

Time and Space Lower Bounds for Implementations Using k -CAS*

Hagit Attiya[†]

Danny Hendler[‡]

September 12, 2006

Abstract

This paper presents lower bounds on the time- and space-complexity of implementations that use the k compare-and-swap (k -CAS) synchronization primitives. We prove that the use of k -CAS primitives cannot improve neither the time- nor the space-complexity of implementations of widely-used concurrent objects, such as counter, stack, queue, and collect. Surprisingly, the use of k -CAS may even *increase* the space complexity required by such implementations.

We prove that the worst-case *average* number of steps performed by processes for every n -process implementation of a counter, stack or queue object is $\Omega(\log_{k+1} n)$, even if the implementation can use j -CAS for $j \leq k$. This bound holds even if a k -CAS operation is allowed to *read* the k values of the objects it accesses and return these values to the calling process.

We also consider more realistic *non-reading* k -CAS primitives. An operation of a non-reading k -CAS primitive is only allowed to return a success/failure indication. For implementations of the *collect* object that use such primitives, we prove that the worst-case average number of steps performed by processes is $\Omega(\log_2 n)$, regardless of the value of k . This implies a *round complexity* lower bound of $\Omega(\log_2 n)$ for such implementations. As there is an $O(\log_2 n)$ round complexity implementation of collect that uses only reads and writes, these results establish that non-reading k -CAS is no stronger than read and write for collect implementation round complexity.

We also prove that k -CAS does not improve the space complexity of implementing many objects (including counter, stack, queue, and single-writer snapshot). An implementation has to use at least n base objects even if k -CAS is allowed, and if all operations (other than read) swap exactly k base objects, then the space complexity must be at least $k \cdot n$.

1 Introduction

Lock-free implementations of concurrent objects require processes to coordinate without relying on mutual exclusion, thus avoiding the inherent problems of *locking*, e.g., deadlock, convoying, and priority-inversion.

* A preliminary version of this paper appeared in the proceedings of the *19th International Conference on Distributed Computing (DISC 2005)*, pp. 169-183.

[†]Department of Computer Science, Technion; supported by the *Israel Science Foundation* (grant number 953/06).

[‡]Department of Computer Science, Ben-Gurion University. This work was written while the second author was a postdoc fellow in the department of computer science of the university of Toronto, and in the faculty of industrial engineering and management of the Technion.

Synchronization primitives are often evaluated according to their power to implement other objects in a lock-free manner. A *conditional* synchronization primitive may modify the value of the object to which it is applied only if the object has a specific value. The *compare-and-swap* synchronization primitive (abbreviated CAS) is an example of a conditional primitive: $CAS(O, old, new)$ changes the value of an object O to new only if its value just before CAS is applied is old ; otherwise, CAS does not change the value of O .

CAS can be used, together with read and write, to implement any object in a deterministic wait-free manner [22]. It has consequently become a synchronization primitive of choice, and hardware support for it is provided in many multiprocessor architectures [24, 28, 30].

In recent years, the question of supporting multi-object conditionals in hardware has been deliberated in both industrial and academic circles [14, 18, 19, 21]. The design of concurrent data structures seems to be easier if conditional primitives can be applied to multiple objects [19]. On the other hand, almost all of the current architectures support conditional primitives only on a single object.

In the context of this debate, it is natural to ask whether multi-object conditionals admit more efficient implementations. Of concrete interest are k -CAS synchronization primitives that atomically check and possibly modify several objects; when $k = 2$, this is the familiar *double compare&swap* (DCAS) primitive.

In this paper, we prove lower bounds on the time- and space-complexity of implementations of widely-used objects that use multi-object conditional primitives such as k -CAS. We show that the use of such primitives does not improve neither the time- nor the space-complexity of implementing these objects.

We start by proving that the worst-case *average* number of steps performed by processes in *solo-terminating*¹ implementations of counters, stacks and queues is $\Omega(\log_{k+1} n)$, assuming the implementation uses only j -word conditionals for $j \leq k$, read and write. This extends a *worst-case* lower bound of $\Omega(\log_2 n)$ on the number of steps needed for implementing these objects using unary conditionals [25]. Both lower bounds hold even when implementations can use *reading* conditional primitives, which read and return the values of all the objects they access. As an example, a reading DCAS operation returns the values of the two objects it accesses just before it is applied.

At this point, it is natural to question the validity of charging only a single unit for a reading k -CAS operation, that compares, possibly swaps, and returns k values. For the purpose of proving lower bounds, it is easier to state what *cannot* be done in one step, rather than to stipulate the correct price tag for a reading k -CAS operation. It is clearly overly optimistic to assume that k values can be read in one step, and thus, we investigate also *non-reading* k -CAS primitives that only return a boolean success indication.

For the weaker wake-up problem [17], even a non-reading k -CAS primitive is more powerful than 1-CAS. The algorithm of Afek et al. [1] can be adapted to yield an $O(\log_{k+1} n)$ worst-case step implementation, whereas [25] proves a worst-case lower bound of $\Omega(\log_2 n)$ steps on implementations of wake-up that can use unary conditional primitives.

Interestingly, there exist widely-used objects such as *collect*, for which non-reading k -CAS is no stronger than read and write. We prove that the worst-case average time complexity of solo-terminating implementations of *collect* is $\Omega(\log_2 n)$, even if k -CAS primitives can be used, for any k . The proof hinges on the fact that a non-reading k -CAS operation only tells us whether the values of the k objects to which it is applied

¹Solo termination [15] requires a process running alone to complete its operation within a finite number of its steps; it is the safety property provided by *obstruction-free* implementations [23]. Clearly, lower bounds on solo-terminating implementations hold also for implementations with stronger liveness conditions, such as wait-freedom [22].

equal a *particular* vector of values or not. Thus, such an operation provides only a single bit of information about the objects it accesses. This intuition is captured, in a precise sense, by adapting a technique of Beame [12], originally applied in the synchronous CRCW PRAM model. This implies that the *round complexity* [13] of such implementations is $\Omega(\log_2 n)$, matching an $O(\log_2 n)$ round complexity implementation of collect using read and write, given by Attiya, Lynch, and Shavit [10].

Finally, we turn to study the space complexity of implementations that use multi-object conditional primitives. We extend a result of Fich, Hendler and Shavit [16], who show a linear space lower bound on implementations that use read, write, and unary conditional primitives. They prove this bound for wait-free implementations of many widely-used concurrent objects, such as stack, queue, counter, and single-writer snapshot. We show that an implementation cannot escape this lower bound by using multi-object conditional primitives. Moreover, if all operations (other than read) swap exactly k locations, then the space complexity is at least $k \cdot n$.

Our results indicate that supporting multi-object conditional primitives in hardware may not yield performance gains: under reasonable cost metrics, they do not improve the efficiency of implementing many widely-used objects.

Several shared object implementations use k -CAS, most often DCAS, to simplify design (e.g. [6, 20]). Doherty et al. [14] argue that in certain cases, e.g., for implementing double-ended queues, even DCAS does not suffice and that simple and easy-to-prove implementations should rely on 3-CAS. There is a variety of algorithms for simulating multi-object k -CAS (and other objects) from single-object CAS, load-linked and store-conditional (e.g. [3, 7, 8, 11, 29]). A few papers investigate the consensus number of multi-object operations [2, 26]. Attiya and Dagan [8] prove that any implementation of two-object conditional primitives from unary conditional primitives requires $\Omega(\log \log^* n)$ steps. The *k-compare-single-swap* synchronization primitive of [27] is a weaker variant of non-reading k -CAS, and all our lower bounds hold for it.

2 The Shared Memory System Model

We consider a standard model of an asynchronous shared memory system, in which processes communicate by applying operations to shared objects. An *object* is an instance of an abstract data type. It is characterized by a domain of possible values and by a set of *operations* that provide the only means to manipulate it. No bound is assumed on the size of an object (i.e., the number of different possible values the object can have). An *implementation* of an object shared by a set $\mathbf{P} = \{p_1, \dots, p_n\}$ of n processes provides a specific data-representation for the object from a set \mathbf{B} of shared *base objects*, each of which is assigned an initial value, and algorithms for each process in \mathbf{P} to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects *primitives* and reserve the term *operations* for the objects being implemented.

