Uniform Solvability with a Finite Number of MWMR Registers
(Extended Abstract)

Marcos K. Aguilera[1], Burkhard Englert[2], and Eli Gafni[3]

[1] HP Labs Systems Research Center, 1501 Page Mill Road, Mail Stop 1250
Palo Alto, CA 94304
`aguilera@hpl.hp.com`

[2] University of California Los Angeles, Dept. of Mathematics
Los Angeles, CA 90095-1555
`englert@math.ucla.edu`

[3] University of California Los Angeles, Dept. of Computer Science
Los Angeles, CA 90095-1596
`eli@cs.ucla.edu`.

Abstract. This paper introduces a new interesting research question concerning tasks. The weak-test-and-set task has a uniform solution that requires only two Multi-Writer Multi-Reader (MWMR) registers. Recently it was shown that if we take the long-lived version and require a step complexity that is adaptive to interval contention then, like mutual exclusion, no solution with finitely many MWMR registers is possible. Here we show that there are simple tasks which provably cannot be solved uniformly with finitely many MWMR registers. This opens up the research question of when a task is uniformly solvable using only finitely many MWMR registers.

1 Introduction

A *uniform protocol* [10,12,26,29] is one that does not use information about the number of processors in the system. Such protocols, by definition, must work with any arbitrary, but finite, number of participants. They are important in dynamic settings in which we do not want to force an upper bound on the number of participating processors. One of the simplest non-trivial uniform protocols is a splitter [31,33], which requires only two shared registers, and can be directly used to achieve a very weak form of mutual exclusion, in which (1) safety is always guaranteed: no two processors enter the critical section, (2) liveness is guaranteed only in solo executions: if a processor runs alone then it enters the critical section, and (3) no re-entry in the critical section is possible (one-shot).[1]

Uniform protocols, however, are usually built on top of one-shot adaptive collect [20], which is implemented by an infinite binary tree of splitters. Can

[1] There are protocols to "reset" the splitter to allow re-entry, but these protocols are not uniform.

one-shot collect be accomplished with a finite number of MWMR registers? It is known that mutual exclusion among n processors requires n shared registers [21], and hence admits no uniform implementations with bounded memory. The proof of this result relies heavily on the long-lived (reusable) nature of mutual exclusion. More recently, [1] has studied another variant of mutual exclusion called weak-test-and-set. Roughly speaking, weak-test-and-set satisfies properties (1) and (2) above, but it allows re-entry after a processor has left the critical section, so that weak-test-and-set is a long-lived (reusable) object. [1] shows that weak-test-and-set has no uniform wait-free implementations with finitely many MWMR registers. Like with mutual exclusion, the result relies heavily on the long-lived nature of weak-test-and-set. For example, as observed above, a "one-shot" version of weak-test-and-set has a trivial uniform wait-free implementation using a splitter, i.e., using a finite number of Multi-Writer Multi-Reader (MWMR) registers. In this paper we show that long-livedness, accompanied with the requirement that complexity adapt to interval-contention [2,3,4,5,7,8,9,11,13, 15,17,18,19,22,28,32,34], is not the only requirement that precludes a solution in finite space. To do so, we introduce a new task [27] that is a simple generalization of the one-shot weak-test-and-set. Roughly speaking, a task assigns a finite set of possible output tuples for each possible set of participating processors. The *generalized weak- test-and-set task* is specified as follows: The set of processors $p_0, p_1, p_2, \ldots$ is a priori partitioned into classes. The output tuple for a set of participating processors that all belong to the same class is all 1's. Otherwise an output tuple for a participating set of mixed classes of processors consists of 0's and 1's with no two processors of different classes outputting 1.

There is a very simple uniform solution to this task: a participating processor registers its name and uses collect [14] to obtain the set of all registered processors. If the set has more than one class then the processor outputs 0, else it outputs 1. This protocol works, but requires an unbounded number of shared MWMR registers. If we are limited to finitely many MWMR registers, we show that the weak-test-and-set task has no uniform wait-free implementation. This opens up the interesting research question of characterizing what tasks are uniformly solvable in finite space! Furthermore, we show that this impossibility relies heavily on two assumptions: (1) that the cardinality of classes is not uniformly bounded, and (2) that the number of classes is infinite. In fact, we show that if we relax any of these assumptions, that is, if (!1) there is a single upper bound on the number of processors in all classes *or* (!2) there are only finitely many classes, then the generalized weak-test-and-set task has a uniform solution with finitely many registers. (Note that neither (!1) nor (!2) means that the system has finitely many processors).

