Theoretical Computer Science 410 (2009) 533–545

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On the cost of uniform protocols whose memory consumption is adaptive to interval contention

Burkhard Englert

California State University Long Beach, Department of Comp. Engr. & Comp. Science, Long Beach, CA 90840, United States

ARTICLE INFO

Keywords:
Asynchronous distributed system
Shared memory
Adaptive algorithms
Uniform protocol
Memory complexity
Storage area network
Infinitely many processes

ABSTRACT

Recently, we introduced a novel term, *memory-adaptive*, whose goal it is to capture what it means for a distributed protocol to most efficiently make use of its shared memory. We proved three results that relate to the memory-adaptive model in the uniform setting. We considered a *store/release* protocol where processes are required to store a value in shared MWMR memory so that it cannot be overwritten until it has been released by the process. We showed that there do not exist uniformly wait-free store/release protocols using only the basic operations read and write that are memory-adaptive to point contention. We further showed that there exists a uniformly wait-free store/release protocol using only the basic operations read, write, and read-modify-write that is memory-adaptive to interval contention and time-adaptive to total contention. This left a significant gap — it remained open as to whether there exists a uniform, memory adaptive to interval contention store/release protocol that only uses read/write (no read-modify-write) registers. In this paper, we close this gap by showing that no such protocol can exist. We furthermore illustrate the validity and practicality of the concept of memory adaptiveness by providing a uniform, memory-adaptive to interval contention store/release protocol for Network Attached Disks.

1. Introduction

Shared memory algorithms, such as collect or renaming provide essential building blocks for many applications. Most often collect or renaming are designed based on an a priori knowledge of an upper bound n on the number of participating processes or of an upper bound N on the ids of participating processes. Algorithms such as collect or renaming, however, become inefficient if only few of the n processes are actually participating. This motivated researchers to look for *adaptive* algorithms whose step complexity only depends on the number of participating processes. Besides a possibly inefficient use of *time*, inefficient use of *space* is also a potential drawback of many distributed algorithms. In particular, many shared memory algorithms require memory space whose size is a function of N (or n) even if only few of the processes are actually participating. Hence, to truly improve the efficiency of distributed algorithms the step complexity should be made adaptive to the number of participating processes, i.e. the contention, and the space requirements should (if possible) depend on the number of participating processes or the contention. Following this approach, one obtains two possible kinds of adaptive algorithms: algorithms where the *step complexity* adapts to the contention are traditionally called *adaptive*. We called such algorithms *time-adaptive* to distinguish them from algorithms where the memory space consumption adapts to the contention that we called *memory-adaptive* [31]. In *memory-adaptive* algorithms processes are only allowed to write to a shared MWMR register whose index is a function of the contention (possibly point-, interval- or total contention) during the processes previous shared memory access.

E-mail address: benglert@csulb.edu.

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.10.007

Time-adaptive algorithms have a worst case step complexity that is bounded by a function of the number of concurrently participating, or actually active processes [6]. Motivated by Lamport's MX algorithm [40], many such time-adaptive algorithms have since been designed [3,6,4,7–9,11,16,18–20,22–26,30,38,41]. The strongest forms of time-adaptiveness in the read/write shared memory model have been defined and achieved in recently presented long-lived time-adaptive collect [9] and renaming [3,23] algorithms. In these algorithms, called *time-adaptive to point contention* the number of steps taken by a process executing an operation is a function of the maximum number of processes that were active simultaneously at some point in time during this operations execution interval. Algorithms *time-adaptive to interval contention* on the other hand, have a slightly weaker level of adaptiveness. Here the number of steps taken during a given operation is a function of the total number of different processes active during the operation's execution interval. Finally, an algorithm is *time-adaptive to total contention* if the number of steps taken by a process is a function of the total number of processes active since the beginning of the execution.

With respect to memory consumption of time-adaptive renaming or collect algorithms Afek, Boxer and Touitou [5] showed that the number of Multi-Writer Multi-Reader (MWMR) registers used must be a function of N. They specifically show that for any constant d there is a large enough N_d such that every long-lived time-adaptive (to interval contention, and hence, point contention as well) read/write implementation of collect (and renaming) with N_d processes must use at least d MWMR registers. In their paper, they use a simple object called Weak-Test and Set [15] to derive their impossibility results. More recently Attiya, Fich and Kaplan [21] significantly improved on [5]. They showed that if a collect algorithm is time-adaptive to total contention, namely, its step complexity is $f(k)$, where k is the number of processes that ever became active during the current execution, then it uses $\Omega(f^{-1}(N))$ MWMR registers, where N is the total number of processes in the system.

In this paper, we will remove the assumption of a known upper bound on the number of participating processes and consider *uniform* protocols [17,33,39], i.e., protocols that do not require a priori knowledge of or an upper bound on the number of processes that may participate. At the same time, we will assume that the number of participating processes is always finite.

While memory is certainly cheap and hence one might argue that its cost does not represent a drawback of non-memory-adaptive algorithms, it remains difficult to efficiently manage a large memory in particular if no upper bound on the number of processes is known in advance. In this case, processes must usually be able to write to memory locations of any index. To investigate whether and when this must be the case we introduced the notion of *memory-adaptiveness* [31]: it requires that each(!) write operation that a process makes must be close to the "front" of shared memory. The idea here is that if protocols allow processes to write to registers whose index depends say on the processes id and no upper bound on the number of participating processes is known in advance then memory must be unpredictably large. On the other hand, if we can guarantee that the memory required by each protocol that runs on a shared memory system is a bounded function of the contention, then a distributed operating system can allocate large memory blocks to each protocol on an ad hoc basis and, on the rare occasions when it is necessary, increase or decrease the individual allocations as necessary.

Also, if processes are allowed to write to registers with arbitrary indices in time-adaptive protocols they eventually must move the values they wrote close to the beginning of memory for the protocol to stay time-adaptive during solo executions. Hence, ideally, processes will want to register in a fixed finite subset of the infinite set of MWMR registers that we called "close to the beginning of shared memory" [31].

Consider, for example, the renaming problem [3,4,6,7,10,22,23,38,42]. Here each process is required to choose a unique name for itself. Processes are allowed to use any shared MWMR register during the execution of the protocol, even a register with an extremely large index, but the final result must lie within a bounded distance from the front of shared memory. In the definition of *memory-adaptiveness*, to capture the notion of having to write close to the front of shared memory *every* time, we require processes to write to a MWMR register whose index is a function of the contention during the previous operation of the same process.

In [31] we investigated simple tasks, *store* and *release*, that require a given process to store a value in shared MWMR memory that cannot be overwritten by any other process and then to erase the value when no longer needed, freeing the memory for other processes to use.[1] We studied whether these simple commands can be implemented memory-adaptively under different assumptions about the contention of the protocol.

We showed that in a system with *infinitely* many MWMR registers and *infinitely* many SWMR registers: 1. There is no uniform, long-lived memory-adaptive to *point-contention* implementation of store/release that uses only read/write registers. 2. There does exist a uniform, long-lived implementation of store and release in the read–write model that is memory-adaptive to *total contention*. 3. Allowing write-plus (read-modify-write) there exists a uniform, long-lived implementation of store and release in the read–write model that is memory-adaptive to *interval contention*.

