Time and Space Lower Bounds for Implementations Using k-CAS
(Extended Abstract)

Hagit Attiya[1] and Danny Hendler[2]

[1] Department of Computer Science, Technion
[2] Department of Computer Science, University of Toronto

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 any 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. This bound is tight.

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. *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 [20]. It has consequently became a synchronization primitive of choice, and hardware support for it is provided in many multiprocessor architectures [22,27,29].

In recent years, the question of supporting multi-object conditionals in hardware has been deliberated in both industrial and academic circles [12,16,17,19]. The design of concurrent data structures seems to be easier if conditional primitives can be applied to multiple objects [17]. On the other hand, almost all of the current architectures support conditional primitives only on a single object.

To help resolve 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* 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 [23]. 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. Solo termination [13,25] requires a process running alone to complete its operation within a finite number of its steps. (This property is provided by *obstruction-free* implementations [21].)

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 *non-reading k-CAS* primitives that only return a boolean success indication.

For the weaker wake-up problem [15], 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 [23] 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 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 [10], originally applied in the synchronous CRCW PRAM model. This implies that the *round complexity* [11] 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 [8].

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 [14], 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 object.

Several shared object implementations use k-CAS, most often DCAS, to simplify design (e.g. [5,18]). Doherty et al. [12] 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. [3,6,7,9,28]). A few papers investigate the consensus number of multi-object operations [2,24]. Attiya and Dagan [7] 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 [26] is a weaker variant of non-reading k-CAS, and 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 $\mathbf{P}=\{p_1,\cdots,p_n\}$ 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. We denote the empty execution fragment by ϵ. 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'.

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 last event of an operation instance Φ has been issued in an execution E, 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.

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 with arity 1 are called *single-object RMW primitives*. RMW primitives with 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 $\langle o_1, \ldots, o_k \rangle$. Then e atomically does the following: it updates the values of objects $o_1, \ldots, o_k$ to the values of the components of the vector $g(\langle v_1, \ldots, v_k \rangle, w)$, respectively, where $\overrightarrow{v} = \langle v_1, \ldots, v_k \rangle$ is the vector of values of the base objects after E, and $w \in W$ is an input parameter to the primitive. We call $\overrightarrow{v}$ the *object-values vector* of e after E. The RMW primitive returns a response value, $h(\overrightarrow{v}, w)$, 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, $\langle old_1, \ldots, old_k, new_1, \ldots, new_k \rangle$, from D^{2k}. Its update function, $g(\overrightarrow{v}, \langle old_1, \ldots, old_k, new_1, \ldots, new_k \rangle)$, changes the values of base objects $o_1, \ldots, o_k$ to values $new_1, \ldots, new_k$, respectively, if and only if $v_i = old_i$ for all $i \in \{1, \ldots, k\}$. 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 $\overrightarrow{v}$.

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., read; it is *trivial*, otherwise. *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, $\left| \left\{ \overrightarrow{v} \mid g(\overrightarrow{v}, w) \neq \overrightarrow{v} \right\} \right| \leq 1$. Let e be an event that applies the primitive $\langle g, h \rangle$ with input w. A vector c_w such that $g(c_w, w) \neq c_w$ is called the change point of e. Any vector $v \neq c_w$ is called 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 $\langle old_1, \ldots, old_k, new_1, \ldots, new_k \rangle$ is the vector $\langle old_1, \ldots, old_k \rangle$. Read is also a conditional primitive.

Next we define the notion of invisible events. This is a generalization of the definition provided in [14] 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 $\langle o_1, \ldots, o_k \rangle$ in an execution E, where $E = E_1 e E_2$. We say that e is* invisible in *E on o_i, for $i \in \{1, \ldots, k\}$, 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 $i \in \{1, \ldots, k\}$.*

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. This can only strengthen our lower bounds.

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 any 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 information flow from q to p is possible in that execution. The following definitions formalize this notion.