Obviously, since collect solves the generalized weak-test-and-set, it cannot be implemented with only finitely many MWMR registers.

Related Work

Uniform protocols, i.e., protocols that do not require a priori knowledge of the number of processors in the system, have been studied, particularly in the context of ring protocols (e.g. [12,29]). Adaptive protocols, i.e. protocols whose step complexity is a function of the size of the participating set, have been studied in [4,5,6,16,22,32]. Adaptive protocols that only use the number n of processors to bound their space complexity can be easily modified into a uniform protocol that uses unbounded memory by replacing n with ∞. Long-lived adaptive protocols that assume some huge upper bound N on the number of processors, but require the complexity of the protocol to be a function of the concurrency have been studied in [2,3,7,8,9,17,18,19,20,28,33].

As we mentioned before, the weak-test-and-set object is defined in [1]. It is a long-lived object, rather than a single-shot task. Our generalized weak-test-and-set task degenerates to a single-shot version of weak-test-and-set when there is only one processor per class.

Generalized weak-test-and-set is related to group mutual exclusion [30], much in the same way that weak-test-and-set is related to mutual exclusion. Group mutual exclusion is a generalization of mutual exclusion in which multiple processors may enter the critical section simultaneously, provided they are part of the same group. It allows a processor to block if the critical section is occupied by processors from another group. This is quite different from generalized weak-test-and-set, which admits non-blocking solutions. Moreover, group mutual exclusion is a long-lived problem, which needs to concern itself with reentry into the critical section, whereas the generalized weak-test-and-set task is single shot.

The covering technique used in our impossibility proof with finitely many MWMR registers first appeared in [21] to show some bounds on the number of registers necessary for mutual exclusion. However, the proof in [21] inherently relies on the long-lived nature of mutual exclusion: it makes processors execute mutual exclusion multiple times, while leaving some harmful residue for each execution. In this way, after a large number of executions, the protocol must finally fail. In our proof, we deal with tasks, which are inherently single-shot, and so we require a different approach to get a contradiction in a single execution. Fich, Herlihy and Shavit [25] gave an $\Omega(\sqrt{n})$ lower bound on the number of multi-writer multi-reader registers needed for randomized consensus. This also shows non-uniformity for a one-shot task using only MWMR registers. Generalized weak test-and-set, however, is a task that is much simpler and can hence be solved much more easily than randomized consensus. Their [25] covering construction relies heavily on the fact that processors need to agree on a decision. In our case, this is not required. Processors can leave by simply outputting 0, whenever they see any processor from a different group. As we will show in Section 5.1, there is, for example, a very simple implementation of generalized weak test-and-set if the number of groups is less or equal than the number of MWMR registers and each group is possibly infinite, i.e. with infinitely many processors and finitely many MWMR registers.

2 Model

We consider an asynchronous shared memory system with registers, in which a set Π of processors can communicate with each other by writing to and reading from a finite number of registers. Registers are Multi-Writer Multi-Reader (MWMR), meaning that they can be read and written by any processor in the system, and they can hold values in $\mathbf{N} = \{1, 2, \ldots\}$. For some of our results, we also consider Single-Writer Multi-Reader (SWMR) registers and, in fact, without loss of generality, we assume that each processor has at most one such register to which it can write (and it can be read by any processor).

The correctness condition for concurrent accesses to registers is linearizability.

The set Π of processors that *may* participate is countably infinite and, in fact, we assume that $\Pi = \mathbf{N}$. Elements of Π are called processor id's. Not all processors actually run simultaneously; in fact, in each run, only a finite number of them execute. These are called the set of *participating processors* or simply *participants*. Protocols running in our model are *uniform*: they do not know a priori the number of participants or a bound on this number.

Processors may fail by crashing. A processor that does not crash is called *correct*. We consider *wait-free* protocols, that is, protocols that guarantee that correct processors always make progress, regardless of the behavior of other processors (e.g., even when other processors have crashed).