The question remained, however, whether in this setting there exists a uniform, long-lived, memory-adaptive to *interval contention* store/release protocol that uses **only** read/write registers. This is of particular interest because one could argue that with adaptiveness to interval contention "true adaptiveness" really starts, since adaptiveness to total contention allows

[1] Another way to view a *store* and *release* protocol is as a simple renaming protocol where the index of the register in which a value is stored becomes the new name and where *release* simply releases this name. To focus attention on the central components of such a protocol and to avoid confusion with already existing renaming protocols we chose the store/release terminology.

for the memory requirements to grow independent of the contention during operations. Moreover, even though we were able to show that there exists a protocol memory-adaptive to interval contention using read-modify-write registers, we were not able to justify the use of these stronger primitives. In this paper we will close this gap and hence significantly strengthen our previous results.

To prove our impossibility result we will use a covering construction as in the memory-adaptiveness to point contention impossibility proof in [31]. Our proof, however, will have to be more complex and requires greater care. In [31] we were able to select all potentially participating processes in advance and construct the run that produced the contradiction as a single run. Using the pigeonhole principle we simply reduced the set of participating processes in pseudo solo runs more and more until all MWMR registers at the beginning of shared memory (that are accessed in solo runs) were covered. All processes that were covering one of the MWMR registers at the end of the construction had been participating from the beginning. If we want to obtain a contradiction to memory-adaptiveness to **interval contention**, however, we cannot proceed in this manner. Every time a new register is covered we must choose a new set of processes that have never acted before since otherwise processes could receive information about the increasing interval contention allowing them to write to more (''new'') MWMR registers and making it impossible for us to get a contradiction. In other words, when we extend the covering from the first j to the first $j+1$ MWMR registers that processes write to during their pseudo solo runs, we first eliminate the traces of the process that now covers the $j+1$st register. This is done by releasing covering writes to overwrite this process. In [31] we were simply able to cover each of the required MWMR registers with infinitely many processes and then release covering writes as need be. Here we are not able to do this anymore. Instead – to ensure that our construction is memory-adaptive to interval contention – we must rebuild our covering after each time the covering writes have been released. Otherwise it might be possible for processes to detect that some of the processes involved in these covering writes were concurrently active with them and hence allow them to write to MWMR registers outside the bounded (by a constant) range of MWMR registers at the ''beginning of shared memory''. As we would cover more registers, processes would be able to memory-adaptively write to more ''new'' MWMR registers outside the ''beginning of shared memory'' making it impossible to get a contradiction.

As a result, our construction is similar to the construction in [5], however, we note that the result there does not directly imply our result since it assumes a finite number of available MWMR registers while we allow for infinitely many available MWMR registers. Therefore, we must always ensure that at all important steps of our construction processes are only able to write to the bounded set of registers at the ''beginning of shared memory''. We will do this by making these processes believe that they execute the protocol solo. Also in [5] algorithms are assumed to be time-adaptive allowing to bound the execution length of each participating process since any time-adaptive protocol is by definition, wait-free. Here, we do not assume that our protocols are time-adaptive. Instead, to bound the execution length (otherwise processes could simply keep reading each others SWMR registers) we assume the protocol to be uniformly wait-free, that is that the length of all executions is uniformly bounded.

The covering techniques used in our impossibility proofs first appeared in [28] to show some bounds on the number of registers necessary for mutual exclusion. Similar covering arguments were used in many recent papers to prove space and time lower bounds. For example, see the survey by Fich and Ruppert [32].

1.1. Network attached disks

In the second part of the paper, we will consider an important and natural application of memory-adaptive algorithms. Recent advances in storage technology [36] have enabled systems such as Storage Area Networks [13,14,27,37,43,45], which have network attached disks or NADs. A NAD is a simple device that just executes requests to read and write blocks of data. It can be accessed by any process in the system, so that the NADs effectively become a shared storage medium that can be used to solve distributed problems such as consensus, as in Disk Paxos [1,29,34]. Unlike message-passing systems, which typically require a majority of processes to be correct to avoid partitioning, NADs allow protocols that can withstand the crash of any number of processes. Therefore – as conventional shared-memory models – the model allows uniform protocols. One difficulty of this model is that a NAD can fail by crashing and thereby become inaccessible.

In [12] Aguilera, Englert and Gafni studied the uniform implementability of fail-free shared registers in various settings on Network Attached Disks (NAD's).

They showed that one cannot uniformly implement a MWMR register on a NAD with a finite number of fail-prone base registers, even if the implementation need not be wait-free. Therefore, one cannot use the standard technique of implementing a MWMR register by first implementing SWMR registers: doing so would blow up the space complexity.

On the positive side, they showed that the four types of MWMR, MWSR, SWMR and SWMR registers have a uniform implementation even in the infinite-arrival model [35], if the number of base registers is infinite. This implementation spreads registers across $2t+1$ disks – each disk with infinitely many registers – such that all registers of up to t disks may crash.

These results imply the need for infinitely many base registers. Since this is however an unrealistic assumption in real NAD's, memory-adaptive algorithms are of particular practical interest on such disks. If uniform protocols that require infinitely many base registers and that run on NAD are memory-adaptive to interval contention they will remain practical since they will allow us to efficiently bound the memory requirements based on this contention.

Based on our impossibility result, we will show how store/release can be – uniformly and memory-adaptively to interval contention – implemented on *Active Disks*. This is possible, since read-modify-write objects are available on Active Disks [2,44].

1.2. Related work

Uniform protocols have been studied (e.g., [17,39]), particularly in the context of ring protocols. Adaptive protocols, i.e. protocols whose step complexity is a function of the size of the participating set, have been studied in [6–8,20,30,41]. Long-lived adaptive protocols that assume some huge upper bound N on the number of processes, but require the complexity of the algorithm to be a function of the concurrency have been studied in [3,4,9–11,22–24,38,42].

1.3. Contributions

We summarize the contributions of our paper.

(1) Interval contention: (Theorem 1) We show that in a system with *infinitely* many MWMR registers, *infinitely* many SWMR registers and infinitely many processes there does not exist a uniformly wait-free, *memory-adaptive* (to INTERVAL contention) implementation of store/release. In other words, we show that under these conditions processes cannot memory-adaptively store a value in shared memory. This closes a gap that remained open in [31] and implies the impossibility of uniform memory adaptive to point contention algorithms [31] with all its consequences. Moreover, it justifies the use of read-modify write registers and similar stronger primitives to uniformly implement memory-adaptive to interval contention store/release [31].
(2) We present a uniform implementation of memory-adaptive to interval contention store/release on Active Disks. While it was shown [12] that one cannot uniformly implement a MWMR register on a NAD with a finite number of fail-prone base registers, these results provide a realistic and practical building block for algorithms on NAD where an upper bound on the interval contention can be enforced.

Paper Organization: We will first in Section 2 present our model, followed by our impossibility proof in Section 3. We conclude with a transfer of our algorithm to NAD's (Section 4) and some final remarks (Section 5).

2. Model and preliminaries

For our impossibility result, we use the standard shared-memory model of distributed computation. There are infinitely many processes each modeled by infinite-state machines that are capable of unbounded computation. Processes participate in a distributed deterministic asynchronous protocol and are indexed by the positive natural numbers so that each process "knows" its own "name". There are two areas of memory: the single-writer multi-reader (SWMR) space and the multi-writer multi-reader (MWMR) space. The SWMR space can be thought of as a large array of numbered registers indexed by the positive natural numbers. Each register is associated with a distinct process so that only this process is allowed to write to this register while all other processes are able to read it. Each SWMR register can store an unbounded number of bits. The MWMR registers have all the same properties as the SWMR registers with the exception that any process may both read *and* write to any register. Intuitively, we think of the SWMR registers as private memory controlled by the individual processes, and the MWMR registers as the memory domain of a separate entity with its own operating system accessible by all the processes.