A *wait-free* implementation of a concurrent object guarantees that any process can complete an operation in a finite number of its own steps. A *solo-terminating* implementation guarantees only that if a process eventually runs by itself while executing an operation then it completes that operation within a finite number of its own steps. Each *step* consists of some local computation and one shared memory *event*, which is a primitive applied to a vector of objects in \mathbf{B} . We say that the event *accesses* these base objects and that it *applies* the primitive to them. In this extended abstract we consider only *deterministic implementations*, in

which the next step taken by a process depends only on its state and the response it receives from the event it applies.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* is an execution fragment that starts from an *initial configuration*. This is a configuration in which all base objects in \mathbf{B} have their initial values and all processes are in their initial states. If $o \in \mathbf{B}$ is a base object and E is a finite execution, then $value(E, o)$ denotes the value of o at the end of E . If no event in E changes the value of o , then $value(E, o)$ is the initial value of o . In other words, in the configuration resulting from executing E , each base object $o \in \mathbf{B}$ has value $value(E, o)$. For any finite execution fragment E and any execution fragment E' , the execution fragment EE' denotes the concatenation of E and E' . Let E and F be two executions. We say that F is an *extension of E* if $F = EE'$ for some execution fragment E' .

An *operation instance*, $\Phi = (O, Op, p, args)$, is an application by process p of operation Op with arguments $args$ to object O . In an execution, processes apply their operation instances to the implemented object. To apply an operation instance Φ , a process *issues* a sequence of one or more events that access the base objects used by the implementation of O . If the first event of Φ has been issued in an execution E , we say that Φ *occurs in E* . If the last event of Φ has been issued in, we say that Φ *completes in E* . The events of an operation instance issued by a process can be interleaved with events issued by other processes. Let Φ_1, Φ_2 be two operation instances that occur in E . If Φ_1 completes in E before the first event of Φ_2 has been issued in E , we say that Φ_1 *precedes Φ_2 in E* and denote that by $\Phi_1 \stackrel{E}{\prec} \Phi_2$. We note that the relation $\stackrel{E}{\prec}$ is a partial order on the operation instances that occur in E .

If a process has not completed its operation instance, it has exactly one *enabled* event, which is the next event it will perform, as specified by the algorithm it is using to apply its operation instance to the implemented object. We say that a process p is *active* after E if p has not completed its operation instance in E . If p is not active after E , we say that p is *idle* after E . We say that an execution E is *quiescent* if every instance that starts in E completes in E .

Processes communicate with one another by issuing events that apply *read-modify-write* (RMW) primitives to vectors of base objects. We assume that a primitive is always applied to vectors of the same size. This size is called the *arity* of the primitive. RMW primitives of arity 1 are called *single-object RMW primitives*. RMW primitives of arity larger than 1 are called *multi-object RMW primitives*. For presentation simplicity we assume that all the base objects to which a primitive is applied are over the same domain. A RMW primitive, applied to a vector of k base objects over some domain D , is characterized by a pair of functions, $\langle g, h \rangle$, where g is the primitive's *update function* and h is the primitive's *response function*. The update function $g : D^k \times W \rightarrow D^k$, for some input-values domain W , determines how the primitive updates the values of the base objects to which it is applied. Let e be an event, issued by process p after execution E , that applies the primitive $\langle g, h \rangle$ to a vector of base objects $\langle o_1, \dots, o_k \rangle$. Then e atomically does the following: it updates the values of objects o_1, \dots, o_k to the values of the components of the vector $g(\langle v_1, \dots, v_k \rangle, w)$, respectively, where $\vec{v} = \langle v_1, \dots, v_k \rangle$ is the vector of values of the base objects after E , and $w \in W$ is an input parameter to the primitive. We call \vec{v} the *object-values vector* of e after E . The RMW primitive returns a response value, $h(\vec{v}, w)$, to process p . If W is empty, we say that the primitive *takes no input*.

A *k -compare-and-swap* (k -CAS), for some integer $k \geq 1$, is an example of a RMW primitive. It receives

an input vector, $\langle \overline{old}_1, \dots, \overline{old}_k, new_1, \dots, new_k \rangle$, from D^{2k} . Its update function,

$$g(\overline{v}, \langle \overline{old}_1, \dots, \overline{old}_k, new_1, \dots, new_k \rangle),$$

changes the values of base objects o_1, \dots, o_k to values new_1, \dots, new_k , respectively, if and only if $v_i = \overline{old}_i$ for all $i \in \{1, \dots, k\}$. If this condition is met, we say that the k -CAS event was *successful*, otherwise we say that the k -CAS event was *unsuccessful*. The response function of a *non-reading* k -CAS primitive returns *true* if the k -CAS event was successful or *false* otherwise. The response function of a *reading* k -CAS primitive returns \overline{v} .

Read is a single-object RMW primitive. It takes no input, its update function is $g(\langle v \rangle) = \langle v \rangle$ and its response function is $h(\langle v \rangle) = v$. Write is another example of a single-object RMW primitive. Its update function is $g(\langle v \rangle, w) = \langle w \rangle$, and its response function is $h(\langle v \rangle, w) = ack$. A RMW primitive is *nontrivial* if it may change the values of some of the base object to which it is applied (e.g., write); otherwise, it is *trivial* (e.g., read). *Fetch&add* is another example of a single-object RMW primitive. Its update function is $g(\langle v \rangle, w) = \langle v + w \rangle$, for v, w integers, and its response function simply returns the previous value of the base object to which it is applied.

Next, we define the concept of conditional synchronization primitives.

Definition 1 A RMW primitive $\langle g, h \rangle$ is conditional if, for every possible input w , $\left| \{ \overline{v} \mid g(\overline{v}, w) \neq \overline{v} \} \right| \leq 1$. Let e be an event that applies the primitive $\langle g, h \rangle$ with input w . The change point of e is the unique vector c_w such that $g(c_w, w) \neq c_w$; any other vector is a fixed point of e .

In other words, a RMW primitive is a conditional primitive if, for every input w , there is at most one vector c_w such that $g(c_w, w) \neq c_w$. k -CAS is a conditional primitive for any integer $k \geq 1$. The single change point of a k -CAS event with input $\langle \overline{old}_1, \dots, \overline{old}_k, new_1, \dots, new_k \rangle$ is the vector $\langle \overline{old}_1, \dots, \overline{old}_k \rangle$. Read is also a conditional primitive, since read events have no change points.

We now define the notion of invisible events. This is a generalization of the definition provided in [16] that can be applied to multi-object primitives. Informally, an invisible event is an event by some process that cannot be observed by other processes.

Definition 2 Let e be a RMW event applied by process p to a vector of objects $\langle o_1, \dots, o_k \rangle$ in an execution E , where $E = E_1 e E_2$. We say that e is invisible in E on o_i , for $i \in \{1, \dots, k\}$, if either the value of o_i is not changed by e or if $E_2 = E' e' E''$, e' is a write event to o_i , E' is p -free, and no event in E' is applied to o_i . We say that e is invisible in E if e is invisible in E on all objects o_i , for $i \in \{1, \dots, k\}$.

All read events are invisible. A write event is invisible if the value of the object to which it is applied equals the value it writes. A RMW event is invisible if its object-values vector is a fixed point of the event when it is issued. A RMW event (and specifically a write event) e that is applied by process p to an objects vector is invisible if, before p applies another event, a write event is applied to each object o_i that is changed by e before another RMW event is applied to o .

If a RMW event e is not invisible in an execution E on some object o , we say that e is visible in E on o . If e is not invisible in E , we say that e is a visible event in E .

