]$ to be different from the value $known[p]$ that p writes to $A[j]$. At least one of those values of $A[j]$ must equal $known[q]$ for some q in Π with $q \neq p$. Since $known[p]$ does not change, this value of $known[q]$ must be a subset of $known[p]$; and since $known[q]$ is unequal to $known[p]$, it must be a proper subset of $known[p]$. This contradicts the minimality of $known[p]$.

Space complexity. A process p writes only to array elements $A1[i]$ with $i \leq Cardinality(known[p])$. Since $known[p]$ is a subset of the set *Proc* of participating processes, this implies that no register $A1[i]$ with $i > Cardinality(Proc)$ is ever written. Therefore, algorithm *GXF* uses at most $Cardinality(Proc) + 1$ registers.

3.2 Algorithm *SnapShot*

The Specification

The *SnapShot* algorithm maintains a set S of values that is initially empty. It provides a *snap* operation whose argument is a value v. Executing $snap(v)$ atomically adds v to S and returns the current value of S. Algorithm *SnapSpec* is specified in Figure 3. The only additional PlusCal construct it introduces is **either**, where the statement **either** Σ_1 **or** Σ_2 is executed by nondeterministically choosing either Σ_1 or Σ_2 and executing it.

```
--algorithm SnapSpec
{ variables myVals = [i ∈ Proc ↦ {}], nextout = [i ∈ Proc ↦ {}] ;
  process (Pr ∈ Proc)
    variable out = {} ;
    { A: while (TRUE)
         {     with (v ∈ Val) { myVals[self] := myVals[self] ∪ {v} } ;
           B: with (V ∈ {W ∈ SUBSET PUnion(myVals) :
                                ∧ myVals[self] ⊆ W
                                ∧ PUnion(nextout) ⊆ W} )
                { nextout[self] := V } ;
           C: either out := nextout[self]
              or     goto B ;
         }
    }
}
```

Fig. 3. Algorithm *SnapSpec*, the specification of *SnapShot*.

The algorithm appears in a TLA$^+$ module that declares *Proc* as for the *GFX* algorithm, declares the set *Val*, which represents the set of all possible values that can be added to S, and defines the operator *PUnion* by

$$PUnion(A) \triangleq \text{UNION } \{A[p] : p \in Proc\}$$

The body of the **while** loop describes the $snap(v)$ operation, where the value v is chosen by executing the **with** $(v \in Val)$ statement. The result returned by the operation is written to the process-local variable *out*. The set S of values maintained by the algorithm equals $PUnion(nextout)$. Thus, action A represents choosing the value v; action B represents adding v to S and reading the current value of S (into $nextout[self]$); and action C represents returning the value read.

The Algorithm

Algorithm *SnapShot* appears in Figure 4. It uses two infinite arrays $A2$ and $A3$ of MWMR registers. The code contains no notation that hasn't appeared in previous algorithms. The single atomic action c atomically reads both $A2$ and $A3$ in evaluating $NUnion(A2)$ and $NUnion(A3)$. However, the value of $A2$ that it reads is used only in the statement that writes to *nextout*, a "history" variable that is never read. This variable is used only to reason about the algorithm. The **else** clause in which *nextout* is set is not meant to be implemented.

A process p begins the algorithm by executing the *GFX* algorithm and writing the value $result[p]$ it obtains into $A2[Cardinality(result[p]) - 1]$. Since *GFX* ensures that two processes cannot obtain different values of *result* having the same cardinality, a value written in any register $A2[i]$ remains there forever. Since $result[p]$ contains p and is a subset of the participating processes, this implies that $NUnion(A2)$ is a subset of the participating processes containing all processes that have finished executing the *GFX* algorithm.