2.1 Tasks

Traditionally, a task is defined to be a function from inputs to sets of outputs. However, for our model with infinitely many processors, we assume without loss of generality that the processor id and the input have been coalesced together and we call both of them the processor id. Hence, we define a task T to be a function that maps finite subsets S of processors to sets of outputs, where each output is a map from S to $\mathbf{N}$, that is $T(S) \in 2^{S \to \mathbf{N}}$. Intuitively, $T(S)$ is the set of all possible outcomes when S is the set of participating processors. Each outcome $f \in T(S)$ is a mapping $f : S \to \mathbf{N}$ that indicates the output of each processor in S.

A protocol solves a task T if, whenever S is the set of participating processors, there exists a mapping $f \in T(S)$ so that every *correct* processor p outputs $f(p)$. Note that tasks are inherently "short-lived": there is only one execution of a task in a run of the system.

We say that a protocol for a task is *adaptive* [2,3,4,5,7,8,9,11,15,17,18,19,20, 22,28,34] if the number of steps (read/write operations) taken by each processor is bounded by a function of the contention. To strengthen our impossibility results, we use a very weak notion of contention: we define contention to be the number of participants from the beginning of the run until the processor outputs a value.

3 The Generalized Weak Test-and-Set Task

Roughly speaking, in the generalized weak test-and-set task, processors are a priori partitioned into classes, and the goal is for processors in at most one of the classes to "win", that is, to output 1. All other processors must output 0. In order to avoid trivial solutions where every processor always outputs 0, we also require that if all participants belong to the same class then they all output 1.

More precisely, the following two properties must be satisfied:

– All participants that output 1 belong to the same class;
– If all participants belong to the same class then they all output 1.

4 Impossibility Results

4.1 Impossibility with Finitely Many Registers

In this section, we show that there is no uniform solution for the generalized weak test-and-set task with finitely many shared registers (either MWMR or SWMR). Without loss of generality, we assume that all the registers are MWMR. In the next section, we strengthen our impossibility result to allow an infinite number of SWMR registers (and still a finite number of MWMR registers), but we assume that the protocol is adaptive.

Theorem 1. *There is no uniform wait-free implementation of the generalized weak-test-and-set task in a system with finitely many shared registers.*

We show the theorem by contradiction: assume there is such an implementation that uses only k registers $r_1, \ldots, r_k$.

Definition 1. *A register configuration is a k-tuple containing the state of each register in the system.*

Note that this notion only makes sense at times when all of the registers have only one possible linearization (e.g., it does not make sense if there is an outstanding write). This will always be the case in the states that we consider.

We now progressively construct many runs $R_1, R_2, \ldots$ such that in R_i, all processors belong to the same class C_i. As we shall see, we will build a run $R_{i,j}$ that is a mix of R_i and R_j (for infinitely many $j \neq i \in \mathbf{N}$). Since all processors in R_i belong to class C_i and all processors in R_j belong to C_j, where $C_i \cap C_j = \emptyset$ for all $i \neq j \in \mathbf{N}$, the id's of processors in R_i are always different from the ones in R_j. In this way, the processors running in $R_{i,j}$ are the disjoint union of the processors in R_i and R_j.

In each run R_i, we start $bignum_0$ processors from class C_i and let them execute solo, one after the other, until they are about to write to their first register. Throughout this proof $bignum_h$ and $bignum'_h$ are large numbers to be determined later (for every $h \in \mathbf{N}$) .

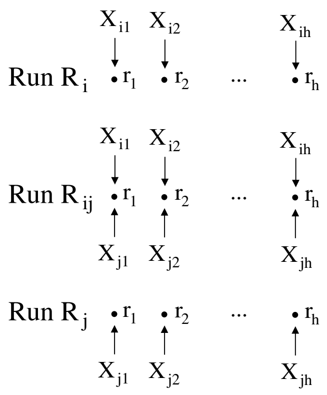

Fig. 1. Situation at the end of phase h of the inductive construction. Each set X_{ik} (resp. X_{jk}) represents a large number of processors from class i (resp. class j) that are about to write to register r_k. In run R_i (resp. R_j) there are only processors from class i (resp. class j), and run R_{ij} has the union of processors from R_i and R_j.