Definition 3. *Let E be an execution and p, q be two distinct processes. Let e_q be an event in E, by process q, that applies a non-trivial primitive to a vector v of base objects. We say that p is* aware of e_q in E through event f *if v contains 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 there is a RMW event f that applies a primitive other than write to o, issued by p, that follows e_q in E',*

- *there is a process $r \notin \{p, q\}$ that is aware of e_q in E through an event g and p is aware of g in E through f.*

If p is aware of an event e in E through one or more (other) events, we say that p is aware of e in E. If p is aware of an event e of q in E, then p is also aware of all of q's previous events in E.

If p is aware of any event by q in E, then p may be aware of q's participation in the execution. The key intuition behind our step lower bound proof is that 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. The following definition provides a quantification of the extent to which a process is aware of the participation of other processes in an execution.

Definition 4. *Let E be an execution and let p and q be processes. We say that p is aware of q after E if either $p = q$ or if p is aware of some event of q in E. We denote by $F(E, p)$ the set of processes that p is aware of after E. We call this set the awareness set of p after E. If p is aware of q after E and $p \neq q$, we denote the last event of q in E that p is aware of in E by* lastAware(E,p,q).

Information about processes that participate in the execution flows through base objects. The following definition provides a quantification to the number of other processes a process can become aware of when it reads from a base object.

Definition 5. *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 exists an event e in E such that all of the following hold. (1) $E = E_1 e E_2$, (2) e is an application of a non-trivial primitive to an objects-vector that contains o by some process r such that $q \in F(E_1 e, r)$, and (3) e is visible in E on o. We define the familiarity set of o after E as the set of all processes that o has record of after E, and denote it by $F(E, o)$.*

Definition 5 only provides an upper bound (not necessarily tight) on the number of other processes that a process may become aware of when it accesses a base object. This can only strengthen our lower bound. We also note that requirement (2) of Definition 5 makes sure 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.

The following lemma proves an intuitively-clear 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 $|F(E, p)| > i$.*

The following corollary is an immediate consequence of Lemma 1.

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

We need the following technical definition and lemma.

Definition 6. *Let $S = \{e_1, \cdots, e_k\}$ be a set of events by different processes that are enabled after some execution E, all about to apply write and/or conditional RMW primitives. 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.*

Lemma 2. *Let $S = \{e_1, \cdots, e_k\}$ be a set of events by different processes that are enabled after some execution E, all about to apply write and/or conditional RMW primitives. Then there is a weakly-visible-schedule of S after E.*

Lemma 2 is proved by a careful ordering of the events of S that is done in an iterative manner. Our step complexity lower bounds follow.

Theorem 1. *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 with 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. We construct an n-process execution, E, of length $\Omega(n \log_{k+1} n)$ in which every process performs a single *fetch&increment* instance. The construction proceeds in rounds, indexed by the integers $1, 2, \cdots, r$ for some $r \in \Omega(\log_{k+1} n)$. We prove that the construction maintains the following invariant: before round i starts, the awareness set of any process and the familiarity set of any base object has size at most $(2k + 1)^{i-1}$.

Before execution starts, objects have no record of processes and processes are only aware of themselves, thus the induction claim holds. 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 $1, \ldots, i$ by E_i.

From Lemma 2, there is a weakly-visible-schedule, $\sigma(E_{i-1}, S_i)$, of the events of S_i after E_{i-1}. E_i is constructed by extending E_{i-1} with $\sigma(E_{i-1}, S_i)$.

Assume the induction hypothesis holds before round i starts. Let o be some base object. From Definition 6, at most one event of S_i is visible on o in E_i. If there is no such event, then $F(E_i, o) = F(E_{i-1}, o)$. Otherwise there is a single such event, e, issued by some process p. Let $o_1, \ldots, o_j$, for some $j, 1 \leq j \leq k-1$ be the base objects accessed by e in addition to o, if any. From Definition 6, p is not

aware of any event from S_i in E_i. Thus $F(E_i, o) \subset F(E_{i-1}, o) \cup F(E_{i-1}, p) \cup_{l=1}^{j} F(E_{i-1}, o_l)$ hence, from the induction hypothesis, $|F(E_i, o)| \leq (k+1)(2k+1)^{i-1}$. Therefore the induction hypothesis for round $i + 1$ holds for all base objects.

