Lower Bounds for Adaptive Collect and Related Objects

(Extended Abstract)

Hagit Attiya
Department of Computer Science
Technion, Israel
hagit@cs.technion.ac.il

Faith Ellen Fich
Department of Computer Science
University of Toronto
fich@cs.toronto.edu

Yaniv Kaplan
Department of Computer Science
Technion, Israel
yanivk@cs.technion.ac.il

ABSTRACT

An *adaptive* algorithm, whose step complexity adjusts to the number of active processes, is attractive for situations in which the number of participating processes is highly variable. This paper studies the number and type of multi-writer registers that are needed for adaptive algorithms.

We prove that if a collect algorithm is f-adaptive to total contention, namely, its step complexity is $f(k)$, where k is the number of processes that ever took a step, then it uses $\Omega(f^{-1}(n))$ multi-writer registers, where n is the total number of processes in the system.

Furthermore, we show that competition for the underlying registers is inherent for adaptive collect algorithms. We consider *c-write* registers, to which at most c processes can be concurrently about to write. Special attention is given to *exclusive-write* registers, the case $c = 1$ where no competition is allowed, and *concurrent-write* registers, the case $c = n$ where any amount of competition is allowed. A collect algorithm is f-adaptive to point contention, if its step complexity is $f(k)$, where k is the maximum number of simultaneously active processes. Such an algorithm is shown to require $\Omega(f^{-1}(\frac{n}{c}))$ concurrent-write registers, even if an unlimited number of c-write registers are available. A smaller lower bound is also obtained in this situation for collect algorithms that are f-adaptive to total contention.

The lower bounds also hold for nondeterministic implementations of *sensitive* objects from historyless objects.

Finally, we present lower bounds on the step complexity in solo executions (i.e., without any contention), when only c-write registers are used: For weak test&set objects, we present an $\Omega(\frac{\log n}{\log c + \log\log n})$ lower bound. Our lower bound for collect and sensitive objects is $\Omega(\frac{n-1}{c})$.

Categories and Subject Descriptors

D.4.1 [**OPERATING SYSTEMS**]: Process Management—*Synchronization*; F.1.2 [**COMPUTATION BY ABSTRACT DEVICES**]: Modes of Computation—*Parallelism and concurrency*

General Terms

Algorithms, Theory

Keywords

Contention, adaptivity, collect, weak test&set, sensitive objects, solo termination, exclusive-write registers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.
Copyright 2004 ACM 1-58113-802-4/04/0007 ...$5.00.

1. INTRODUCTION

To solve certain problems, processes need to collect up-to-date information about the other participating processes. For example, in the renaming problem, the participating processes need to choose distinct names from a small name space. To ensure that it chooses a different name, a process needs information about which names other processes have chosen.

One way information about other processes can be communicated is to use an array of registers indexed by process identifiers. An active process can update information about itself by writing into its register. A process can *collect* the information it wants about other participating processes by reading the entire array of registers.

When there are only a few participating processes, it would be nice to be able to collect the required information more quickly. An *adaptive* algorithm is one whose step complexity is a function of the number of participating processes. Specifically, if it performs at most $f(k)$ steps when there are k participating processes, we say that it is f-adaptive. There are three common ways to count the number of participating processes. An algorithm is *adaptive to point contention* if the number of steps taken by a process while performing this algorithm is a function of the maximum number of processes that were simultaneously active at some point in time during that period of time. It is *adaptive to interval contention* if this number of steps is a function of the total number of different processes that were active during that period of time. Finally, it is *adaptive to total contention* if this number of steps is a function of the total number of different processes that were active before or during that period of time (i.e., since the beginning of the execution). Note that an algorithm that is f-adaptive to point contention is also f-adaptive to interval contention and an algorithm that is f-adaptive to interval contention is also f-adaptive to total contention.

A number of different adaptive collect algorithms have been presented [2, 8, 9, 11]. In particular, there is an algorithm that features an asymptotically optimal $O(k)$-adaptive collect, but its memory consumption is exponential in n, the total number of processes in the system [9]. Applying ideas from this algorithm with the matrix structure of Moir and Anderson's adaptive renaming algorithm [25] leads to a collect algorithm with polynomial (in n) memory complexity, but $\Theta(k^2)$ step complexity.