Our construction proceeds in phases that we construct inductively. We begin with phase 0 of the construction:

Since there are only finitely many registers, we can pick an infinite subset $R_{j_1}, R_{j_2}, \ldots$ of the runs $R_1, R_2, \ldots$ such that at least $bignum'_0$ processors in each run cover the same register, call it r_1. To avoid cluttering up notation, we will call these runs $R_1, R_2, \ldots$ rather than $R_{j_1}, R_{j_2}, \ldots$

Now for every $i \neq j \in \mathbf{N}$, we can construct another run $R_{i,j}$ as a combination of R_i and R_j, that is, processors in R_i and R_j run solo, one after the other in some irrelevant order, until they are about to write to r_1. Note that for every $i \neq j$, all runs R_i and $R_{i,j}$ have the same register configuration, since no processor has yet written to any registers.

In phase 1, we extend R_i by starting $bignum_1$ new processors (processors that have not executed before) from class C_i and letting them run solo, one after the other, until they either (a) terminate or (b) they cover a register different from r_1 (if they attempt to write to r_1, we let them do so and let them continue execute until they cover a register different from r_1).

We now show inductively for any phase $1 \leq h \leq k$ that (a) cannot happen in phase h.

Claim. For all $1 \leq h \leq k$, $i \in \mathbf{N}$, no processor executing steps in phase h of run R_i will terminate in phase h.

Proof. We proceed inductively to build phase h of the construction from phase $h-1$. At the end of phase $h-1$, we have runs R_i and $R_{i,j}$ (for all $i \neq j \in \mathbf{N}$) such that (1) in R_i there is a large number of processors covering $r_1, r_2, \ldots, r_{h-1}$, (2) each processor in R_i and in R_j have the same state as in $R_{i,j}$. See Figure 1.

To ensure that R_i and $R_{i,j}$ have the same register configuration, we pick in both runs processors (one for each register) that cover each of $r_1, r_2, \ldots, r_{h-1}$, and let them execute one step. We do so for R_i (we will get back to $R_{i,j}$ below). Then, in R_i we start $bignum_{h-1}$ new processors from class C_i and let them run solo, one after the other, until they either (a) terminate or (b) they cover a register different from $r_1, r_2, \ldots, r_{h-1}$.

We now show that no processor can terminate in phase h (i.e. that (a) cannot happen):

By way of contradiction, assume that a processor p terminates before writing to a register different from $r_1, r_2, \ldots, r_{h-1}$. Then p must output 1 by property (1) of the generalized test-and-set task, since in R_i all processors are from class C_i. Moreover, in $R_{i,j}$ we can execute the same processors that executed one step in R_i to overwrite $r_1, r_2, \ldots, r_{h-1}$, and then let p execute. Then, p will behave exactly as in R_i; hence p outputs 1 in $R_{i,j}$. Now pick processors in R_j that cover $r_1, r_2, \ldots, r_{h-1}$ (one for each register) and let them all execute one step (overwriting the contents of $r_1, r_2, \ldots, r_{h-1}$). Do the same in $R_{i,j}$. We can now start a new processor in $R_{i,j}$ from class C_j and let it execute until completion. Such a processor will behave as if it were running in R_j and hence will decide 1 in $R_{i,j}$. This violates property (2) of the generalized test-and-set task and shows the claim. □

Now since there are only finitely many registers, we can pick an infinite subset $R_{j_1}, R_{j_2}, \ldots$ of the runs $R_1, R_2, \ldots$ such that at least $bignum'_{h-1}$ processors cover the same register, call it r_h. To avoid cluttering up notation, we will call these runs $R_1, R_2, \ldots$rather than $R_{j_1}, R_{j_2}, \ldots$

We now extend $R_{i,j}$ by "pasting" runs R_i and R_j one after the other (Note that, since in both runs we begin by allowing for each register $r_1, r_2, \ldots, r_{h-1}$, one processor covering each such register to complete its write, it does not matter which run comes first, say R_i as follows.) In $R_{i,j}$ we allow the same processors that executed one step in R_i to execute the same step in $R_{i,j}$ and to overwrite $r_1, r_2, \ldots, r_{h-1}$. Then we start $bignum_{h-1}$ new processors from class C_i and let them run solo. Such processors will behave exactly as in R_i and, after doing so, processors in R_i will have the same state in both R_i and $R_{i,j}$.