Processes access the memory space using basic atomic operations. The atomic operations we will allow in this paper are *read, write,* and *read-modify-write*.

- READ: To execute a read command, a process specifies a register to be read and upon completion of the read, the process has gained a snapshot of the contents of the specified register.
- WRITE: A process specifies which register to write to (in either private or shared memory) and the data to be written. Upon completion of the write command, all previous data is overwritten with the new data specified by the process. (Note that we do not allow a process to overwrite "part" of a register.)
- READ-MODIFY-WRITE (RMW): The RMW command allows the unbreakable execution of the following code (where X is a shared variable and f is a mapping):

```
function RMW(X,f)
  begin
    temp←X;
      X←f(X);
      return(temp);
  end
```

```
weak test & set(V: memory address) //returns binary value:
reset (V: memory address):
   V:=0
```

Fig. 1. Weak test & set object.

A *protocol* is an algorithm that accomplishes a task using basic operations. An *adaptive* protocol is one in which the resources consumed by the protocol are functions of the number of processes that actually participate in the protocol (a.k.a. active processes) rather than the total number of processes. In an adaptive protocols, the size of the resources (time or space) consumed is a function of the contention. The contention can be measured in three different ways, effecting the strength of adaptiveness of a protocol: *Total contention* refers to the total number of processes that become active during the entire execution of the protocol. *Interval contention* during a given processes protocol is defined to be the total number of processes that become active during the execution interval of a processes protocol. Finally, *point contention* during a given processes protocol refers to the maximum number of processes that are simultaneously active at any point during the execution interval of this processes protocol.

Since protocols, when executing, consume time and space there are at least two natural ways to measure the complexity of an adaptive protocol. A protocol is *time-adaptive* to a particular type of contention if the maximum number of basic operations executed during the protocol by any given process is a bounded function of the contention type. This type of definition has been studied extensively [3,6,4,7–9,11,16,18–20,22–26,30,38,41].

To analyze the memory consumption of a protocol we are mainly interested in the write processes make to shared memory (MWMR registers). We hence introduced [31] a measure of complexity that we called *memory-adaptive* to capture this consumption. We say that a basic operation is *memory-adaptive* to a type of contention if and only if, the following is true. Whenever a process executes a basic operation, if the next basic operation changes the state of a shared memory register, the index of the register at which this change occurs is a bounded function of the contention (point, interval or total) at the time of the previous basic operation. (In the asynchronous model, without loss of generality, we may assume that the first basic operation in any protocol is a read, which does not change the state of any register.) In other words, a process can read wherever it wants, but it can only write to places that are as close to the "front" of shared memory as possible.

Most time-adaptive algorithms that were presented [3,6,4,7–9,11,16,18–20,22–26,30,38,41] are not memory-adaptive in this sense. They might however force that the final result of a computation lies within a bounded distance of the "front" of shared memory. For example, in the renaming protocol, each active process is required to choose a unique name for itself that is as small as possible by storing its index in a shared memory register. Its new "name" becomes the index of this shared register. Processes are allowed to use any shared register during the execution of the protocol, even a register with an extremely large index, but the final result must lie within a bounded distance from the "front of shared memory" in the sense that the new name must be a function of the contention during the execution of the protocol.

The three protocols that we will focus on in this paper are *store*, *release* and *Weak Test and Set*.

- STORE: A data value is specified in advance by the process. The goal is for the process to store the data value in some shared MWMR register in such a way that upon completion the process knows that the value will not be moved or erased by any other process until the register is explicitly released. Essentially, this amounts to storing a value in shared (MWMR) memory and locking the location. In this case we say that a process captures the register. Ideally, the index of the shared MWMR register in question will be as close to the "front" as possible.
- RELEASE: This assumes that the process has already executed a previous store protocol. Upon completion, the shared register occupied by the process is released.
- WEAK TEST AND SET: The weak-test-and-set (WT&S) object (Fig. 1) provides two operations, *weak-test-and-set (V)* and *reset(V)*. The operation *weak-test-and-set (V)* returns either 0 or 1. The *reset(V)* operation sets the value of the object to 0. A process to which 0 is returned owns the WT&S object. Only a process that owns the WT&S object can access the reset operation. The object only returns 0 to a process if its current value is 0 and if there is no concurrent (with this invocation of *weak-test-and-set (V)*) access to the object by another process. In all other cases, the object returns 1. The WT&S object can be used to implement a weak mutual exclusion property. While at most, one process can own the object at any point in time, there is no guarantee that if several processes access the object concurrently that at least one of these processes will eventually own the object. Hence if a WT&S object is used to implement mutual exclusion, deadlock freedom is not guaranteed.

  A WT&S object satisfies the following two properties:

  . **Exclusion:** At most one process owns the WT&S object at any system state of the WT&S execution.
  . **Solo Execution:** If only one process takes steps and this process accesses the WT&S object and no other process currently owns the object then this process must eventually own the object.

Note that STORE and RELEASE are fundamental building blocks useful for many distributed protocols (e.g. collect, mutual exclusion, consensus, approximate agreement, and so on). WEAK TEST AND SET on the other hand is useful in the proof of lower bounds.

We call a protocol *uniformly wait-free* if there exists a uniform bound applicable to all processes on the number of basic operations that the protocol requires before termination. All protocols considered in this paper will be uniformly wait-free.

We make the following definitions:

- A system state consists of the state of all processes and the value of all registers in the system. A system has one or more initial system states in which the system starts its execution.
- We say that a system state s is an *idle-state*, if no process accesses or owns the WT&S object at s.
- A run α is a finite or infinite sequence of events, starting from an initial system state. If the sequence is finite, we say that the run is finite.
- A run segment x_α of a run α is a finite, continuous subsequence of events of α.
- A *solo run segment* of p is a run segment starting at an idle-state, in which only p takes steps.
- A run segment is called a *p-segment* if it starts in a state s in which p neither accesses nor owns the WT&S object, and in which only p takes steps. Note that a *p-segment* is not necessarily a *p-solo-segment*.
- We say that a state s is *transparent* with respect to process p, if p neither accesses nor owns the WT&S object in s, and there is a p-segment starting in s, that p cannot distinguish from some solo run segment of p starting at an idle-state and ending with p owning the WT&S object. State s is transparent with respect to a set of processes Q, if s is transparent $\forall p \in Q$.
- A *pseudo-solo run segment* of p denoted $solo_p$, is a p-segment starting at state s s.t., s is transparent with respect to p, and ending with p owning the WT&S object. Note that by the definition of transparent, p cannot distinguish a pseudo-solo run segment from some sole run segment starting at an idle-state and ending with p owning the WT&S object.
- A register r is *covered* by process p at the end of run α, if a write operation by p to r is enabled at the end of α.
- Given a run segment x and a state s, $s \cdot x$ denotes the concatenation of x after some run α ending at s, assuming that α exists.

3. Interval contention

In [31] we showed that there is no uniform, long-lived and *memory-adaptive to point contention* store and release protocol. We will now strengthen this result by showing that there is no uniform and *memory-adaptive to interval contention* weak test and set protocol. We begin, by showing that we can implement WT&S from memory-adaptive store and release. The reduction uses the fact that store/release is uniformly wait-free and that store/release, if successful guarantees exclusive possession of a MWMR register. Store/Release does not guarantee, however, that at the same time other processes may not exclusively "own" other registers "nearby". Hence to implement WT&S from store and release it will be necessary for a process that "wins" a register to first examine these other registers before returning either the 0 or 1 bit value.