Let us now consider the maximal size of the awareness set of any process right after round i terminates. Clearly, $F(E_i, p) = F(E_{i-1}, p)$ for any process p that is not active in round i. From Definition 3, the same holds for all the processes that issue a write event in round i. Let p be a process that issues a read event in round i that accesses some base object o. As reads are scheduled before any event changes a base object in round i, we have $F(E_i, p) \subset F(E_{i-1}, p) \cup F(E_{i-1}, o)$ hence $|F(E_i, p)| \leq 2(2k+1)^{i-1}$.

Consider a conditional RMW event e by process p that is issued in round i and accesses base objects $o_1, \ldots, o_j$ for some $j \leq k$. From Definition 6, if e is visible in E_i, then p is aware of no event of S_i in E_i. Hence $F(E_i, p) \subset F(E_{i-1}, p) \cup_{l=1}^{j} F(E_{i-1}, o_l)$. Otherwise e is invisible in E_i and, again from Definition 6, p is aware of at most a single event e' from S_i in E_i. Let q be the process that issues e', then q is not aware of any event of S_i in E_i. Let $o'_1, \ldots, o'_{j_1}$, for some $1 \leq j_1 \leq k - 1$, be the base objects accessed by e' in addition to o, if any.

Thus we have $F(E_i, p) \subset F(E_{i-1}, p) \cup F(E_{i-1}, q) \cup_{l=1}^{j} F(E_{i-1}, o_l) \cup_{l=1}^{j_1} F(E_{i-1}, o'_l)$. Consequently, we have $|F(E_i, p)| \leq (2k+1)^i$ regardless of whether e is visible in E_i or not. Thus the induction hypothesis holds for all processes before round $i + 1$ starts.

From Corollary 1, there are at least $n/3$ processes the awareness set of each of which contains at least $n/4$ other processes after E. Consequently each of these processes is active in at least the first $\log_{2k+1}(n/4 - 1)$ rounds, hence each of these processes performs at least $\log_{2k+1}(n/4 - 1)$ events in E. ∎

The full version contains a similar result for stacks and queues.

Theorem 2. *Let A be an n-process solo-terminating implementation of a stack or a queue from base objects that support only read, write and either reading or non-reading conditional primitives with 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.*

By using techniques from [23], Theorems 1 and 2 can be extended to hold also if base objects support the *validate, swap* and *move* primitives.

4   Step and Round Lower Bounds for Collect

In this section we consider a variation on collect that we call the *input collection problem* (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. It can easily be seen that these bounds hold also for the ordinary collect

object (defined in, e.g., [4]) by considering executions of collect in which every process performs a *store* instance immediately followed by a *collect* instance.

Round complexity is defined as follows. Let E be an execution. A *round* of 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. Let A be an implementation. A's round complexity 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.

Attiya et al. [8] present an $O(\log_2 n)$ round complexity implementation of ICP from read and write. They prove a matching lower bound for such implementations. In this section we prove an $\Omega(\log_2 n)$ round complexity lower bound for ICP implementations that can use non-reading k-CAS primitives for any k, in addition to read and write. Thus we show that non-reading k-CAS is no stronger than read and write in terms of ICP implementation round complexity.

Beame [10] proves a lower bound of $\Omega(\log_2 n)$ for a problem similar to ICP in the concurrent-read concurrent-write (CRCW) PRAM model. We use a variation on his technique to prove a similar lower bound for solo-terminating implementations of ICP even when non-reading conditional primitives *of any arity* may be used. Clearly a fan-in argument would not work in this case.

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 base object indexed by j. The base objects of the input array are $o_1, \ldots, o_n$.

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, which we will shorty describe. As we consider deterministic implementations, this implies the following: 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. Thus, we denote the single execution of $\mathcal{E}$ that results when the input vector is I by E_I. An execution $E_I \in \mathcal{E}(A)$ terminates after the *collect* instances of all the processes complete.