We then paste R_j in the same manner. Thus, we have (1) in R_i there is a large number of processors covering $r_1, r_2, \ldots, r_h$, and (2) each processor in R_i or R_j have the same state as in $R_{i,j}$. This concludes phase h.

By carrying out these constructions until phase $h = k+1$ (recall that k is the number of registers), we get a contradiction because a processor will be executing forever (by Claim 4.1, the processor cannot terminate before it attempts to write to a register different from $r_1, r_2, \ldots, r_k$, but unfortunately there is no such register).

4.2 Infinitely Many SWMR Registers

We will now show that our impossibility result holds for adaptive implementations even if we allow processors to use an infinite number of SWMR registers (but finite number of MWMR registers). Without loss of generality, we assume that each processor has one SWMR register assigned to it. In this case the runs constructed in Theorem 1 might not be valid anymore since in $R_{i,j}$ processors from R_i might read the SWMR register of a processor from R_j and not write to r_1 ($r_1, ..., r_{h-1}$ respectively) anymore and instead terminate with 0. Moreover, processors from R_i in $R_{i,j}$ might read the SWMR register of a processor from R_j causing them to terminate. Note, however, that in R_i and R_j itself, we do not need to worry about covering processors being discovered through their SWMR registers. As long as processors from say R_i do not "discover" any processor from R_j in $R_{i,j}$ they are still forced to write to a new MWMR register. Note, moreover, that in $R_{i,j}$ there are no traces of the processors covering the MWMR registers $r_1, ..., r_{h-1}$ in any MWMR register that could "point" a processor say from R_i to the SWMR registers of a processor from R_j.

To avoid these problems, we use a technique first presented by Afek, Boxer and Touitou [1]. We prevent processors whose SWMR registers are later read from taking part in the constructed runs. So, at any given state in a run $R_{i,j}$, if processor p from R_i (R_j) reads the SWMR register of processor q from R_j (R_i) and q is participating and currently covering one of the MWMR registers, then we construct another run in which q is replaced by another processor q'. Processor p will still read the same SWMR registers, i.e. of q and not of q'. Let $R_{i,j}$ be the run in which q is participating and $R'_{i,j}$ be the run in which q' is participating. As in [1], the behavior of q and q' is somewhat equivalent, i.e. they are both writing and covering the same MWMR registers in $R_{i,j}$ and $R'_{i,j}$. A processor like q' always exists because: (1) There is a large enough set of processors to select q' from since only a bounded number of processors participated in the run so far and we are able to choose from infinitely many processors. (2)Processor p can perform only a bounded number of read operations - exactly a function of the number of processors that performed steps so far.

During all the runs $R_{i,j}$, some i, j, previously constructed, we maintain a large enough set of "equivalent" runs. These runs allow us to replace the run at any given point in which we enter a *dangerous* state, i.e. a state where a processor from one class reads the SWMR register of a participating processor from another class.

Definition 2. *Let p be a processor participating in run R_i. We say that a state s in run $R_{i,j}$ is i-transparent for p, if there is a run segment starting at s in which p takes steps that p cannot distinguish from a run segment in R_i.*

Definition 3. *Two runs $R_{i,j}$ and $R'_{i,j}$ are equivalent with respect to a set of processors G from R_i (R_j) if (1) the state at the end of both runs $R_{i,j}$ and $R'_{i,j}$ is i- transparent (j-transparent) with respect to G, (2) the MWMR registers $r_1, ..., r_h$ covered in $R_{i,j}$ and $R'_{i,j}$ are the same and they are covered by processors from the same classes and (3) if processor $p \in G$ participating in R_i (R_j) participates in both $R_{i,j}$ and $R'_{i,j}$ then p cannot distinguish between the two.*

When constructing a run $R_{i,j}$ in the proof, whenever a covering processor q from R_i (R_j), that we selected to participate in the run is discovered by a processor from R_j (R_i) we need to replace it by a processor q' from R_i (R_j) that cannot be discovered.

We achieve this by considering an equivalent run in which q' takes steps instead of q. Note that if we remove q from $R_{i,j}$ we also automatically remove it from R_i.