3 Step Lower Bounds for Counters and Related Objects

In this section we prove a lower bound on the average step complexity of solo-terminating implementations of a counter that use only read, write and conditional primitives. We then prove the same result for stacks and queues by using a simple reduction to counters. For the lower bounds obtained in this paper, we only consider executions in which every process performs at most a single operation instance.

A *counter* is an object whose domain is \mathcal{N} . It supports a single operation, *fetch&increment*. A counter implementation A is correct if the following holds for every non-empty *quiescent* execution E of A : the responses of the *fetch&increment* instances that complete in E constitute a contiguous range of integers starting from 0.

In order to prove the lower bound we argue about the extent to which processes must be aware of the participation of other processes in any execution of a counter implementation. Intuitively, a process p is aware of the participation of another process q in an execution if there is information flow from q to p in that execution. The following definitions formalize this notion.

Definition 3 Let e_q be an event by process q in an execution E , that applies a non-trivial primitive to a vector v of base objects. We say that an event e_p in E by process p is aware of e_q if e_p accesses a base object o such that at least one of the following holds:

- There is a prefix E' of E such that e_q is visible on o in E' and e_p is a RMW event that applies a primitive other than write to o , and it follows e_q in E' , or
- there is an event e_r that is aware of e_q in E and e_p is aware of e_r in E .

If an event e_p of process p is aware of an event e_q of process q in E , we say that p is aware of e_q and that e_p is aware of q in E .

The key intuitions behind our step lower bound proof are that first, in any n -process execution of a counter implementation, ‘many’ processes need to be aware of the participation of ‘many’ other processes in the execution, and second, if processes only use read, write and conditional primitives, then a scheduling adversary can order events so that information about the participation of processes in the computation accumulates ‘slowly’.

The following definition quantifies the extent to which a process is aware of the participation of other processes in an execution.

Definition 4 Process p is aware of process q after an execution E if either $p = q$ or p is aware of an event of q in E . The set of processes that p is aware of after E is called the awareness set of p after E , and it is denoted $F(E, p)$.

The following lemma proves a relation between the value returned by a *fetch&increment* operation instance of a process in some execution and the size of that process’ awareness set after that execution.

Lemma 1 Let E be an execution of a counter implementation. If the *fetch&increment* instance by p returns i in E then $|F(E, p)| > i$.

Proof: Assume, by way of contradiction, that there is an execution E and a process p such that a *fetch&increment* instance by p returns i in E and $|F(E, p)| \leq i$.

We construct a new execution E' in the following manner. For any process $q \notin F(E, p)$, we first remove all the events of q from E ; then, for any process q' , we remove all the events by q' that are aware of q . Note that if an event $e_{q'}$ of q' is aware of q , then all following events by q' are also aware of q and are removed. Also, no events of p are removed since p is not aware of any process not in $F(E, p)$.

We argue that E' is an execution, and that it is indistinguishable to p from E ; in fact, this holds for the execution prefix of every process.

We consider events in the order they appear in E' . Let $e_{q'}$ be an event by process q' that appears in E' , namely, $E' = E'_1 e_{q'} E'_2$. Since $e_{q'}$ appears also in E , we can also write $E = E_1 e_{q'} E_2$. For the induction, assume that E'_1 is an execution and that it is indistinguishable to every process that appears in it from E_1 .

In particular, q' does not distinguish between E_1 and E'_1 , and hence it takes the same step after E_1 and E'_1 . To see why q' obtains the same response in $e_{q'}$ after E_1 and after E'_1 , note that it can return a different response only if, in E , $e_{q'}$ is aware of an event e that was removed from E_1 . This happens only if e is aware of some process $q \notin F(E, p)$, implying that, in E , $e_{q'}$ is also aware of q , contradicting the fact $e_{q'}$ was not removed. Hence, $E'_1 e_{q'}$ is an execution and q' does not distinguish between $E'_1 e_{q'}$ and $E_1 e_{q'}$.

This implies that p 's *fetch&increment* instance returns i also in E' ; on the other hand, at most i processes participate in E' . Let E'' be the extension of E' in which the processes that participate in E' complete their operation instances, one at a time. This execution exists by solo-termination, and results in a quiescent execution. However, at most i instances of *fetch&increment* complete in E'' , and one of them (by p) returns i . Thus, the responses of the *fetch&increment* instances do not constitute a contiguous range starting from 0, violating the specification of a counter. ■

The following corollary is an immediate consequence of Lemma 1.

Corollary 2 *Let E be a quiescent n -process execution of a solo-terminating counter implementation, then $\sum_{p \in P} F(E, p) \geq (n + 1) \cdot (n + 2)/2$.*

We need the following technical definition and lemma.

Definition 5 *Let $S = \{e_1, \dots, e_k\}$ be a set of events by different processes that are enabled after some execution E , each about to apply write or a conditional RMW primitive. We say that an ordering of the events of S is a weakly-visible schedule of S after E , denoted by $\sigma(E, S)$, if the following holds. Let $E_1 = E\sigma(E, S)$, then*

1. *at most a single event of S is visible on any one object in E_1 . If $e_j \in S$ is visible on a base object in E_1 , then e_j is issued by a process that is not aware of any event of S in E_1 ,*
2. *any process is aware of at most a single event of S in E_1 , and*
3. *all the read events of S are scheduled in $\sigma(E, S)$ before any event of $\sigma(E, S)$ changes a base object.*

We use weakly-visible schedules in the sequel for constructing executions that ‘slow down’ the rate in which processes become aware of other processes. The following lemma states that every set of outstanding write and conditional events has a weakly-visible schedule.

Lemma 3 Let $S = \{e_1, \dots, e_k\}$ be a set of events by different processes that are enabled after some execution E , each about to apply write or a conditional RMW primitive. Then there is a weakly-visible schedule of S after E .

Proof: We construct a schedule $\sigma = \sigma_1\sigma_2\sigma_3$ of S after E as follows. The events e_i that are invisible in Ee_i , if any, are scheduled first (in an arbitrary order); let σ_1 denote the resulting execution fragment. We call all of the remaining events of S the *potentially visible events*. We next schedule all the remaining potentially-visible write events; let σ_2 denote the resulting execution fragment. Fragment σ_3 is composed from all the remaining events in a manner we describe shortly.

Fragment σ_1 consists of all the read events in S , the write events applied to some base object o with argument $value(E, o)$, and the conditional RMW events whose object-values vector is a fixed point after E . By Definition 2, none of these events is visible in $E\sigma$; by Definition 3, no process is aware of these events in $E\sigma$ and the processes that issue them are not aware of any event of S in $E\sigma$. Also, by Definition 3, none of the processes that issue the events of σ_2 is aware of any event of S in $E\sigma$.

All the events from which we construct σ_3 , if any, are conditional events. We now describe the construction of σ_3 . We first schedule all the remaining events e_i that are invisible in $E\sigma_1\sigma_2$ (in an arbitrary order); let σ'_3 denote the resulting fragment. No process is aware of the events of σ'_3 in $E\sigma$. We are left with events that access base objects that have not yet been changed by events in $\sigma_1\sigma_2\sigma'_3$. These events are scheduled iteratively as follows:

In iteration $j \geq 1$, we choose the next event to be scheduled, e_{i_j} , arbitrarily from the set of remaining events; e_{i_j} is called the *pivot event of iteration j* . Let v_{i_j} be the vector of base objects accessed by e_{i_j} . We then schedule all of the remaining events that access some base object that is changed by e_{i_j} , if there are such events, in an arbitrary order; these are the *non-pivot events of iteration j* .

We iterate in this manner until all the events have been scheduled. Consider the events scheduled in some iteration j . As the object-values vector of e_{i_j} is a change point of e_{i_j} , e_{i_j} changes at least one base object. Thus, the object-values vector of any non-pivot event in iteration j is a fixed point of this event. This implies that all the non-pivot events are invisible in $E\sigma$, no process is aware of these events in $E\sigma$, and the processes that issue them are only aware of e_{i_j} in $E\sigma$. Additionally, all the pivot events are visible in $E\sigma$ and the processes that issue them are not aware of any event of S in $E\sigma$.