Reduction from memory-adaptive and uniformly wait-free store and release to memory-adaptive and uniformly wait-free WT&S:

In the uniform memory-adaptive store and release protocol, processes repeatedly store and release values in shared memory. The index of the MWMR registers to which they write must be in the range $\{1, \ldots, f(k)\}$ where k is the number of processes that are active concurrently with the process that is trying to store or release a value. So, when a process runs solo (i.e. it is a candidate to win the WT&S object) the index of the MWMR registers it writes to must be bounded by some constant $f(1)$. In an implementation of WT&S from memory-adaptive store and release we use one copy of the memory adaptive store and release object and an array of $f(1)$ boolean MWMR registers.

To perform the WT&S operation a process first attempts to memory adaptively store a value in shared memory. If at any point in time during the execution of the algorithm it writes to a MWMR register with an index greater than $f(1)$ it returns 1 and – if it already stored a value – releases the value it stored. It failed to win the WT&S object. Otherwise, once a process captures a MWMR register to store a value it writes "active" into the captured MWMR register and reads all other MWMR registers with indices up to $f(1)$. If it sees any other process as active in any of these registers it returns 1 and writes asleep into the boolean MWMR register it previously captured. It failed to win the WT&S object. In all other cases the process returns 0. It owns the WT&S object. To release the WT&S object a process writes asleep in the boolean MWMR register it previously captured and so resets the value it stored in shared memory. This clearly implements the desired object with the required exclusion and solo execution properties.

We furthermore assume that each process has only one single-writer, single-reader (SWMR) register: all SWMR registers of a process can always be replaced by a single SWMR register.

A condition or property holds in a run if it holds at the end of that run (unless we state otherwise).

It hence suffices to show that there is no uniform memory-adaptive to interval contention (uniformly wait-free) Weak Test And Set implementation using only read/write registers.

3.1. The theorem

Clearly, if an implementation of a weak test and set object is memory-adaptive then there exists a constant i such that, no *solo run segment* writes to a MWMR register with an index greater than i. We say that the algorithm is *"i-solo-memory-adaptive"*.

Theorem 1. *For any constant i there is no long-lived, uniformly wait-free, i-solo-memory-adaptive to interval contention implementation of Weak-Test & Set in a system with infinitely many processes and infinitely many MWMR and SWMR read/write registers.*

Note that in contrast to [5], we do not require our algorithms to be time-adaptive. Hence the result in [5] does not immediately imply our result. Moreover, the number of available MWMR registers is now unbounded, that is when covering writes are released processes that detect contention can possibly write to more than the first k MWMR registers. After addressing such issues our proof proceeds in a similar manner as [5].

Summary of Proof: The proof is by way of contradiction. First, assume that there is a memory-adaptive to interval contention WT&S implementation with infinitely many MWMR registers for a system with infinitely many processes. Then, we show that under these conditions there is a run in which two processes p and q are in the critical section, i.e. are owning the WT&S object at the same time.

(1) We construct a run prefix α s.t., the state at the end of α is transparent with respect to p, and every MWMR register that p writes in its pseudo-solo run starting at α is covered. As in [5] we construct this cover inductively.
(2) Let $solo_p$ be the pseudo-solo run segment of process p starting after α. Hence, p owns the WT&S object in $\alpha \cdot solo_p$. Let $\{r_1, \ldots, r_{i'}\}$ be the set of MWMR registers written by p in $solo_p$, where $i' \leq i$.
(3) We now enable the covering writes and wait until no process accesses or owns the WT&S object anymore. This is guaranteed by the fact that we are dealing with a uniformly wait-free WT&S implementation.
(4) We ensure that processes that are active do not detect each other by selecting them in such a way that they do not read each others SWMR registers. (This also follows from the protocol being uniformly wait-free. We show later in detail that this is possible.)
(5) We select a process q that does not read the SWMR register of p. This process will enter the critical section together with p, a contradiction.

3.2. Main inductive proof

The proof is based on [5]. We construct α by first, for explanatory reasons, making strong assumptions. We then remove these assumptions to obtain the claimed result.

We use the following notations. For an infinite set R_{MW} of MWMR registers, we consider W to be an i-solo-memory-adaptive implementation of WT&S in the Read/Write shared memory model. Note that by the definition of memory-adaptiveness i must be a constant.

Phase 1:

Assumption A. There are no write operations to SWMR registers in all legal runs. That is, we assume for the moment that there is a uniformly wait-free WT&S protocol that is i-solo-memory-adaptive to interval contention and uses no SWMR registers.

Assumption B. If G is a set of processes and s is a state that is transparent with respect to G, then during their pseudo-solo runs starting at s all processes in G write in the same MWMR registers in the same order.

These assumptions will later be removed. We will be able to remove Assumption B because of the i-solo-memory-adaptiveness of the algorithm, that is processes can only write to a fixed number of MWMR registers in pseudo-solo runs and the fact that our protocol is uniformly wait-free. Hence, using a Ramsey theoretic argument we can find a large enough set of processes that will write in the same order into these registers.

In the following lemma α is denoted by $s \cdot \beta$ and satisfies the properties of α: Property 1: the state at the end of $s \cdot \beta$ is transparent with respect to some set of processes called G_e, and property 2: there is a cover on all the MWMR registers written by processes in G_e in their pseudo-solo run segments, starting after $s \cdot \beta$. The size of the set G_e is a parameter and is determined in the proof of the theorem.

Lemma 1.1. *Let W be a long-lived, uniformly wait-free, i-solo-memory- adaptive WT&S algorithm satisfying Assumptions A and B. Then, for any constant e and for any set of infinitely many processes G, $|G| = \infty$ and for any state s transparent with respect to G, there is a run segment β and a set of processes $G_e \subseteq G$ s.t., the following holds:* (1) $|G_e| \geq e$, (2) *the state at the end of $s \cdot \beta$ is transparent with respect to G_e, and* (3) *all the MWMR registers written in the pseudo-solo run segments of processes in G_e, after $s \cdot \beta$, are covered in $s \cdot \beta$.*

Proof of Lemma. The proof is by induction on j, the inductive claim is:

Lemma 1.2. *Let W be a long-lived, uniformly wait-free, i-solo-memory- adaptive WT&S algorithm satisfying Assumptions A and B. Then for every j, $0 \leq j \leq i$ and for every constant e and for any set of infinitely many processes G, $|G| = \infty$ and for any state s transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds:* (1) $|G_j| \geq e$, (2) *the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and* (3) *there exists a set $R_j = \{r_1, \ldots, r_j\}$ of MWMR registers*

that are covered in $s \cdot \beta_j$ and $r_1, \ldots, r_j$ are the first j MWMR registers written (in this order) in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta_j$, or, the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to a set $R_{j'} \subset R_j$ (in the same order) of MWMR registers, where $R'_j = \{r_1, \ldots, r_{j'}\}$, $1 \leq j' < j$ and all these writes are covered in $s \cdot \beta_j$.