A simple adversary argument shows that at least one multi-writer register is needed by an adaptive collect algorithm, even if every process can write to a single-writer register.

Afek, Boxer, and Touitou [1] improve this lower bound by proving that any constant number of multi-writer registers is not sufficient. They actually prove that any long-lived weak test&set adaptive to interval contention requires a non-constant number of multi-writer registers. The lower bound for collect is obtained by a simple reduction, since long-lived weak test&set can easily be implemented using collect. Their proof is complicated, using Ramsey-theoretic arguments, and achieves only a small non-constant lower bound, because concurrency grows rapidly. Their lower bound does not apply to adaptive *one-shot* weak test&set, which can be implemented using only two multi-writer registers.

Aguilera, Englert and Gafni [4] specially construct a *generalized weak-test&set* object, and show that a one-shot implementation of this object adaptive to total contention requires a small non-constant number of multi-writer registers. Their proof also relies on Ramsey-theoretic arguments. A lower bound for adaptive one-shot collect is obtained by reduction, since one-shot generalized test&set is also easily implemented using collect.

Here, we significantly improve these space lower bounds. We prove (in Section 3.1) that any f-adaptive one-shot collect algorithm requires $\Omega(f^{-1}(n))$ multi-writer registers. By focusing directly on collect, our proof is significantly simpler and, by careful management of concurrency in the executions we construct, our proof yields a higher bound.

A *multi-writer* register can be written to by all processes, whereas a *single-writer* register can be written to by only one specific process. We refine this classification by considering the amount of competition multi-writer registers allow.

An *exclusive-write* register is not *owned* by a single process and, in principle, all processes may write to it. However, at any point, no more than one process can be about to write to it. In contrast, any number of processes may be simultaneously about to write to the same *concurrent-write* register. This distinction is analogous to the well-known distinction between exclusive-write and concurrent-write registers used in synchronous parallel computing [17, 24].

Exclusive-write registers avoid *data races*, allowing significant simplifications to the memory architecture, for example, the caching protocols. A few distributed algorithms use exclusive-write registers to improve the step complexity of atomic snapshots [7, 10]. In these algorithms, exclusive access is guaranteed by making sure that, in every execution, each register is written to by only one process; however, the identity of this process can change in different executions. These restricted exclusive-write registers are called *dynamic single-writer* registers.

Taking a broader perspective, multi-writer registers can be parameterized by the amount of competition they allow. All processes may write to a *c-write* register, but no more than c processes may concurrently be about to write to it [14, 19]. Exclusive-write registers are the special case where $c = 1$. Concurrent-write registers are the special case where $c = n$.

In Section 3.2, we extend our lower bound to the number of concurrent-write registers needed when an unbounded number of c-write registers are available. Specifically, we prove that a long-lived collect algorithm that is f-adaptive to point contention requires $\Omega(f^{-1}(\frac{n}{c}))$ concurrent-write registers. For long-lived collect algorithms that are f-adaptive to total contention, we have a somewhat smaller bound. Both of these lower bounds can be modified to hold for one-shot collect, at the cost of reduced bounds.

These lower bounds use *covering* arguments, first presented by Burns and Lynch [13] to prove that $\Omega(n)$ registers are necessary for n-process mutual exclusion. This technique has been used in many other papers to prove space lower bounds. (For more examples, see the survey by Fich and Ruppert [18].)

Our proofs depend on the fact that an adaptive algorithm can only read a small number of single-writer registers. Fatourou, Fich and Ruppert [15] use a similar property to prove a tight step lower bound for space optimal implementations of multi-writer snapshot objects.

In Section 4, we consider the number of steps taken in solo executions, when only c-write registers are available. Specifically, we prove an $\Omega(\frac{n-1}{c})$ lower bound for one-shot collect and sensitive objects. Using an information flow graph, we also show an almost optimal bound of $\Omega(\frac{\log n}{\log c + \log\log n})$ for one-shot weak test&set objects.