Consider all the potentially-visible events that access a specific base object o , if there are any. We now show that at most one of them is visible on o in $E\sigma$. If there are any potentially-visible write events that access o , then let e_l be the last of them. All the potentially-visible write events, except for e_l , are invisible in $E\sigma$, because each is followed by e_l that writes to o before a non-write event accesses o .

If there are potentially-visible write events on o , then none of the conditional potentially-visible events on o is visible in $E\sigma$. This is because the value of o when they access it is the value written by e_l , hence the object-values vector of each such conditional event is a fixed point of the event. Moreover, e_l is the only event in S that all the processes that issue these events are aware of in $E\sigma$.

Otherwise, if there are no potentially-visible write events that access o , then o can only be modified by a pivot event of a single iteration j . In this case only e_{i_j} can be visible on o in $E\sigma$, the process that issues e_{i_j} is aware of no event of S in $E\sigma$, and all the non-pivot events of iteration j , if there any, are only aware of e_{i_j} in $E\sigma$. This implies that σ is a weakly-visible-schedule of S after E . ■

Information about processes that participate in an execution is transferred through base objects. The following definition quantifies the number of other processes a process can become aware of when it reads a base object.

Definition 6 *Let E be an execution, o be a base object, and q be a process. We say that o has record of q after E if there is an event e , visible on o in E , such that the following hold:*

1. $E = E_1eE_2$,
2. e is an application of a non-trivial primitive to an objects-vector that contains o by some process r such that $q \in F(E_1e, r)$.

The familiarity set of o after E , denoted $F(E, o)$, contains all processes that o has record of after E .

Definition 6 provides only an upper bound (not tight, in general) on the number of processes that a process may become aware of when it accesses a base object. For example, consider an execution E consisting of four events applied to a base object o : a write by process p , followed by a read by process q , followed by a write by process r , followed by a read by process s . As the write by p is visible in E , p is in the familiarity set of o after E . The read by s , however, cannot convey to s any information about p .

We also note that requirement (2) of Definition 6 guarantees that a RMW event e that modifies an object o extends o 's familiarity set with the familiarity sets of all other objects accessed by e .

Definition 7 *Let E be an execution. We let $\mathcal{M}(E) = \max(\{|F(E, p)| \mid p \in \mathbf{P}\} \cup \{|F(E, o)| \mid o \in \mathbf{B}\})$ denote the maximum size of a process awareness set and object familiarity set after E .*

Definition 8 *Let \mathcal{P} be a set of synchronization primitives. We say that \mathcal{P} is c -bounded, for some constant c , if for every execution E and for every set S of events that are enabled after E , applying primitives from \mathcal{P} , there is a schedule σ of S such that $\mathcal{M}(E\sigma)/\mathcal{M}(E) \leq c$ holds.*

From Definition 8, it is clear that the smaller c is, the more can a scheduling adversary slow down the rate in which processes become aware of others.

Lemma 4 *The set of primitives that contains write and all the conditional primitives of arity k or less is $(2k + 1)$ -bounded.*

Proof: Let S be a set of events by different processes that are enabled after some execution E , each about to apply a write or a conditional RMW primitive of arity k or less. From Lemma 3, $\sigma(E, S)$ exists. We show that $\mathcal{M}(E\sigma(E, S))/\mathcal{M}(E) \leq 2k + 1$ holds.

Let $E_1 = E\sigma(E, S)$ and let o be some base object. By Definition 5, at most one event of S is visible on o in E_1 . If there is no such event, then $F(E_1, o) = F(E, o)$. Otherwise, there is a single such event, e , issued by some process p . Let o_1, \dots, o_j , for some $j < k$, be the base objects accessed by e in addition to o , if any. By Definition 5, p is not aware of any event from S in E_1 . Thus $F(E_1, o) \subset F(E, o) \cup F(E, p) \cup \bigcup_{l=1}^j F(E, o_l)$, hence $|F(E_1, o)| \leq (k + 1)\mathcal{M}(E)$ holds.

Let us now consider the maximum size of process awareness sets after E_1 . Clearly, $F(E_1, p) = F(E, p)$ for any process p that issues no event in S . By Definition 3, the same holds for all the processes that issue

write events in S . Let p be a process that issues a read event in S and let o be the base object accessed by that event. From Definition 5, the read events in S are scheduled before any event of S changes a base object. It follows that $F(E_1, p) \subset F(E, p) \cup F(E, o)$ holds, which implies, in turn, that $|F(E_1, p)| \leq 2\mathcal{M}(E)$ holds.

Consider a conditional RMW event $e \in S$ by process p , that accesses base objects o_1, \dots, o_j for some $j \leq k$. By Definition 5, if e is visible in E_1 , then p is aware of no event of S in E_1 . Hence, $F(E_1, p) \subset F(E, p) \cup_{l=1}^j F(E, o_l)$. Otherwise, e is invisible in E_1 and, from Definition 5, p is aware of at most a single event e' from S in E_1 . Let q be the process that issues e' then, again from Definition 5, q is not aware of any event of S in E_1 . Let o'_1, \dots, o'_{j_1} , for some $j_1 < k$, be the base objects accessed by e' in addition to o , if any. Thus, we have $F(E_1, p) \subset F(E, p) \cup F(E, q) \cup_{l=1}^j F(E, o_l) \cup_{l=1}^{j_1} F(E, o'_l)$. We get that $|F(E_1, p)| \leq (2k + 1)\mathcal{M}(E)$ holds, regardless of whether or not e is visible in E_i . This concludes the proof of the lemma. ■

Lemma 5 *Let A be an n -process solo-terminating implementation of a counter from base objects that support only primitives from a c -bounded set \mathcal{P} . Then A has an execution E that contains $\Omega(n \log_c n)$ events, in which every process performs a single fetch&increment instance.*

Proof: We construct an n -process execution, E , with $\Omega(n \log_c n)$ events, in which every process performs a single *fetch&increment* instance. The construction proceeds in rounds, indexed by the integers $1, 2, \dots, r$, for some $r \in \Omega(\log_c n)$, and it maintains the following invariant: before round i starts, the size of the awareness set of any process and the size of the familiarity set of any base object is at most c^{i-1} .

If a process p has not completed its *fetch&increment* instance before round i starts, we say that p is *active in round i* . All processes are active in round 1. All the processes that are active in round i have an enabled event in the beginning of round i . We denote the set of these events by S_i . We denote the execution that consists of all the events issued in rounds $1, \dots, i$ by E_i . We also let E_0 denote the empty execution.

For the induction base, note that, before execution starts, objects have no record of processes and processes are only aware of themselves. Thus $\mathcal{M}(E_0) = 1$ holds.

Assume that $\mathcal{M}(E_i) \leq c^{i-1}$ holds. Since \mathcal{P} is c -bounded, there is an ordering σ_i of the events of S_i such that $\mathcal{M}(E_{i-1}\sigma_i) \leq c\mathcal{M}(E_{i-1}) \leq c^i$. We let $E_i = E_{i-1}\sigma_i$.

By Corollary 2, the awareness set of at least $n/3$ processes must contain at least $n/4$ other processes after E . Therefore, each of these processes is active in at least the first $\log_c(n/4 - 1)$ rounds, performing at least $\log_c(n/4 - 1)$ events in E . ■

Our step complexity lower bound is immediate from Lemmas 4 and 5.

Theorem 6 *Let A be an n -process solo-terminating implementation of a counter from base objects that support only read, write and either reading or non-reading conditional primitives of arity k or less. Then A has an execution E that contains $\Omega(n \log_{k+1} n)$ events, in which every process performs a single fetch&increment instance.*

A similar result holds for stacks and queues.

Theorem 7 *Let A be an n -process solo-terminating implementation of a stack or queue from base objects that support only read, write and either reading or non-reading conditional primitives of arity k or less. Then A has an execution E , in which the average number of events issued while performing an operation instance is $\Omega(\log_{k+1} n)$.*

Proof: Assume, by way of contradiction, that there is a solo-terminating implementation of a one-time queue or a stack, A , so that the average number of events issued per operation instance in every execution of A is $o(\log_{k+1} n)$. We show how to implement a solo-terminating counter by using A .