Let E_I be the execution of $\mathcal{E}(A)$ for some input vector I. We denote by $E_{I,t}$ the prefix of E_I that contains all the events issued in rounds $1, \ldots, t$ of E_I. We denote by $S(E_{I,t})$ the set of the events that are enabled just before round t of E_I starts. Then 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 2 guarantees that this can be done.

The following definition formalizes the notion of partitions, which is the key concept of the technique of Beame [10] that we apply.

Definition 7. *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. 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}. We define an object partition, $C(j,t)$, similarly.*

From Definition 7, we have $|PV(i,t)|, |CV(j,t)| \le |\mathcal{E}(A)| = 2^n$, $|PV(i,t)| = |P(i,t)|$ and $|CV(j,t)| = |C(j,t)|$, for any process p_i, object o_j and round t.

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. This assumption can obviously only strengthen our lower bound.

Theorem 3. *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 before the response is returned, 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 the sequences r_t, c_t satisfy the recurrences: (1) $r_{t+1} \le r_t \cdot c_t$, and (2) $c_{t+1} \le n \cdot r_t + c_t$ with initial conditions: (3) $r_0 = 1$, and (4) $c_0 \le 2$.

Before any execution starts, we have $\forall j \in \{1, \ldots, n\} : |C(j,1)| = 2$, as the single bit in every input base object partitions the set of input vectors to 2. We also have $\forall j > n : |C(j,1)| = 1$, as other base objects have the same initial value regardless of the input. Additionally we have $\forall i : |P(i,1)| = 1$, as 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 $1, \cdots, t$ and consider round $t + 1$. Let us consider $|P(i, t+1)|$, the partition size of process p_i right after round $t + 1$ terminates. p_i's partition size can grow in round $t+1$ only because of executions in which p_i applies a read or a non-reading conditional primitive in round $t+1$. The primitive applied by p_i in round $t + 1$ and the base objects to which it is applied are only a function of p_i's state before round $t + 1$ begins. Thus the

number of different events applied by p_i in round $t + 1$ of all the executions of
$\mathcal{E}(A)$ is at most $P(i, t) \leq r_t$.

We consider the following two possibilities. If p_i applies a non-reading con-
ditional primitive in round $t + 1$ of an execution, then it receives a single bit
response. In this case every state of p_i before round $t + 1$ starts can change
to one of at most two states. If p_i applies a read to some object o_j in round
$t + 1$ of an execution, then, from Definition 6, the read is applied before o_j is
changed in round $t + 1$. Thus, from induction hypothesis, the event can read
at most $|CV(i, t)| \leq c_t$ different values. Hence p_i's state in each such execu-
tion can change to one of at most c_t different states. In either case we get:
$|P(i, t + 1)| \leq r_t \cdot c_t$, which proves recurrence (1).

Let o_j be some base object. We now consider the set of values, $CV(j, t+1)$,
that object o_j may assume right after round $t+1$ terminates in all the executions
of $\mathcal{E}(A)$. There may be executions in which no process writes to o_j during 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. From Definition 6, at most a single
event of $S(E_{I,t+1})$ may be 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 [10], 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 lower bound on the *average* step complexity of ICP also follows
from the proof of Theorem 3.

Theorem 4. *Let A be a solo-terminating implementation of ICP from base ob-
jects 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.*

5   Space Complexity

Fich et al. [14] consider wait-free implementations of a class of *visible* objects.
Intuitively, a visible object supports some operation Op such that any instance
of Op must issue a visible event before it completes. This class contains widely-
used objects such as counter, stack, queue, and single-writer snapshot. They

show that any wait-free implementation of a visible object from base objects that support only *unary* conditional primitives, read and write must use $\Omega(n)$ such objects. In this section we generalize this result and show that it holds also for implementations that may use conditional primitives of any arity. 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. Lemma 3.2 in [14] proves 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 is being formalized by the following definition.

Definition 8. *The state resulting from a finite execution E is n-levelled if there is a sequence $e_1, e_2, \ldots, e_n$ 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 $e_1, e_2, \ldots, e_n$ 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 only supports read, write and may be accessed by 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.