Proof. (1) For $j = 0$ the claim holds trivially by setting $G_0 = G$ and β_0 to be the empty sequence.
(2) Assume that the claim holds for $j = k$. We will show that it holds for $j = k + 1$. Recall that $|G| = \infty$ and let s be a system state transparent with respect to G. By the induction hypothesis, there exists $G_{k,1}$ and $\beta_{k,1}$ such that at the end of $s \cdot \beta_{k,1}$ writes to the first k MWMR registers in R_j that are written in the pseudo-solo run segments of processes in $G_{k,1}$ are enabled and $s \cdot \beta_{k,1}$ is transparent with respect to $G_{k,1}$. By Assumption B, let $(r_1, \ldots, r_k)$ be the first k MWMR registers written in the pseudo-solo run segments of processes in $G_{k,1}$.
(3) Select any process $p_1 \in G_{k,1}$ and a pseudo-solo run segment $\tilde{} \; solo_{p_1}$ starting at $s \cdot \beta_{k,1}$.
(4) If there are less than $k+1$ write operations to MWMR registers in $\tilde{} \; solo_{p_1}$ (until p_1 owns the WT&S object), then we are done and $\beta_j = \beta_{k,1}$ for $j = k, k+1, \ldots, i$ (i.e. if p_1 does not have more than k write operations to MWMR registers, none of the processes in $G_{k,1}$ does, since by Assumption B they all write to the same MWMR registers in the same order).
(5) Otherwise, p_1 has at least $k + 1$ write operations to MWMR registers in $\tilde{} \; solo_{p_1}$, before owning the WT&S object. Let $W_{p_1,k+1}$ be the $k + 1$st write to a MWMR register by p_1. Let r_{k+1} be the register that process p_1 writes in $W_{p_1,k+1}$, and let $\tilde{} \; solo_{p_1}(k + 1)$ be the longest prefix of $\tilde{} \; solo_{p_1}$ that does not include $W_{p_1,k+1}$. Any pseudo-solo run segment that starts after $s \cdot \beta_{k,1} \cdot \tilde{} \; solo_{p_1}(k + 1)$ can read p_1's write operations to $r_1, \ldots, r_k$ and therefore the state is not transparent.
(6) We now erase all the traces of p_1: we enable each of the processes $q_1, \ldots, q_k$ covering $r_1, \ldots, r_k$ respectively to write in $r_1, \ldots, r_k$ and to run until they do not access the WT&S object anymore. This run segment is denoted as *clean(k)*. Such an extension to $s \cdot \beta_{k,1} \cdot \tilde{} \; solo_{p_1}(k + 1)$ exists, since W is uniformly wait-free WT&S implementation. Hence each process $q_1, \ldots, q_k$ reaches this state after a finite number of steps. (Note that at this stage processes $q_1, \ldots, q_k$ do not necessarily write to only the first i registers of shared memory anymore, that is to MWMR registers with an index of at most i. However this will not affect the remaining construction since we will let them run until they terminate their executions.) Let $\alpha_{k+1,1} = \beta_{k,1} \cdot \tilde{} \; solo_{p_1}(k + 1) \cdot$ *clean(k)*. By the end of run $s \cdot \alpha_{k+1,1}$ the state is transparent with respect to $G_{k,1} - \{p_1\}$. We need to use Assumption A here, that writes are only enabled to MWMR registers but not to SWMR registers. At the end of run $s \cdot \alpha_{k+1,1}$ only one process p_1 is enabled to write in r_{k+1} while the registers $r_1, \ldots, r_k$ are not necessarily covered anymore.
(7) Since the state at $s \cdot \alpha_{k+1,1}$ is transparent with respect to $G_{k,1} - \{p_1\}$, we can again activate the induction hypothesis. Hence, a cover of a sequence of k MWMR registers $r_1, \ldots, r_k$ is constructed by activating a run segment $\beta_{k,2}$ s.t., the state at the end of $s \cdot \alpha_{k+1,1} \cdot \beta_{k,2}$ is transparent to a set of processes $G_{k,2} \subseteq G_{k,1} - \{p_1\}$.
(8) Let $r'_1, \ldots, r'_k, r'_{k+1}$ be the first $k + 1$ registers written in the pseudo-solo segments of the processes in $G_{k,2}$. Assume that $r'_{k+1} = r_{k+1}$. In this case, we are done — we have a cover on the first $k + 1$ registers processes in $G_{k,2}$ will write to.
(9) If $r'_{k+1} \neq r_{k+1}$ we construct the required cover on r'_{k+1} by repeating the construction above but starting at $s \cdot \alpha_{k+1,1} \cdot \beta_{k,2}$. We now have a cover on r_{k+1} and r'_{k+1} by two different processes. We continue this construction until either we finish in step 4 or step 8 or until all i registers at the beginning of shared memory are covered.

Note that $G_0 = G$ and hence $|G_0| = |G| = \infty$. Since at each step of this inductive construction at most *finitely many* processes are removed from $G_{k,l}$ it follows trivially that $|G_{k,l}| = \infty$ and $|G_{k+1}| = \infty > e$. Hence, the inductive claim follows. □

Phase 2:
We now relax Assumption A. To do this we use techniques developed in [5]. The run constructed in the previous lemma may no longer be valid, as processes are allowed to write to their SWMR registers. The argument presented above may collapse in one of the following two ways:

(1) The participating processes in any *clean* run segment may read the SWMR registers of other active processes. In particular, they may read the SWMR register of the processes whose traces their writes are supposed to eliminate. They would then leave the system in a non-transparent state by writing about the value they read.
(2) After a *clean* run segment, a process q might start its q-segment execution and may read the SWMR register of another concurrently active process p. Hence, q will not perform a pseudo-solo run anymore, that is it may write to a MWMR register with an index greater than i and it may stop without covering the MWMR registers. Moreover, q may decide "on the spot" to write into MWMR registers different from what the "original" construction (Proof of Lemma 1.2) required and from what we had assumed.

As in [5] or [12], we will avoid the two *dangerous* situations by not allowing processes, whose SWMR registers are later read to take part in the constructed run. So, if in any given state in the run, if process q reads the SWMR register of process p and p is active, we construct another run in which p is replaced by another process p'. Process q will still read the same SWMR registers. The behavior of p and p' is in some sense "equivalent". They both write and cover the same MWMR registers. All we need to do is to show that a process like p' always exists since (1): there is a large enough set of processes to select p' from s.t., p' did not participate in the run before and has the same general properties as p. We can do this since at any give point in time at most finitely many processes participate in the execution while infinitely many processes are available.

(2): Process q can perform only a constant number of read operations, since the number of concurrently active processes in the run is a function of i and since the algorithm is uniformly wait-free.

We maintain a large enough set of 'equivalent' runs, which allows us to replace at any point in time at which we fail to reach a transparent state. This set will shrink as the construction progresses.

Definition 1.1. Two runs β and β' are equivalent with respect to a set of processes G if (1) the state at the end of both runs β and β' is transparent with respect to G, (2) the sets of MWMR registers covered in β and β' are the same, and (3) if process p participates in both β and β' then p cannot distinguish between the two runs.

In our construction, whenever a process p that was previously selected to participate in the run is discovered by a covering process, we need to replace it with some other process p' that cannot be discovered. We achieve this by considering an equivalent run in which p' takes steps instead of p.

Note that we also need to modify the proof of the Main Theorem, Theorem 1 in the same way as the proof of the Main Lemma 1.2. Given the proof of the modified lemma this modification is straightforward and we leave the details to the reader.

We now restate the central inductive lemma as follows:

Lemma 1.3. *Let W be a long-lived, uniformly wait-free, i-solo-memory- adaptive WT&S algorithm. Then, for every j, $0 \leq j \leq i$ and for every constant e and for any set of processes G, $|G| = \infty$ and for any state s transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds:* (1) *$|G_j| \geq e$,* (2) *the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and* (3) *there exists a set $R_i = \{r_1, \ldots, r_j\}$ of MWMR registers that are covered in $s \cdot \beta_j$ and $r_1, \ldots, r_j$ are the first j MWMR registers written (in this order) in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta$, or, the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to a set $R_{j'} \subset R_j$ (in the same order) of MWMR registers, where $R'_j = \{r_1, \ldots, r_{j'}\}$, $1 \leq j' < j$ and all these writes are covered in $s \cdot \beta_j$. And in addition, there is a large enough set of runs equivalent to β_j with respect to a large enough set $G'_j \subseteq G_j$.*

Proof. The new proof closely follows the Proof of Lemma 1.2.