The implementation uses a queue, or stack, implemented by A , that is initialized as follows. If A is a queue, then it is initialized by enqueueing into it the integers $0, \dots, n-1$, in an increasing order. If A is a stack, then it is initialized by pushing into it the integers $0 \dots n-1$, in a decreasing order. A *fetch&increment* operation performs a *dequeue*, if A implements a queue, or a *pop*, if A is a stack. Clearly, this is a solo-terminating implementation of a counter, each of whose executions contains $o(n \log_{k+1} n)$ steps, which is a contradiction to Theorem 6. ■

Jayanti [25, Theorem 6.2] proves a (base 2) logarithmic lower bound on the *worst-case* time complexity of obstruction-free implementations of a set of objects that includes counter, stack and queue from the following primitives: *load-linked*, *store-conditional*, *validate*, *move* and *swap*. When instantiated with $k = 1$, our Theorems 6 and 7 establish a (base 2) logarithmic lower bound on the *average* time complexity of such implementations and so are stronger. It can be shown that Theorems 6 and 7 hold also for implementations that can use the primitives considered in [25]. As the focus of our paper is the k -CAS primitives family, and since the load-linked/store-conditional primitives require a more complex abstraction of base objects than that required for the rest of the paper, we present these proofs in an appendix.

4 Step and Round Lower Bounds for Collect

In this section we present time lower bounds for the collect problem. We start by establishing lower bounds on a variant of collect, called the *input collection problem* (ICP), and prove the lower bounds on ordinary collect by reduction to ICP.

The input to ICP is an n -bit vector that is given in an array of n base objects, each of which stores one bit. An ICP object supports a single operation called *collect*, which every process performs at most once. The response of the *collect* operation is an n -bit number whose i 'th bit equals the i 'th input bit. We prove step- and round complexity lower bounds on implementations of ICP.

We define round complexity similarly to [10]. A *round* of an execution E is a consecutive sequence of events in E , in which every process that is active just before the sequence begins issues at least one event. A *minimal round* is a round such that no proper prefix of it is a round. Every execution can be uniquely partitioned into minimal rounds. The *round complexity* of E is the number of rounds in this partition. The *round complexity* of an implementation is the supremum over the round complexity of all its executions. Round complexity is a meaningful measure of time for fail-free executions in which processes operate at approximately the same speed.

We use a variation on a technique of Beame [12] to prove an $\Omega(\log_2 n)$ lower bound on the round complexity of solo-terminating implementations of ICP. This bound holds even when non-reading conditional

primitives of *any arity* may be used, in addition to multi-writer multi-reader registers.

Fix an implementation A of ICP. For notational simplicity, we assume in this section that all base objects are indexed, where o_j denotes the j 'th base object. The base objects of the input array are o_1, \dots, o_n .

The proofs presented in this section consider only the subset of synchronous executions of A , denoted $\mathcal{E}(A)$, in which the participating processes issue their events in lock-step. Clearly $\mathcal{E}(A)$ is a proper subset of all the possible executions of A . Proving our lower bound for this subset can only strengthen it.

In detail, an execution E in $\mathcal{E}(A)$ proceeds in *rounds*. In the beginning of each round, each of the participating processes whose instance of *collect* has not yet been completed has an enabled event; all processes have an enabled event in the beginning of round 1. In each round, these enabled events are scheduled in a specific order, according to a weakly-visible schedule. As we consider deterministic implementations, this implies that the states of all processes and the values of all base objects right after each round of E terminates depend solely on the input vector. The unique execution of \mathcal{E} with input vector I is denoted E_I . An execution $E_I \in \mathcal{E}(A)$ terminates after the *collect* instances of all the processes complete.

For input vector I , $E_{I,t}$ is the prefix of E_I containing all the events issued in rounds $1, \dots, t$ of E_I , and $S(E_{I,t})$ is the set of the events that are enabled just before round t of E_I starts. In round t , we extend $E_{I,t-1}$ with a weakly-visible schedule of $S(E_{I,t})$ after $E_{I,t-1}$ to obtain $E_{I,t}$. Lemma 3 guarantees that such a schedule exists.

The following definition formalizes the notion of *partitions*, the key concept that we borrow from Beame's technique. Similarly to [12], in the following we consider a *full-information model*, i.e., we assume that the state of any process reflects the entire history of the events it issued (and their corresponding responses) and that objects are large enough to store any such state.

We let $PV(i, t)$ (respectively $CV(j, t)$) denote the set of all possible states of process p_i (respectively the possible values of object o_j) right after round t of an execution $E \in \mathcal{E}(A)$ terminates. The sets $PV(i, t)$ and $CV(j, t)$ induce a partitioning of the input vectors to equivalence classes.

Definition 9 *The process partition $P(i, t)$ is the partition of the input vectors to equivalence classes that is induced by the set $PV(i, t)$. Two input vectors I_1, I_2 are in the same class of $P(i, t)$ if and only if there is a state $s \in PV(i, t)$ so that p_i is in state s after round t of both executions E_{I_1} and E_{I_2} . The object partition, $C(j, t)$, is defined similarly.*

By Definition 9, we have that for every process p_i , object o_j and round t :

$$\begin{aligned} |PV(i, t)|, |CV(j, t)| &\leq |\mathcal{E}(A)| = 2^n \\ |PV(i, t)| &= |P(i, t)| \\ |CV(j, t)| &= |C(j, t)| \end{aligned}$$

Theorem 8 *Let A be a solo-terminating implementation of ICP from base objects that support only read, write and non-reading conditional primitives of any arity. Then there is an execution of $\mathcal{E}(A)$ in which some process issues $\Omega(\log_2 n)$ events as it performs its instance of *collect*.*

Proof: Assume there is a process p_i whose instance of *collect* completes in round m or an earlier round in every execution of $\mathcal{E}(A)$; we show that $m \in \Omega(\log_2 n)$. The *collect* instance of p_i returns different responses

for different input vectors. As the response of the *collect* instance performed by p_i depends only on p_i 's state after the m 'th step, we have: $|P(i, m)| = 2^n$. Let $r_t = \max_i |P(i, t)|$ and $c_t = \max_j |C(j, t)|$, respectively, denote the maximum size of all process and object partitions right after round t . Let r_0 and c_0 respectively denote the maximum size of any process partition and object partition just before execution starts. We prove that r_t, c_t satisfy the recurrences:

$$(1) \quad r_{t+1} \leq r_t \cdot c_t, \text{ and}$$

$$(2) \quad c_{t+1} \leq n \cdot r_t + c_t$$

with initial conditions:

$$(3) \quad r_0 = 1, \text{ and}$$

$$(4) \quad c_0 \leq 2.$$

Before any execution starts, we have that for every j , $1 \leq j \leq n$, $|C(j, 1)| = 2$, since the single bit in every input base object partitions the set of input vectors into two. We also have for every $j > n$, $|C(j, 1)| = 1$, since other base objects have the same initial value, regardless of the input. Additionally, we have that for every i , $|P(i, 1)| = 1$, since the initial state of a process does not depend on the input vector. Thus initial conditions (3) and (4) hold.

Assume the claim holds for rounds $1, \dots, t$ and consider round $t + 1$. The primitive applied by p_i in round $t + 1$ and the base objects to which it is applied depend only on p_i 's state before round $t + 1$ begins. Therefore, the number of different events applied by p_i in round $t + 1$ of all the executions of $\mathcal{E}(A)$ is at most $|P(i, t)| \leq r_t$.