Theorem 2. *There is no uniform adaptive wait-free implementation of the generalized weak-test-and-set task in a system with finitely many MWMR registers and infinitely many SWMR registers.*

Proof. We proceed as in the proof of Theorem 1: We progressively construct many runs $R_1, R_2, \ldots$ such that in R_i, all processors belong to the same class C_i. We will build a run $R_{i,j}$ that is a mix of R_i and R_j (for infinitely many $j \neq i \in \mathbf{N}$). In this way, the processors running in $R_{i,j}$ are the disjoint union of the processors in R_i and R_j.

In each run R_i, we start $bignum_0$ processors from class C_i and let them execute solo, one after the other, until they are about to write to their first register. As before, $bignum_h$ and $bignum'_h$ are large numbers to be determined later (for every $h \in \mathbf{N}$) .

The construction proceeds in phases that we construct inductively. We begin with phase 0 of the construction:

Since there are only finitely many registers, we can pick an infinite subset $R_{j_1}, R_{j_2}, \ldots$ of the runs $R_1, R_2, \ldots$ such that at least $bignum'_0$ processors in each run cover the same register, call it r_1. To avoid cluttering up notation, we will call these runs $R_1, R_2, \ldots$ rather than $R_{j_1}, R_{j_2}, \ldots$

Now for every $i \neq j \in \mathbf{N}$, we can construct another run $R_{i,j}$ as a combination of R_i and R_j, that is, processors in R_i and R_j run solo, one after the other in some irrelevant order, until they are about to write to r_1. Note that for every $i \neq j$, all runs R_i and $R_{i,j}$ have the same register configuration, since no processor has yet written to any registers.

In phase 1, we extend R_i by starting $bignum_1$ new processors (processors that have not executed before) from class C_i and letting them run solo, one after the other, until they either (a) terminate or (b) they cover a register different from r_1 (if they attempt to write to r_1, we let them do so and continue executing). We show inductively for any phase $1 \leq h \leq k$ that (a) cannot happen in phase h.

Claim. For all $1 \leq h \leq k$, $i \in \mathbf{N}$, no processor executing steps in phase h of run R_i will terminate in phase h.

Note that this is the inductive claim from Theorem 1. The proof of this claim, however, might not be valid anymore, since in each phase $1 \leq h \leq k$, a participating processor p in $R_{i,j}$ might read the SWMR register of a processor q from class C_j and immediately output 0. To avoid this problem, we construct in each phase h of the construction for each $i \neq j \in \mathbf{N}$, i.e. each $R_{i,j}$ a run $R'_{i,j}$ that is *equivalent* to $R_{i,j}$. In the new run $R'_{i,j}$ we remove active processors that are later "discovered" by other active processors, i.e. any processor covering a register, whose SWMR register is later read by another participating processor, has been removed in $R'_{i,j}$. So, for each participating processor p that reads the SWMR register of participating processors $q_0, ..., q_n$ (from class C_j), we replace in $R'_{i,j}$, $q_0, ..., q_n$ with processors $q'_0, ..., q'_n$ from class C_j. Hence in $R'_{i,j}$ the processors $q'_0, ..., q'_n$ take steps instead of $q_0, ..., q_n$ and no participating processor reads the SWMR register of $q'_0, ..., q'_n$. Intuitively, we construct such a new run $R'_{i,j}$ that is equivalent to $R_{i,j}$ by repeating the construction of the current run from the beginning. In the new run we remove participating processors from R_j (R_i) that are later discovered from processors in R_i (R_j). We replace them with other processors that are from R_j (R_i) that are not discovered and whose behavior is *equivalent*.

It remains to show that there are always such replacement processors q'. This follows from the fact that (1) in any given phase a processor is allowed to read at most a bounded (by a function of the number of processors that performed any steps up to this stage) number of SWMR registers. By the generalized weak test-and-set task specification, a participating processor cannot perform at any given stage an unbounded search of all (infinitely) many SWMR registers, since it would then possibly enter a deadlocked state. (2) Only a finite number of processors participated up to any given stage. This means, we are always able to select all needed processors q' from an infinite set of processors. (Note, that this might require us to adjust $bignum_h$ for each phase h such that we always are able to chose q' from sufficiently many processors.)