The execution of algorithm *GFX* and writing into $A2$ is represented by action a of *SnapShot*. Action a consists of action A of algorithm *GFXSpec* plus the assignment to $A2$. Having proved that *GFX* implements *GFXSpec*, we can represent the code of *GFX* by the corresponding code of *GFXSpec*. More precisely, we proved that algorithm *GFX* implements *GFXSpec* under a refinement mapping in which *result* is implemented by variable *result* of *GFX*. From this, it follows that proving the correctness of algorithm *SnapShot* proves the correctness of an algorithm in which the code from *GFXSpec* in step a is replaced by the corresponding code of *GFX*.

```
--algorithm SnapShot
{ variables result = [p ∈ Proc ↦ {}],
            A2 = [i ∈ Nat ↦ {}],  A3 = [i ∈ Nat ↦ {}];
  process (Pr ∈ Proc)
    variables myVals = {},  known = {},  notKnown = {},
              lnbpart = 0,  nbpart = 0,  nextout = {},  out = {} ;
    { a: with (P ∈ {Q ∈ SUBSET Proc :
                        ∧ self ∈ Q
                        ∧ ∀p ∈ Proc \ {self} :
                            ∨ Cardinality(result[p]) ≠ Cardinality(Q)
                            ∨ Q = result[p]
                  } )
        { result[self] := P } ;
      A2[Cardinality(result[self]) − 1] := result[self] ;
      b: while ( TRUE )
           {    with (v ∈ Val) { myVals := myVals ∪ {v} } ;
                known := myVals ∪ known ;
                nbpart := Cardinality(NUnion(A2)) ;
            c: lnbpart := nbpart ;
                known := known ∪ NUnion(A3) ;
                notKnown := {i ∈ 0 .. (nbpart − 1) : known ≠ A3[i]} ;
                if (notKnown ≠ {}) { d: with (i ∈ notKnown)
                                        { A3[i] := known };
                                     goto c }
                else if (nbpart = Cardinality(NUnion(A2)))
                        { nextout := known } ;
            e: nbpart := Cardinality(NUnion(A2)) ;
                if (lnbpart = nbpart) {out := known}
                else {goto c}
           }
    }
}
```

Fig. 4. Algorithm *SnapShot*.

The **while** loop at label b implements the **while** loop of *SnapSpec*. Action b, the first action of the loop, first chooses the value v for which the process is performing the *snap* operation and adds it to *known*. It then writes $Cardinality(NUnion(A2))$, which is an upper bound on the number of processes executing the **while** loop, into *nbpart*. The loop body then executes the Leaky Repository Protocol to write *known* into registers $A3[0], \ldots, A3[nbpart - 1]$. The properties of the protocol ensure that if the write succeeds, then the value that was written will remain forever a subset of $NUnion(A3)$ if there are still at most *nbpart* processes executing the **while** loop. If so, the *snap* operation finishes and returns that value (by writing *out*); otherwise, the process tries again.

Observe the similarity of actions c and d of algorithm *SnapShot* and the process code (actions a and b) of algorithm *GFX*. If you understand why algorithm *GFX* is correct, you will see why algorithm *SnapShot* is. In fact, algorithm *SnapShot* is less subtle because it makes use of a possibly incorrect upper bound on the number of participants, trying again if the bound was not correct.

The safety property satisfied by algorithm *SnapShot* is that it implements *SnapSpec* under a suitable refinement mapping. However, a single process of *SnapShot* executing its a action can implement the simultaneous execution of action C by multiple processes of *SnapSpec*, each executing the action's **or** clause. To define the refinement mapping, we would have to add a special kind of auxiliary variable that adds "stuttering steps" to algorithm *SnapShot* [1]. Instead of doing that, we modify our specification to allow such simultaneous steps. The necessary specification cannot be expressed in PlusCal, but it is easily written in TLA$^+$ starting from the PlusCal translation of the algorithm in Figure 3.