The size of p_i 's partition can grow in round $t + 1$ only when p_i applies a read or a non-reading conditional primitive in round $t + 1$. We now consider these two possibilities. First, if p_i applies a non-reading conditional primitive, then it receives a single bit as its response; in this case, every state of p_i before round $t + 1$ starts can change to one of at most two states. Second, if p_i applies a read to some object o_j , then, by Definition 5, the read is applied before o_j is changed in round $t + 1$. By induction hypothesis, the value read by the event belongs to a set of size at most $|CV(i, t)| \leq c_t$. It follows that p_i 's state can change to one of at most c_t different states. In both cases, we get that $|P(i, t + 1)| \leq r_t \cdot c_t$, which proves recurrence (1).

We now consider the set of values, $CV(j, t + 1)$, that some object o_j may assume right after round $t + 1$ in all the executions of $\mathcal{E}(A)$. There may be executions in which no process writes to o_j in round $t + 1$, thus we may have:

$$CV(j, t) \subseteq CV(j, t + 1). \tag{1}$$

Let $n(j, t + 1)$ denote the number of distinct values that o_j may assume right after round $t + 1$ in all of the executions in which its value is modified during that round. Let E_I be such an execution. By Definition 5, at most a single event of $S(E_{I, t+1})$ is visible on o_j after $E_{I, t+1}$; if there is such an event, then it is issued by a process that is not aware of any event of $S(E_{I, t+1})$. Thus, the number of distinct values written to o_j by any process p_i in round $t + 1$ of all executions is at most $|P(i, t)| \leq r_t$. As any process may write to o_j in round $t + 1$ we get:

$$n(j, t + 1) \leq \sum_{k=1}^n |P(k, t)| \leq n \cdot r_t. \tag{2}$$

Combining Equations 1 and 2 proves recurrence (2). As shown in [12], solving the recurrences for the sequences r_i, c_j yields $m \geq \log_2 n + 1 - \log(1 + \log_2 2n)$. Thus, there is an execution in which some process performs $\Omega(\log_2 n)$ events. ■

The following result follows immediately from Theorem 8.

Theorem 9 *The round complexity of any solo-terminating implementation of ICP from base objects that support only read, write and non-reading conditional primitives of any arity is $\Omega(\log_2 n)$.*

Theorem 8 also implies the following lower bound on the *average* step complexity of ICP.

Theorem 10 *Let A be a solo-terminating implementation of ICP from base objects that support only read, write and non-reading conditional primitives of any arity. Then A has an execution that contains $\Omega(n \log_2 n)$ events.*

Proof: Assume, by way of contradiction, that there is a solo-terminating implementation A of ICP from such base objects so that every execution of A contains $o(n \log_2 n)$ events. We use A to construct another implementation of ICP, A' , in the following manner. We name the algorithms used by A and A' for implementing the *collect* operation $A.collect$ and $A'.collect$, respectively.

A' uses a base object, named *result*, in addition to all the base objects used by A , that is initialized with the special value *null*. $A'.collect$ alternates between steps in which it emulates $A.collect$ and steps in which it reads *result*; the former are called *emulation steps*. Step number $2 \cdot i - 1$ of $A'.collect$, for $i \geq 1$, is identical to step i performed by $A.collect$ on the same input vector, unless an emulated step of $A.collect$ returns a response and terminates.

If a step by $A.collect$ returns response v and terminates, then A' writes v to the *response* object and only then returns response v and terminates. After performing any emulation step that does not terminate, $A'.collect$ performs an additional step during which it reads the *result* object. If the read value is non-*null*, then $A'.collect$ returns that value as its response and terminates. Otherwise, $A'.collect$ proceeds to emulate the next step of $A.collect$. It is easily seen that A' is a solo-terminating implementation of ICP and that every execution of $\mathcal{E}(A')$ contains $o(n \log_2 n)$ events. Also, for any pair of processes p, q and any execution E of $\mathcal{E}(A')$, the difference between the number of events issued by p and q in E is at most 1. It follows that the number of events performed by any process in any execution of $\mathcal{E}(A')$ is $o(\log_2 n)$, contradicting Theorem 8. ■

Deriving the Worst-Case Lower Bound for Ordinary Collect: The lower bound on the worst-case step complexity of ordinary collect is proved by reduction to the ICP problem.

A *Collect object* supports two operations. A *store*(v) operation by process p_i makes v be the latest value stored by p_i . The *collect* operation returns a set of process-value pairs; it should not miss a preceding *store* operation, nor include a *store* operation that has not yet begun. We let V_i denote the value returned for process p_i , and require the following properties from every *collect* operation and every process p_i :

- If $V_i = \perp$, then no *store* operation by p_i completes before the *collect* operation Φ .

Algorithm 1 Implementing ICP collect from ordinary collect, pseudo-code for process p_i .

shared *one-time-Collect* A **initially** {empty,..., empty}, *register* *result* **initially** empty

ICP-collect()

```
1: int b ← input bit  $i$ 
2:  $A$ .store $_i(b)$ 
3:  $V \leftarrow A$ .collect()
4: if  $\forall i \in \{1, \dots, n\} : V_i \neq \text{empty}$ 
5:    $\text{result} \leftarrow V_1 \cdot V_2 \cdots V_{n-1} \cdot V_n$ 
6:   return result
7: else
8:   do  $V \leftarrow \text{result}$  until  $\text{result} \neq \text{empty}$  od
9:   return result
```

- $V_i = v \neq \perp$ implies that v is the parameter of a *store* operation Ψ by p_i that does not start after Φ , and there is no *store* operation by p_i that follows Ψ and precedes Φ .

This definition captures a weak version of a Collect object, often called *gather* [4]; clearly, our lower bound holds for stronger variants of collect [4, 5, 9].

Our bound holds even for *one-time Collect*, in which every process can only apply each operation once.

Theorem 11 *Let A be a solo-terminating implementation of one-time Collect from base objects that support only read, write and non-reading conditional primitives of any arity. Then the worst-case step complexity of A is in $\Omega(\log_2 n)$.*

Proof: Assume, by way of contradiction, that there is a solo-terminating implementation A of one-time Collect from such base objects, so that every instance of *store* and *collect* applied by it issues $o(\log_2 n)$ events.

Algorithm 1 presents A' , an implementation of ICP that uses a one-time Collect object, implemented by A . The entries of the object are from the domain {empty, 0, 1} and are initialized to empty. A' also uses a register *result*. To apply its ICP-collect operation, process p_i reads the i 'th input bit (line 1) and applies an *update* operation to the one-time Collect object to store the value of its input bit to the i 'th entry of A (line 2). Then, p_i performs a *collect* on A and stores the result to the local variable V (line 3). If the collect returns all the input bits, p_i stores the concatenation of these bits to *result* and returns this value as its response (lines 5, 6); otherwise, p_i repeatedly reads the *result* register until it becomes non-empty and then returns this value as its response (lines 8, 9).

Consider the behavior of A' in a synchronous execution $E \in \mathcal{E}(A')$. Let r be the first round of E after which the *store* operations of all processes terminate and let p_j be a process whose *store* operation terminates in round r . The *ICP-collect* instance of p_j returns a concatenation of the input bits. The code and the properties of A imply that the *ICP-collect* instances of all processes return the same value. Thus A' is a correct implementation of ICP.

By assumption, p_j returns a response after $o(\log n)$ rounds. Hence, all processes return their responses after $o(\log n)$ rounds, contradicting Theorem 10. ■

5 Space complexity

In this section, we show that any wait-free implementation of a large class of objects from base objects that support conditional primitives, read and write must use $\Omega(n)$ such objects. The result holds for any *visible* object, which is, intuitively, an object that supports some operation Op that must issue a visible event in any instance. This class contains widely-used objects such as counter, stack, queue, and single-writer snapshot.

Our result holds for conditional primitives of arbitrary *arity*, and extends a lower bound of Fich et al. [16], which allows only *unary* conditional primitives. The results of this section apply to both reading and non-reading conditional primitives.

Let A be a wait-free implementation of a visible object. Fich et al. [16] prove that A can be brought to a state where all processes have pending indexed events whose visibility depends on their index: an event with index i cannot be made invisible by events with indices larger than i . Such a state is called an *n-levelled state*. This concept is formalized by the following definition.