Hence we obtain a construction where no processor reads the SWMR register of any other participating processor. Such a construction has the properties of the run in Theorem 1. Hence the proof of the claim follows. As in the proof of Theorem 1, this proves the theorem. □

5 Possibility Results

The following two possibility results prove that our impossibility result for an implementation with a constant number of MWMR registers relies heavily on the following two assumptions: (1) that there are infinitely many classes of processors, and (2) that there is no finite bound b on the number of processors that are in the same class. In fact, we now provide uniform implementation with finitely many MWMR registers when either (!1) there is a single upper bound on the number of processors in all classes *or* (!2) there are only finitely many classes.

5.1 Number of Classes Is Finite

To implement generalized weak-test-and-set with a finite number k of classes of processors (where each class can have an infinite number of processors), we simply assign a MWMR register to each class. All MWMR registers initially contain $\perp$. A processor executing the generalized weak-test-and-set algorithm $\mathcal{A}$ simply first writes 1 to the MWMR register corresponding to its class and then scans all other MWMR registers. If it sees 1 only in the MWMR register assigned to its class, it outputs 1. Otherwise, it outputs 0.

Theorem 3. *Algorithm $\mathcal{A}$ is an implementation of the generalized weak-test-and-set task with finitely many classes of processors using only a constant number of MWMR registers.*

Proof. Let C be a class of processors. Since all processors in C write the same value (1) to the same MWMR register and since this value once written is never changed, the processors in C, for each of the finitely many such classes C, emulate a single writer. Hence the claim follows. □

5.2 Number of Processors in Each Class Is Uniformly Bounded

Assume there is an upper bound $b < \infty$ on the size of each class C of processors (but there may be an infinite number of classes). We now provide an implementation $\mathcal{B}$ of generalized weak-test-and-set using only a constant number of MWMR registers.

The implementation uses splitters in a way described by Attiya, Fouren and Gafni [20]. A splitter is defined as follows: A processor entering a splitter exits with either *stop, left* or *right*. It is guaranteed that if a single processor enters the splitter, then it obtains *stop*, and if two or more processors enter it, then there are two processors that obtain different values. Implementation $\mathcal{B}$ uses a complete binary tree of depth $b + 1$. Each vertex of the tree contains a splitter. As in the adaptive collect algorithm of Attiya, Fouren and Gafni [20], a processor acquires a vertex v; from this point on the processor stores its values in $v.val$.

A processor moves down the tree according to the values obtained in the splitters along the path: If it receives *left* it moves to the left child; if it receives *right*, it moves to the right child. A processor marks each vertex it accesses by raising a flag associated with the vertex; a vertex is marked if its flag is raised. The processor acquires a vertex v when it obtains *stop* at the splitter associated with v; then it writes its id and class identifier into $v.id$. A processor that acquired a vertex then as in [20], by traversing the part of the tree containing marked vertices, in DFS order, collects the values written in the marked vertices. If it sees a processor from a different class than its own, it outputs 0, else 1.

If a processor p executing this implementation reaches a leaf of the tree, i.e. depth $b + 1$, without acquiring a vertex, it immediately outputs 0. Since the bound on the membership of the class of processors is b, there must be at least one processor participating that does not have the same initial value as p.

Theorem 4. *Algorithm $\mathcal{B}$ is an implementation of the generalized weak-test-and-set task with a bounded by b number of processors in each class of processors using only a constant number of MWMR registers.*

Proof. Immediate. □

6 Conclusion

Afek et al [1] showed that a long-lived task requires an infinite number of MWMR registers in case there is no a priori bound to the number of the participating processors. We have extended this result showing that the long-liveness is not necessarily the crucial ingredient that raises the need for infinite memory. This raises the interesting question on when such a memory is really required. Another way of extending the result in [1] is to notice that the proof there actually requires the concurrency to exceed the number of MWMR registers. What if we do not have an a priori bound on the concurrency? Can we nevertheless use a fortiori a finite number of MWMR registers? I.e., can splitters be reused in a uniform protocol? We conjecture they cannot—a conjecture that if proven will make the research area of adaptive algorithms a bit less appealing.

Acknowledgement. We are thankful to the DISC referees for valuable comments.