The modified specification is implemented under the refinement mapping that substitutes the variables *myVals*, *nextout*, and *out* of *SnapShot* for the corresponding variables of *SnapSpec*, and that substitutes the following expression for variable *pc* of *SnapSpec*:

$$\begin{array}{l}[p \in Proc \mapsto \text{CASE } pc[p] \in \{\text{“a”}, \text{“b”}\} \to \text{“A”} \\ \qquad\qquad\qquad\quad \square\ pc[p] \in \{\text{“c”}, \text{“d”}\} \to \text{“B”} \\ \qquad\qquad\qquad\quad \square\ pc[p] = \text{“e”} \to \text{IF } lnbpart[p] = Cardinality(NUnion(A2)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN “C” ELSE “B” }]\end{array}$$

As usual, the proof of this implementation rests on an invariance proof. The key part of the inductive invariant is:

$$\forall p \in Proc : \forall P \in PA3 : nextout[p] \subseteq NUnion(P)$$

where $PA3$ is the set of potential values of $A3$, defined the same way as the set $PA1$ of potential values of $A1$ for algorithm *GFX*. A rigorous proof is available on the Web [15].

To show that the algorithm is non-blocking, we first deduce from the non-blocking property of algorithm *GFX* that every participating process eventually reaches the **while** loop. At that point, $A2$ never again changes, so eventually the **if** condition in step e is forever true. The proof is then the same as the proof for algorithm *GFX*, being based on the monotonicity of the values of $known[p]$ and the fact that a nonterminating process keeps reading and writing $A3$.

It is obvious that the *SnapShot* algorithm uses at most $O(k)$ of the $A2$ and $A3$ registers plus the $O(k)$ registers used by algorithm *GFX*, where k is the cardinality of *Proc*.

4 Conclusion

We have built on earlier work of Delporte-Gallet et al. (DFGR) [11]. Unlike previous implementations of SWMR registers using arrays of MWMR registers, DFGR provided a non-blocking implementation that did not first solve the renaming problem to allocate registers to processes.

However, their implementation required a known bound n on the number of participating processes. It used the Leaky Repository Protocol with n registers, so there were not enough different processes to destroy all traces of a write. To eliminate this requirement, we take full advantage of the protocol in algorithm *GFX*, which allows all traces of a write to be destroyed if each register's value is overwritten by a different process. Using algorithm *GFX*, processes can determine the current number n of participants. We then use a variant of the DFGR algorithm that assumes there are at most n participants, but that aborts and retries if n changes while performing an operation.

We have tried to make our algorithm easier to understand by breaking it into the *GFX* algorithm and the *SnapShot* algorithm that uses *GFX* as a "subroutine". The proof that the two algorithms are non-blocking is straightforward. The safety properties of both algorithms depend on their use of the Leaky Repository Protocol. Here, we have given informal correctness proofs. We have written short, completely formal PlusCal descriptions of the algorithms. Formal machine-checked proofs of their safety properties are available [15].

We have considered only complexity in the number of registers. DFGR showed that at least n registers are required to implement n SWMR registers, so the linear number of registers used by our algorithms is optimal up to a constant factor. The question of step complexity is still completely open. We conjecture that there is an adaptive snapshot algorithm with a linear number of registers with cubic step complexity.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
3. Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *FOCS*, pages 262–272. IEEE Computer Society, 1999.
4. James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.
5. E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
6. James Aspnes. Slightly smaller splitter networks. *CoRR*, abs/1011.3170, 2010.
7. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
8. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002.
9. Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.
10. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51. ACM Press, 1993.
11. Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Linear space bootstrap communication schemes. In *ICDCN*, volume 7730 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2013.
12. Faith Ellen Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.
13. Eli Gafni. A simple algorithmic characterization of uniform solvability. In *FOCS*, pages 228–237. IEEE Computer Society, 2002.
14. Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC*, pages 161–169. ACM, 2001.
15. Leslie Lamport. Proofs for adaptive linear-space renaming. `http://research.microsoft.com/en-us/um/people/lamport/tla/snapshot.html%` .
16. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

17. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
18. Leslie Lamport. The pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.
19. Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.