We now, however, need to construct a set of runs equivalent to β_{j+1} s.t., each run is equivalent with respect to a sub-set of the set of processes G_{j+1}. Unless stated otherwise, we use the same notations and symbols as before.

As explained above, we need to address the following two cases:

(1) After the run $s \cdot \beta_{k,1} \cdot\ solo_{p_1}(k+1) \cdot clean(k)$, if any of the processes participating in *clean(k)* $(q_1, \ldots, q_k)$ reads from the SWMR register $R_{SW_{p_1}}$, which p_1 wrote, then the state is not transparent with respect to $G_{k,1}$. The algorithm W, however, is uniformly wait-free and hence the number of steps in *clean(k)* is a constant. Hence, the total number of SWMR registers that may be read by $q_1, \ldots, q_k$ during any *clean(k)* segment is bounded by some constant, say l. Therefore, because of the one-to-one correspondence between SWMR registers and processes, at most l processes may be discovered during a *clean(k)* segment. So removing these processes from $G_{k,1}$ leaves us with a large enough set $G_{invis,k,1} \subseteq G_{k,1}$, where $G_{invis,k,1}$ is the subset of processes in $G_{k,1}$ whose SWMR register is not read throughout the construction. Clearly $|G_{invis,k,1}| = \infty$. Note that, based on the construction this step may repeat $f(i)$ (where $f(i) < \infty$) many times (whenever a cover is released), each time removing at most finitely many processes from $G_{invis,k,l}$. Hence, even after removing these processes from the construction, infinitely many unused processes from which we may choose remain.

We note that $q_1, \ldots, q_k$ are allowed to read each others SWMR registers, but that the state at the end of the run is still transparent with respect to $G_{invis,k,1}$.

(2) The state after some *clean(k)* run segment may not be transparent with respect to processes that start their execution after *clean(k)*. In particular, one of the processes executing a pseudo-solo run might read the SWMR register of a currently active process (covering a register). Hence, this process might not run its pseudo-solo run segment after *clean(k)*.

We again argue that a run in which all such "dangerous" processes are removed and that is equivalent to the previous run exists. We select processes from the set of currently available processes $G_{invis,k,1}$. We select a subset $G^*_{k,1}$ of $G_{invis,k,1}$. This subset is constructed as follows: first, after any transparent state, the processes perform at most a constant number m of operations when running a pseudo-solo run segment. Furthermore, the length of the run segment $\beta^*_{k,1}$ ($\beta_{k,1}$ previously) is bounded by m and i and it also bounds the number of processes that are writing into SWMR registers in $\beta^*_{k,1}$. We remove processes from $G_{invis,k,1}$ whose SWMR register are read by other processes. We call the resulting set $G^*_{invis,k,1}$.

It remains to show that $|G^*_{invis,k,1}| = \infty$. In other words we must show that there exists a finite set S (of sufficient size) of processes so that none of the remaining infinitely many processes in $G^*_{invis,k,1}$ will read any of the SWMR registers of the processes in S. We will show this by proving a combinatorial lemma.

Lemma 1.4. *There exists a finite set of integers S, where $|S| \geq m$ for some constant m, such that for the collection of uniformly bounded finite sets of integers $\{S_i\}_{1 \leq i \leq \infty}$, $|S_i| \leq r$ there exists an infinite set X such that $\forall x \in X$, $S_x \cap S = \emptyset$.*

Proof. The proof is by contradiction. Assume not. Then, for all finite sets of integers S ($|S| \geq m$), $\exists a_S \in S$ such that a_S is an element of all but finitely many S_i. Hence, if there are infinitely many *distinct* such a_S all but finitely S_i are unbounded, a contradiction. Therefore, there are at most finitely many such a_S.

We now claim that we can one-by-one replace all of these finitely many a_S with a_S' such that a_S' is in at most finitely many S_i. So assume this is not possible, i.e. that when looking for a replacement for a_S we never succeed. Then there exist infinitely many distinct ("replacements") such that each of them is in all but finitely many S_i, making infinitely many S_i unbounded, a contradiction. □

We apply this lemma now to our construction. It shows that it is possible for us to replace processes that cover registers and are later discovered with other, equivalent processes such that we preserve an infinite set of processes $G^*_{invis,k,1}$ for whom the state at the beginning of their pseudo-solo-runs is transparent.

With these changes applied to Step 6 of the Proof of Lemma 1.2, $G^*_{invis,k,1}$ replacing $G_{k,1}$ we are able to finish the proof of Lemma 1.3. □

Phase 3:

It remains to remove Assumption B. During a WT&SET operation processes are now allowed to write to different MWMR registers in different orders. This means that the cover we constructed earlier might not be on the "right" registers anymore since two processes p and q may write into the MWMR registers in different orders.

To overcome this difficulty, we first recall that we are only interested in pseudo solo runs. We know, however, that processes executing such runs are only allowed to write to the first i MWMR registers in shared memory. Hence, in pseudo-solo runs the number of MWMR under consideration is a constant. Second we recall that our algorithm is uniformly wait-free that is the length of every pseudo solo run is a constant. Hence we can consider the different sequences of write operations to MWMR registers by the different pseudo-solo run segments of processes in G. The number of these sequences is bounded by i and m where m is the uniform bound on the length of a solo execution. Each such sequence defines an equivalence class in G. Since G is infinite, we can always find a subset of processes that in pseudo solo runs performs the same sequence of writes to MWMR registers starting at s.

But, since in two different states s and s' that are transparent with respect to G the sequence of MWMR registers that processes in G write to in pseudo-solo runs need not be the same, the required subset of processes cannot be computed in advance. Instead it is computed iteratively in rounds as in [5].

We restate the main inductive lemma with Assumption B removed. □

Lemma 1.5. *Let W be a long-lived, uniformly wait-free, i-solo-memory- adaptive WT&S algorithm. Then, for every j, $0 \leq j \leq i$ and for every constant e, for any set of processes G, $|G| = \infty$ and for any state s transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds:* (1) *$|G_j| \geq e$,* (2) *the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and* (3) *either the first j MWMR registers written in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta_j$, are the same and covered in $s \cdot \beta_j$, or the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to the same MWMR registers and all these writes are covered in $s \cdot \beta_j$. And in addition, there is a large enough set of runs equivalent to β_j with respect to a large enough set $G_j' \subseteq G_j$.*

Proof. The proof is similar to the proof in [5].

We will show that if a sufficiently large set of processes writes to less than $k+1$ MWMR registers in their pseudo-solo run segments then there is sufficiently large subset of this set of processes so that all processes in this subset write to the same k covered MWMR registers in the same order. Otherwise, we will argue that a set of $k+1$ covered MWMR registers and a sufficiently large set of processes must exist such that all processes in this set write to the same $k+1$ covered MWMR registers in the same order.

Instead of using Assumption B in the proof of the inductive lemma we will now use a new property that we call *multi-unique* [5]. In step 3 of the proof we reduce the set $G_{k,1}$ to a subset $G'_{k,1}$. $G'_{k,1}$ is selected so that all processes in $G'_{k,1}$ write to the same MWMR registers in the same order when executing their pseudo-solo run segments after $s \cdot \beta_{k,1}$.