References

1. Y. Afek, P. Boxer and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proc. of 20th Annual ACM Symp. on Principles of Distributed Computing*, pp. 81–89, 2000.
2. Y. Afek, H. Attiya, A. Fouren, G. Stupp and D. Touitou. Long-Lived Renaming made adaptive. *Proc. of 18th Annual ACM Symp. on Principles of Distributed Computing*: pp. 91–103, 1999.
3. Y. Afek, H. Attiya, G. Stupp and D. Touitou. Adaptive long-lived renaming using bounded memory. *Proc. of the 40th IEEE Ann. Symp. on Foundations of Computer Science*, pp. 262–272, 1999.
4. Y. Afek, D. Dauber and D. Touitou. Wait-free made fast. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*: pp. 538–547, 1995.
5. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. of the 18th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 105–112, 1999.
6. Y. Afek, M. Merritt, G. Taubenfeld and D. Touitou. Disentangling multi-object operations. In *Proc. of 16th Annual ACM Symp. on Principles of Distributed Computing*, pp. 111–120, 1997.
7. Y. Afek, G. Stupp and D. Touitou. Long lived adaptive collect with applications. *Proc. of the 40th IEEE Ann. Symp. on Foundations of Computer Science*, pp. 262–272, 1999.
8. Y. Afek, G. Stupp and D. Touitou. Long lived adaptive splitter and applications. Unpublished manuscript, 1999.
9. Y. Afek, G. Stupp and D. Touitou. Long lived and adaptive atomic snapshot and immediate snapshot. *Proc. of the 19th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 71–80, 2000.
10. M.K. Aguilera, B. Englert and E. Gafni. On using network attached disks as shared memory. *To appear in Proc.of the 22nd Ann. ACM Symp. on Principles of Distributed Computing*, 2003.

11. J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proceedings of the 14th International Conference, DISC 2000*, pp. 29–43, 2000.
12. D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pp. 82–93, 1980.
13. J. Aspnes, G. Shah and J. Shah. Wait free consensus with infinite arrivals. *Proc. of the 34th Annual ACM Symposium on the Theory of Computing,* pp. 524–533, 2002.
14. J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proc. 28th ACM Symp. Theory of Comp.*, pp. 237–246, 1996.
15. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pp. 91–100, 2000.
16. H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th Annual ACM Symp. on Principles of Distributed Computing*, pp. 223–232, 1996.
17. H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, pp. 277–286, 1998.
18. H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 0956, Faculty of Computer Science, Technion, Haifa, 1999. http://www.cs.technion.ac.il/ hagit/pubs/tr0956.ps.gz.
19. H. Attiya and A. Fouren. Algorithms adaptive to point contention. *JACM*, 2001, submitted for publication.
20. H. Attiya, A. Fouren and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2), pp. 87–96, 2002.
21. J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation* 107(2):pp. 171–184, 1993.
22. M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1), pp. 1–17, 1994.
23. P. Fatourou, F. Fich and E. Ruppert. Space-optimal multi-writer snapshot objects are slow. *Proc. 21st Annual ACM Symp. on Principles of Distributed Computing*, pp. 13–20, 2002.
24. P. Fatourou, F. Fich and E. Ruppert. A tight lower bound for space-optimal implementations of multi-writer snapshots. *Proc. 35th Annual ACM Symp. on Theory of Computing*, pp. 259–268, 2003.
25. F. Fich, M. Herlihy and N. Shavit. On the space complexity of randomized synchronization. *JACM* 45(5), pp. 843–862, 1998.
26. E. Gafni. A simple algorithmic characterization of uniform solvability. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pp. 228–237, 2002.
27. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *JACM* 46(6), pp. 858–923, 1999.
28. M. Inoue, S. Umetani, T. Masuzawa and H. Fujiwara. Adaptive long-lived $O(k^2)$ renaming with $O(k^2)$ steps. *Proceedings of the 15th International Conference on Distributed Computing*, pp. 123–135, 2001.
29. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1): pp. 60–87, 1990.
30. Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pp.51–60, 1998.
31. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): pp. 1–11, 1987.

32. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45: pp. 137–142, 1993.
33. M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1): pp. 1–39, 1995.
34. G.L. Peterson. Time efficient adaptive mutual exclusion algorithms. Unpublished manuscript, 2001.