In what follows we consider an implementation, A, that uses base objects that only support read, write and conditional primitives of any arity. We 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 3. *Assume that after execution E, A is in an n-levelled state. Let $S = \{e_1, \ldots, e_n\}$ be a corresponding n-levelled sequence. Let S_w and S_c respectively denote 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 $SPACE(A) \geq max\left((\sum_{i=1}^{n} c(e_i)) - |S_w|, \lceil n/2 \rceil\right)$.*

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 otherwise

to obtain a contradiction, then there is some object $o \in \mathcal{C}(e_i) \cap \mathcal{C}(e_j)$. From Definition 8, 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). As $i < j$, again from Definition 8, e_i is visible in Ce_je_i. However, as o is in the change set of e_j, its value is changed by e_j. Consequently the object-values vector of e_i after Ee_j is a fixed-point of e_i. This is a contradiction to 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 *(1)*.

Assume that A uses only read-write-multi-conditional objects. As 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 it must be that e_i is a write event and e_j a conditional event, thus $|\mathcal{C}(e_i) \cap \mathcal{C}(e_j)| = 1$. This proves *(2)*.   ∎

The above lemma and Lemma 3.2 of [14] immediately imply the following:

Theorem 5. *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 $SPACE(P) \geq \lceil n/2 \rceil$.*

References

1. Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In STOC, pages 538547, 1995.
2. Y. Afek, M. Merritt, and G. Taubenfeld. The power of multi-objects. Information and Computation, 153(1):117138, 1999.
3. Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In PODC, pages 111120, 1997.
4. Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In FOCS, page 262, 1999.
5. 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 deques. Theory Comput. Syst., 35(3):349386, 2002.
6. 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 184193, New York, NY, USA, 1995. ACM Press.
7. H. Attiya and E. Dagan. Improved implementations of binary universal operations. Journal of the ACM, 48(5):10131037, 2001.

8. H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? Journal of the ACM, 41(4):725763, July 1994.
9. G. Barnes. A method for implementing lock-free shared data structures. In SPAA, pages 261270, 1993.
10. P. Beame. Limits on the power of concurrent-write parallel machines. Information and Computation, 76(1):1328, 1988.
11. R. Cole and O. Zajicek. The apram: incorporating asynchrony into the pram model. In SPAA, pages 169178, 1989.
12. 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 SPAA, pages 216224, 2004.
13. F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. Journal of the ACM, 45(5):843862, Sept. 1998.
14. F. E. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In PODC, pages 8087, 2004.
15. M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. SIAM Journal on Computing, 25(6):13321357, Dec. 1996.
16. K. Fraser. Practical Lock-Freedom. PhD thesis, Kings College University of Cambridge, Sept. 2003.
17. M. Greenwald. Non-Blocking Synchronization and System Design. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, Aug. 1999.
18. M. B. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In OSDI, pages 123136, 1996.
19. P. H. Ha and P. Tsigas. Reactive multi-word synchronization. In 12th International Conference on Parallel Architectures and Compilation Techniques, pages 184193, 2003.
20. M. Herlihy. Wait-free synchronization. ACM Transactions On Programming Languages and Systems, 13(1):123149, Jan. 1991.
21. M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Doubleended queues as an example. In ICDCS, pages 522529, 2003.
22. Intel Corporation. Intel itanium processor-specific application binary interface, 2001.
23. P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In PODC, pages 201210, 1998.
24. P. Jayanti and S. Khanna. On the power of multi-objects. In WDAG, pages 320332, 1997.
25. P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for non-blocking implementations. Siam J. Comput., 30(2):438456, 2000.
26. V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In SPAA, pages 314323, 2003.
27. Motorola. MC68020 32-Bit Microprocessor Users Manual. Prentice-Hall, 2nd edition, 1986.
28. N. Shavit and D. Touitou. Software transactional memory. Distributed Computing, 10(2):99116, February 1997.
29. SPARC International, Inc., Mountain View, CA. The SPARC Architecture Manual Version 9, 1/e. Prentice Hall, 1994.