$G'_{k,1}$ is constructed as follows: In each iteration of steps 3–7 the corresponding set $G'_{k,1}$ is constructed in several rounds. In each new round a subset of the subset selected in the previous round is selected. Let $S_0 = G_{j,1}$ and let the set of processes after round h be $S_h \subseteq S_0$. In round $h+1$ we simply observe the $h+1$'st write operations of the processes in S_h and we construct S_{h+1} as follows:

(1) If infinitely many of the processes in S_h do not have an $h+1$'st write to MWMR registers than let S_{h+1} be this infinite subset of the processes in S_h. S_{h+1} is the set of processes that holds the multi-unique property at $s \cdot \beta_{k,1}$.
(2) Else we denote the (infinitely many) remaining processes (writing to $h+1$ registers) by S'_{h+1} and $|S'_{h+1}| = \infty$. Hence there is at least one MWMR register that a set of infinitely many processes memory-adaptively is writing to.

Since there are at most $i < \infty$ rounds, then $|S_i| = \infty$. Now let $G'_{k,1} = S_i$ hence $|G'_{k,1}| = \infty$. □

4. Uniform memory-adaptive algorithms for NAD's

We will now discuss what our results imply for the design of memory adaptive algorithms (e.g. store/release) for NAD's. Earlier in this paper, we showed that there is no uniformly wait-free, uniform store/release protocol memory-adaptive to interval contention that uses only read/write registers. In [12] it was shown that one cannot uniformly implement a

MWMR register on a NAD with a finite number of fail-prone base registers, even if the implementation need not be wait-free. This implies the need for infinitely many base registers. Since this is however an unrealistic assumption, memory-adaptive algorithms are of particular practical interest in the uniform setting on NAD's. If uniform protocols that require infinitely many base registers and that run on NAD are memory-adaptive they will remain practical since they will allow us to efficiently bound the memory requirements based on the contention. Hence memory-adaptive algorithms are not only attractive but essential for uniform algorithms on NAD's.

In [31] we provided a uniform memory-adaptive to interval contention implementation of store/release using stronger primitives namely an operation we called *write-plus* which is weaker than the standard read-modify-write.

The write-plus command is equivalent to specifying that the function f in the definition of read-modify-write (see the model section) is required to be a constant independent of X (the value read).

Active Disks [36] on the other hand are capable of supporting stronger semantics that are not normally provided by disk drives. In particular, they can provide read-modify-write operations. Our results imply that to run realistic uniform algorithms on a NAD – that is algorithms that are memory adaptive to interval contention – read/write registers are not sufficient. Our results justify the use of Active Disks in the uniform setting. We will now show how to implement memory adaptive to interval contention store/release on active disks if disks and hence registers may fail.

Our protocol transfers the uniformly wait free store/release protocol [31] that uses only read, write and read-modify-write operations and that is memory-adaptive to interval contention and time adaptive to total contention.

Protocol 4: We assume that memory is arranged in the form of a two-dimensional grid, this time indexed by $\mathcal{N} \times \mathcal{N}$. Whenever a process executes a read-modify-write into shared memory, it keeps a copy of what was previously written there in its private memory space along with whatever it writes into the register. As a result, the process always has a complete record of all of its operations starting from the beginning till the current time in its private space along with the values that it overwrites. During each store and with each write, the process keeps track of the number of times it has stored a value in shared memory. Each write will contain a field with this parameter. The algorithm uses splitters [42]. We assume that splitters are able to hold values. Each process when executing the algorithm attempts to capture a splitter so that it can store its value in this splitter.

Using these assumptions, we showed in [31] that a process has the ability to tell whether a splitter is "clean" or "dirty". In other words, the process is able to tell whether, given a splitter, there exists another process that has previously written into the splitter's slot #1 and yet has not either written into slot #2 or written into some other shared register. Based on these processes execute the following protocol: whenever a process executes a store, it begins at splitter $(1, 1) = (i, j)$. If the splitter is taken with a value, then the process moves to $(i + 1, 1)$. If the splitter is dirty, it moves to $(i, j + 1)$. If the splitter is clean, it competes. It writes his name into slot #1 and checks slot #2. If there is a "new" name (i.e. a name that has been written in the splitter after the process started competing) in slot #2, the process moves to $(i+1, 1)$. If there is no new name, then the process writes its name into slot #2 and checks slot #1. If there is a new name in slot #1, then the process moves to $(i, j + 1)$. If the process's name is still written in slot #1, then the process has won the splitter and the right to use its value register. It notes this in the register and writes its value.

In order to execute a release, the process simply indicates that the splitter is now clean.

With Protocol 4 we easily obtain the following Theorem.

Theorem 2. *There exists a long-lived, uniformly wait-free and uniform store/ release protocol for Active Disks using only the basic operations read, write, and read-modify-write that is memory-adaptive to interval contention, time-adaptive to total contention.*

Proof. We simply transfer Protocol 4 to Active Disks. To do so we use Active Disks that provide Read-Modify-Write registers. Active Disks however may fail. So to make this algorithm fault-tolerant assuming that at most t disks may fail we simply let each process execute a store on $2t + 1$ active disks. Each process is guaranteed to receive responses from a majority of disks so it suffices to wait for these responses when executing either store or release.

It remains to show that Protocol 4 is memory-adaptive to interval contention and time-adaptive to total contention:

We assume that the total number of processes that ever become active during the interval under consideration is k. Consider the protocol restricted to a given column i. Assume that only at most j active processes are ever in the column at any given time. Because all processes are required to start at position $(i, 1)$, note that the furthest downward that any process could ever move is (i, j). We now claim that in column i, at most $k - i + 1$ processes will ever exist in the column at any given time. Clearly, this is true for $i = 1$. In order for a process to move to the right, another process must be "left behind". Thus, there can be at most one fewer active process in a given column than in the column to the left. Inductively, our claim is therefore proved. The processes are therefore restricted to move in the space above and to the left of the grid points $(i, k - i + 1)$, $1 \leq i \leq k$. Because the processes are only allowed to move right and down, this restricts the number of moves to at most k before an empty splitter is found. This implies a bound of $O(k^2)$ on the memory-adaptiveness of the protocol.

For the time-adaptive claim, note that the number of reads that is necessary for a given process to determine the history of a given splitter is proportional to the number of processes that has previously written into the splitter. □

5. Conclusion and open problems

In this paper, we showed that there are no uniform, memory-adaptive to interval contention store/release protocols that use only read/write registers. This proves that to implement protocols that are memory-adaptive to interval contention

in this setting we must use stronger primitives, such as read-modify-write registers, validating our uniform and memory adaptive to interval contention protocol from [31]. We furthermore showed that Active Disks are an ideal environment for the employment of such a protocol. It would be interesting to investigate closer the relationship between time-adaptive and memory-adaptive protocols. What conditions must be met for a memory-adaptive protocol to be also time-adaptive? How about the reverse? Answering these questions will allow us to better understand the true cost of distributed protocols and if and when they can be made adaptive.