Definition 10 [16] *The state resulting from a finite execution E is n-levelled if there is a sequence e_1, e_2, \dots, e_n of events by different processes, all about to apply non-trivial primitives, such that, for every nonempty execution fragment E' consisting of some subset of these events (in any order), e_j is visible in EE' , where $j = \min\{i | e_i \in E'\}$. We call e_1, e_2, \dots, e_n an n-levelled sequence and say that event e_j is at level j .*

An object that only supports read and write primitives is called a *register*. An object that can only be accessed by conditional primitives (of any arity) is called a *multi-conditional* object. An object that may be accessed by read, write and conditional primitives (of any arity) is called a *read-write-multi-conditional* object.

Let e be a write or a conditional event. The *change set* of e , denoted $\mathcal{C}(e)$, is the set of base objects whose values may be changed by e ; its size is called the *change multiplicity* of e and denoted $c(e)$. If e is a write event, then \mathcal{C} contains the single object accessed by e . If e is a conditional event, then $\mathcal{C}(e)$ is the set of objects whose values are changed by e if e is issued when its object-values vector is a change-point of e .

Assume the implementation A uses base objects that support only read, write and conditional primitives (of any arity); let $SPACE(A)$ denote the number of base objects used by A . We prove that if A can be brought to an n -levelled state, then $SPACE(A) = \Omega(n)$. In fact, multi-object conditionals may *worsen* the implementation's space complexity: the lower bound on space complexity that we obtain is proportional to the sum of the change multiplicities of the issued events.

Lemma 12 *Assume that after execution E , A is in an n-levelled state. Let $S = \{e_1, \dots, e_n\}$ be a corresponding n-levelled sequence, where S_w and S_c are, respectively, the subset of write events of S and the subset of conditional events of S .*

1. *If A uses only registers and multi-conditional objects, then $SPACE(A) \geq \sum_{i=1}^n c(e_i)$.*

2. If A uses only read-write-multi-conditional objects, then
 $SPACE(A) \geq \max\left(\left(\sum_{i=1}^n c(e_i)\right) - |S_w|, \lceil n/2 \rceil\right)$.

Proof: Let e_i, e_j be two events of S , $i < j$. Assume first that both e_i and e_j are conditional events. We now show that $\mathcal{C}(e_i) \cap \mathcal{C}(e_j) = \phi$. Assume, by way of contradiction, that there is some object $o \in \mathcal{C}(e_i) \cap \mathcal{C}(e_j)$. By Definition 10, e_i is visible in Ee_i and e_j is visible in Ee_j . Thus, the object-values vector of e_i (respectively e_j) after E is a change-point of e_i (respectively e_j). Since $i < j$, again by Definition 10, e_i is visible in Ce_je_i . However, since o is in the change set of e_j , its value is changed by e_j . Therefore, the object-values vector of e_i after Ee_j is a fixed-point of e_i . This contradicts the assumption that e_i is visible in Ee_j . It is easily seen that $\mathcal{C}(e_i) \cap \mathcal{C}(e_j) = \phi$ also when e_i, e_j are both write events. This proves the first part of the lemma.

Assume that A uses only read-write-multi-conditional objects. Since at most one write event and one conditional event may change any one object, we have $SPACE(A) \geq \lceil n/2 \rceil$. If $\mathcal{C}(e_i) \cap \mathcal{C}(e_j) \neq \phi$, then e_i must be a write event and e_j must be a conditional event. Thus, $|\mathcal{C}(e_i) \cap \mathcal{C}(e_j)| = 1$, proving the second part of the lemma. ■

Since any wait-free implementation of a visible object has an n -levelled state [16, Lemma 3.2], Lemma 12 implies the following theorem:

Theorem 13 *Let A be an n -process wait-free implementation of a visible object.*

- *If A uses only registers or multi-conditional objects, then $SPACE(A) \geq n$.*
- *If A uses only multi-conditional objects and $\mathcal{C}(e) \geq k$ for any conditional event e issued in an execution of A , then $SPACE(A) \geq k \cdot n$.*
- *If A uses only read-write-multi-conditionals objects, then $SPACE(P) \geq \lceil n/2 \rceil$.*

6 Summary

The lower bounds presented in this paper indicate that using k -CAS does not reduce neither the time- nor the space-complexity of implementations of many widely-used concurrent objects. Thus, supporting k -CAS primitives yields little performance gains, unless they are *reading*, namely, return the values of the base objects they access. Implementing such primitives in hardware is challenging, however, due to *fan-in* limitations.

We emphasize that our results do not bear on the merits of k -CAS primitives in terms of simplifying the design of non-locking algorithms. The question of whether or not the possible reduction in design complexity justifies the cost of implementing k -CAS in hardware is still debatable.

In Section 4, we proved a logarithmic lower bound on the average step complexity of the input collection problem (ICP). We then proved a logarithmic lower bound on the *worst-case* step complexity of ordinary collect by way of reduction from ICP. Whether or not a logarithmic lower bound on the average step-complexity of ordinary collect holds is an interesting open question.

Acknowledgements: The authors thank Maged Michael who triggered this research by asking whether the results of [16] hold with k -CAS primitives. We would also like to thank Faith Ellen Fich for referring us to Beame’s paper, and Nir Shavit for helpful discussions on the topics of this paper. Danny Hendler was supported by Sun Microsystems.

References

- [1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 1995 ACM Symposium on Theory of Computing*, pages 538–547, 1995.
- [2] Y. Afek, M. Merritt, and G. Taubenfeld. The power of multi-objects. *Information and Computation*, 153(1):117–138, 1999.
- [3] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 111–120, 1997.
- [4] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 262–272, Phoenix, 1999. IEEE Computer Society Press.
- [5] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 71–80, New-York, 2000. ACM Press.
- [6] O. Agesen, D. Detlefs, C. H. Flood, A. T. Garthwaite, P. A. Martin, M. Moir, N. Shavit, and G. L. S. Jr. Dcas-based concurrent dequeues. *Theory of Computing Systems*, 35(3):349–386, 2002.
- [7] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 184–193, New York, NY, USA, 1995. ACM Press.
- [8] H. Attiya and E. Dagan. Improved implementations of binary universal operations. *Journal of the ACM*, 48(5):1013–1037, 2001.
- [9] H. Attiya and A. Fouren. Algorithms adaptive to point contention. *Journal of the ACM*, 50(4):444–468, July 2003.
- [10] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.
- [11] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
- [12] P. Beame. Limits on the power of concurrent-write parallel machines. *Information and Computation*, 76(1):13–28, 1988.

- [13] R. Cole and O. Zajicek. The apram: incorporating asynchrony into the pram model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.
- [14] S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224, 2004.
- [15] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, Sept. 1998.
- [16] F. E. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 80–87, 2004.
- [17] M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. *SIAM Journal on Computing*, 25(6):1332–1357, Dec. 1996.
- [18] K. Fraser. *Practical Lock-Freedom*. PhD thesis, Kings College University of Cambridge, Sept. 2003.
- [19] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, Aug. 1999.
- [20] M. B. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136, 1996.
- [21] P. H. Ha and P. Tsigas. Reactive multi-word synchronization. In *12th International Conference on Parallel Architectures and Compilation Techniques*, pages 184–193, 2003.
- [22] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, Jan. 1991.
- [23] M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 522–529, 2003.
- [24] Intel Corporation. Intel itanium processor-specific application binary interface, 2001.
- [25] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *PODC*, pages 201–210, 1998.
- [26] P. Jayanti and S. Khanna. On the power of multi-objects. In *WDAG*, pages 320–332, 1997.
- [27] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 314–323, 2003.
- [28] Motorola. *MC68020 32-Bit Microprocessor User’s Manual*. Prentice-Hall, 2nd edition, 1986.

- [29] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.
- [30] SPARC International, Inc., Mountain View, CA. *The SPARC Architecture Manual Version 9, 1/e*. Prentice Hall, 1994.