References

[1] I. Abraham, G. Chockler, I. Keidar, D. Malkhi, Byzantine Disk Paxos: Optimal resilience with byzantine shared memory, in: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, 2004, pp. 226–235.
[2] A. Acharya, M. Uysal, J. Saltz, Active disks: Programming model, algorithms and evaluation, in: Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VIII, 1998, pp. 81–91.
[3] Y. Afek, H. Attiya, A. Fouren, G. Stupp, D. Touitou, Long-lived renaming made adaptive, in: Proc. of 18th Annual ACM Symp. on Principles of Distributed Computing, May 1999, pp. 91–103.
[4] Y. Afek, H. Attiya, G. Stupp, D. Touitou, Adaptive long-lived renaming using bounded memory, in: Proc. of the 40th IEEE Ann. Symp. on Foundations of Computer Science, October 1999, pp. 262–272.
[5] Y. Afek, P. Boxer, D. Touitou, Bounds on the shared memory requirements for long-lived and adaptive objects, in: Proc. of the 19th Ann. ACM Symp. on Principles of Distributed Computing, PODC 2000, July 2000, pp. 81–89.
[6] Y. Afek, D. Dauber, D. Touitou, Wait-free made fast, in: Proc. of the 27th Ann. ACM Symp. on Theory of Computing, May 1995, pp. 538–547.
[7] Y. Afek, M. Merritt, Fast, wait-free $(2k-1)$-renaming, in: Proc. of the 18th Ann. ACM Symp. on Principles of Distributed Computing, May 1999, pp. 105–112.
[8] Y. Afek, M. Merritt, G. Taubenfeld, D. Touitou, Disentangling multi-object operations, in: Proc. of 16th Annual ACM Symp. on Principles of Distributed Computing, August 1997, pp. 111–120.
[9] Y. Afek, G. Stupp, D. Touitou, Long-lived adaptive collect with applications, in: Proc. of the 40th Ann. Symp. on Foundations of Computer Science, October 1999, pp. 262–272.
[10] Y. Afek, G. Stupp, D. Touitou, Long lived adaptive splitter and applications, Distributed Computing 15 (2) (2002) 67–86.
[11] Y. Afek, G. Stupp, D. Touitou, Long lived and adaptive atomic snapshot and immediate snapshot, in: Proc. of the 19th Ann. ACM Symp. on Principles of Distributed Computing, 2000, pp. 71–80.
[12] M. Aguilera, B. Englert, E. Gafni, Uniform Solvability with a finite number of MWMR registers, in: Proc. 17th International Conference DISC 2003, October 2003, pp. 16–30.
[13] K. Amiri, G.A. Gibson, R. Golding, Highly concurrent shared storage, in: Proceedings of the International Conference on Distributed Computing Systems, ICDCS 2000, 2000, pp. 298–307.
[14] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang, Serverless network file systems, ACM Transactions on Computer Systems 14 (1) (1996) 41–79.
[15] J. Anderson, J.-H. Yang, Time/contention trade-offs for multiprocessor synchronization, Information and Computation 124 (1) (1996) 68–84.
[16] J. Anderson, Y.-J. Kim, Adaptive mutual exclusion with local spinning, in: Proceedings of the 14th International Conference, DISC 2000, October 2000, pp. 29–43.
[17] D. Angluin, Local and global properties in networks of processes, in: Proceedings of the 12th ACM Symposium on Theory of Computing, STOC 1980, pp. 82–93.
[18] J. Aspnes, G. Shah, J. Shah, Wait free consensus with infinite arrivals, in: Proc. of the 34th Annual ACM Symposium on the Theory of Computing, May 2002, pp. 524–533.
[19] H. Attiya, V. Bortnikov, Adaptive and efficient mutual exclusion, in: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, PODC 2000, 2000, pp. 91–100.
[20] H. Attiya, E. Dagan, Universal operations: Unary versus binary, in: Proc. 15th Annual ACM Symp. on Principles of Distributed Computing, May 1996, pp. 223–232.
[21] H. Attiya, F. Fich, Y. Kaplan, Lower bounds for adaptive collect and related objects, in: Proc. 23 Annual ACM Symp. on Principles of Distributed Computing, July 2004, pp. 60–70.
[22] H. Attiya, A. Fouren, Adaptive wait-free algorithms for lattice agreement and renaming, in: Proc. 17th Annual ACM Symp. on Principles of Distributed Computing, June 1998, pp. 277–286.
[23] H. Attiya, A. Fouren, Algorithms adaptive to point contention, Journal of the ACM 50 (4) (July 2003) 444–468.
[24] H. Attiya, A. Fouren, E. Gafni, An adaptive collect algorithm with applications, Distributed Computing 15 (2) (2002) 87–96.
[25] H. Attiya, F. Kuhn, M. Wattenhofer, R. Wattenhofer, Efficient adaptive collect using randomization, in: Proc. 18th Annual Conference on Distributed Computing, DISC, October 2004.
[26] H. Attiya, I. Zach, Fully adaptive algorithms for atomic and immediate snapshots, 2003. www.cs.technion.ac.il/~hagit/pubs/AZ03.pdf.
[27] R. Burns, Data managment in a distributed file system for storage area networks, Ph.D. Thesis, Department of Computer Science, University of California Santa Cruz, 2000.
[28] J. Burns, N. Lynch, Bounds on shared memory for mutual exclusion, Information and Computation 107 (2) (1993) 171–184.
[29] G. Chockler, D. Malkhi, Active Disk Paxos with infinitely many processes, in: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, 2002, pp. 78–87.
[30] M. Choy, A.K. Singh, Adaptive solutions to the mutual exclusion problem, Distributed Computing 8 (1) (1994) 1–17.
[31] B. Englert, D. Goldstein, Can memory be used adaptively by uniform algorithms? in: Proc. 9th International Conference on Principles of Distributed Systems, OPODIS, 2005, pp. 25–35.
[32] F. Fich, E. Ruppert, Hundreds of impossibility results for distributed computing, Distributed Computing 16 (2–3) (2003) 121–163.
[33] E. Gafni, A simple algorithmic characterization of uniform solvability, in: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2002, 2002, pp. 228–237.
[34] E. Gafni, L. Lamport, Disk Paxos, Distributed Computing 16 (1) (2003) 1–20.
[35] E. Gafni, M. Merritt, G. Taubenfeld, The concurrency hierarchy and algorithms for unbounded concurrency, in: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, 2001, pp. 161–169.
[36] G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka, A cost-effective high-bandwidth storage architecture, in: ACM SIGOPS Operating Systems Review, 1998, pp. 92–103.
[37] S. Hotz, R. Van Meter, G. Finn, Internet protocols for network-attached peripherals, in: Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Conjunction with 15th IEEE Symposium on Mass Storage Systems, 1998.
[38] M. Inoue, S. Umetani, T. Masuzawa, H. Fujiwara, Adaptive long-lived $O(k^2)$ renaming with $O(k^2)$ steps, in: Proceedings of the 15th International Conference on Distributed Computing, DISC 2001, 2001, pp. 123–135.
[39] A. Itai, M. Rodeh, Symmetry breaking in distributed networks, Information and Computation 88 (1) (1990) 60–87.
[40] L. Lamport, A fast mutual exclusion algorithm, ACM Transactions on Computer Systems 5 (1) (1987) 1–11.

[41] M. Merritt, G. Taubenfeld, Speeding Lamport's fast mutual exclusion algorithm, Information Processing Letters 45 (1993) 137–142.
[42] M. Moir, J. Anderson, Wait-free algorithms for fast, long-lived renaming, Science of Computer Programming 25 (1) (1995) 1–39.
[43] National Storage Industry Consortium. http://www.nsic.org/nasd.
[44] E. Riedel, C. Faloutsos, G.A. Gibson, D. Nagle, Active Disks for large scale data processing, IEEE Computer (June) (2001).
[45] Chandramohan Thekkath, Timothy Mann, Edward K. Lee, Frangipani: A scalable distributed file system, in: Proceedings of the 16th ACM Symposium on Operating System Principles, ACM Press, New York, 1997, pp. 224–237.