A Implementations Using Additional Primitives

In this appendix, we extend the results presented in Section 3 to show that the logarithmic lower bounds of Theorems 6 and 7 hold also when implementations can use some non-conditional primitives, in addition to write. We now describe the extended set of primitives we allow and the shared-memory model extensions required for defining them. These extensions are required for allowing a formalization of the *load-linked* (LL), *store-conditional* (SC), and *validate* primitives, since the effect of these operations depends on the process that applies them. The formalism we use here is very similar to that used by Jayanti [25].

A base object o is composed of two components: $val(o)$, storing the object data, and $Pset(o)$, which is an array of process identifiers. In the following description, we let u denote the value of $val(o)$ just before the primitive is applied to o . We also let p_i be the process that applies the primitive.

Base objects support one or more of the following primitives:

- $read(o)$ returns u and leaves o unchanged.
- $write(o, v)$ sets $o.val \leftarrow v$ and returns ack . If $v \neq u$ holds, the event also sets $Pset(o) \leftarrow \phi$.
- $LL(o)$ sets $Pset(o) \leftarrow Pset(o) \cup \{p_i\}$ and returns u .
- If $p_i \in Pset(o)$ holds, then $SC(o, v)$ sets $val(o) \leftarrow v$ and returns $(true, u)$. In this case we say the event is *successful*. If $v \neq u$ holds, the event also sets $Pset(o) \leftarrow \phi$. If $p_i \notin Pset(o)$ holds, then $SC(o, v)$ returns $(false, u)$ and does not change o . We say the event is *unsuccessful*.
- If $p_i \in Pset(o)$ holds, then $validate(o)$ returns $true$, otherwise it returns $false$.
- $swap(o, v)$ sets $val(o) \leftarrow v$ and returns u . If $v \neq u$ holds, the event also sets $Pset(o) \leftarrow \phi$.
- $move(o, o_1)$ sets $val(o_1) \leftarrow u$ and returns ack . If $val(o_1) \neq u$ holds, the event also sets $Pset(o_1) \leftarrow \phi$.
- *conditional primitives of arity k or less*: Conditional primitives, as specified in Definition 1, retain their semantics and operate on o 'th val component with the following addition: whenever a conditional event changes $val(o)$, it also sets $Pset(o) \leftarrow \phi$.

In other words, the *swap* primitive atomically writes a value to a base object and returns its previous value. The *move* primitive copies the value of one base object to another. Both primitives initialize the $Pset$ component of the object they write to if they modify its value. The write and conditional primitives retain their semantics with the following addition: whenever they change a base object's value, they initialize its $Pset$ component. The SC primitive, applied by p_i , succeeds in changing base object o 's value only if o 's

value was not changed since the last time p_i applied LL to it, otherwise it fails. If the SC operation does change o 's value, o 's $Pset$ component is initialized. If the SC operation fails, o is not changed. Finally, $validate$ by p_i returns true iff an SC by p_i can still succeed.

In the following proofs we use the same definition of visibility (Definition 2) used in the paper, with the understanding that it applies to the val components of base objects. We let \mathcal{P} denote the set containing the primitives described above. Let e be an event, issued by p_i , applying a primitive from \mathcal{P} to base object o . We emphasize that e may be observed by processes other than p_i only if it changes $val(o)$. This is because e resets $Pset(o)$ only if it changes $val(o)$. (LL by p_i adds p_i to $Pset(o)$, but this cannot be observed by other processes.) Thus Lemma 1 and Corollary 2 hold also in this extended model.

Our proof uses the following definition and lemma, defined and proven in [25], for ordering a set of outstanding *move* events so that ‘little’ information is transferred. Phrasing is slightly adapted according to the terminology we use.

Definition 11 [25] *Let S be a set of move events, enabled after some execution E . An ordering σ of the events of S is called *secretive*, if the value originating at any object is moved in σ by at most 2 processes.*

Lemma 14 [25] *For all executions E and for any set S of move events enabled after E , a secretive ordering of the events of S after E exists.*

Lemma 15 *The set of primitives \mathcal{P} is $24(2k + 1)$ -bounded.*

Proof: Let S be a set of events by different processes that are enabled after some execution E , each about to apply a primitive from \mathcal{P} . We show that there is an ordering σ of S such that $\mathcal{M}(E\sigma)/\mathcal{M}(E) \leq 24(2k + 1)$.

Let $S' \subset S$ be the subset of events in S that apply write or conditional primitives. From Lemma 4, there is an ordering α of S' such that $\mathcal{M}(E\alpha)/\mathcal{M}(E) \leq 2k + 1$.

We next schedule all the *validate* events in S , if any, in an arbitrary order. Let β be the resulting execution fragment, let e be an event in β issued by p , and let o be the object accessed by e . Since *validate* does not change $val(o)$, e is not visible on o . It follows that $F(E\alpha\beta, o) = F(E\alpha, o)$. Since e accesses only o , $F(E\alpha\beta, p) = F(E\alpha, p) \cup F(E\alpha, o)$. This implies in turn that $\mathcal{M}(E\alpha\beta)/\mathcal{M}(E) \leq 2(2k + 1)$.

We next schedule all the LL events in S , in some arbitrary order, followed by all the SC events in S , in some arbitrary order. Let γ be the resulting execution fragment. From the semantics of LL and SC , at most one of the events of γ is visible on any single base object o . Thus $|F(E\alpha\beta\gamma, o)| \leq 4(2k + 1)\mathcal{M}(E)$. Let e be an event in γ , issued by p , that accesses o . Since e accesses only o , $F(E\alpha\beta\gamma, p) = F(E\alpha\beta, p) \cup F(E\alpha\beta, o)$. It follows that $\mathcal{M}(E\alpha\beta\gamma) \leq 4(2k + 1)\mathcal{M}(E)$.

Next we schedule all the *swap* events in S , in some arbitrary order. Let δ be the resulting execution fragment and let o be a base object. From the semantics of *swap*, o 's value is either not changed by δ or its value after δ equals the input of the last *swap* event e that accesses o in δ . In either case, $|F(E\alpha\beta\gamma\delta, o)| \leq 2\mathcal{M}(E\alpha\beta\gamma)$. Let p be a process that issues an event of δ to o . The response p receives is either o 's value right after $E\alpha\beta\gamma$ (if p applies the first event in δ that accesses o), or the input of the preceding event in γ that accesses o . In either case, $|F(E\alpha\beta\gamma\delta, o)| \leq 2\mathcal{M}(E\alpha\beta\gamma)$. Thus $\mathcal{M}(E\alpha\beta\gamma\delta) \leq 8(2k + 1)\mathcal{M}(E)$ follows.

Finally, we schedule all the *move* events in S . From Lemma 14, there is a secretive ordering, ϵ , of these events. We let $\sigma = \alpha\beta\gamma\delta\epsilon$. From the semantics of *move*, process awareness sets are not changed by ϵ . Since

ϵ is secretive, the value of each base object o after $E\sigma$ was moved into o in ϵ by the events of at most two processes. Thus $|F(E\sigma, o)| \leq 3\mathcal{M}(E\alpha\beta\gamma\delta) \leq 24(k+1)\mathcal{M}(E)$. ■

Our lower bounds follow.

Theorem 16 *Let A be an n -process solo-terminating implementation of a counter from base objects that support only read, write, move, swap, LL, SC, validate, and either reading or non-reading conditional primitives of arity k or less. Then A has an execution E that contains $\Omega(n \log_{k+1} n)$ events, in which every process performs a single fetch&increment instance.*

Proof: Immediate from Lemmas 5 and 15. ■

Theorem 17 *Let A be an n -process solo-terminating implementation of a stack or queue from base objects that support only read, write, move, swap, LL, SC, validate, and either reading or non-reading conditional primitives of arity k or less. Then A has an execution E , in which the average number of events issued while performing an operation instance is $\Omega(\log_k n)$.*

Proof: Immediate from the proof of Theorem 7 and from Theorem 16. ■