The *latency* of an algorithm is the maximal number of shared variables accessed by a single process in executing the algorithm. Hendler and Shavit [19] show that any implementation of an object in the *Influence(n)* class, using only read/write registers and *read-modify-write* objects, has latency at least $\frac{n-1}{c}$. Our bound for collect in Section 4 is actually on the latency. It is stronger than the result of [19] in that it applies to solo executions, but it does not allow the use of read-modify-write objects.

Dwork, Herlihy and Waarts [14] show that any n-process implementation of consensus, using only read/write registers and read-modify-write objects must allow $\Omega(n)$ processes to be simultaneously about to change a single register or read-modify-write object. For randomized consensus, they give a tradeoff between latency and the maximum amount of competition c allowed: the latency must be at least $(n-1)/c$. They also show that an n-process mutual exclusion algorithm has latency at least $\frac{\log n}{c}$.

Anderson and Kim [6] show that $\Omega(\log n / \log\log n)$ remote memory references are needed for n-process mutual exclusion. Their proof inductively constructs a run in which concurrent processes have no knowledge of each other; this is done by choosing independent processes in the information flow graph. The same construction was used to show an $\Omega(k)$ lower bound on the step complexity of an adaptive mutual exclusion algorithm, where k is the point contention [5].

Our bound for weak test&set does not follow from the results on mutual exclusion in [6, 14] since those results do not apply to solo executions. Moreover, the proof in [14] relies heavily on the fact that some process is guaranteed to win the mutual exclusion object, a property that is not guaranteed for weak test&set.

Jayanti, Tan and Toueg [23] prove that any implementation of a *perturbable* object requires $\Omega(n)$ *historyless* objects (defined in [16]) and has $\Omega(n)$ step complexity. Only a weak liveness condition, *nondeterministic solo termination* [16], is required for their proof. Their lower bound implies that, if there is an f-adaptive implementation of such an object, then $f(k) \in \Omega(k)$. Their proof does not rely on adaptivity, nor does it place any restrictions on how processes access objects. They also show that increment, fetch&add, modulo-b counter (for $b \geq 2n$), LL/SC bit, and b-valued compare&swap (for $b \geq n$) are perturbable.

Our lower bounds for f-adaptive collect can be similarly generalized to hold for f-adaptive implementations of *sensitive* objects from historyless objects under the nondeterministic solo termination condition. Although the set of sensitive objects is a proper subset of perturbable objects, it includes all the specific perturbable objects mentioned in [23]. These results appear in the Appendix.

2. MODEL

We assume a standard asynchronous shared-memory model of computation [20]. A system consists of n *processes*, $p_1, \ldots, p_n$, communicating by accessing shared *objects*, $Y_1, \ldots, Y_\ell$.

A shared object has a *type* that defines a domain of possible *values* (including a special initial value, $\perp$) and a set of *operations*, providing the only means to manipulate the object. The current value of an object and the operation applied to it determine the response to the operation and the resulting value of the object. The most common object is a *register*, providing two operations: *read*, returning the value of the register without changing it, and *write*, changing the register value to the value of its input.

Processes are deterministic state machines, each with a (possibly infinite) set of local states, which includes a unique *initial state*. In each *step*, the process determines the memory-access operation (e.g. read or write) to perform according to its local state, and changes its local state according to the value returned by the operation.

A *configuration* consists of the states of the processes and the values of the objects, namely, it is a vector

$$\langle s_1, \ldots, s_n, v_1, \ldots, v_\ell \rangle$$

where s_i is the local state of process p_i and v_j is the value of the shared object Y_j. In the *initial configuration*, every process is in its initial state and every object has $\perp$ as its initial value.

A *schedule* is a (possibly infinite) sequence $p_{i_1}, p_{i_2}, \ldots$ of process identifiers. For a finite schedule α and a (possibly infinite) schedule β, $\alpha\beta$ denotes their concatenation. An *execution* consists of the initial configuration and a schedule, representing the interleaving of steps by processes. An execution α *reaches* a configuration C if C is the configuration at the end of α.

For a set of processes P, a P*-only schedule* contains only processes in P, and a P*-free schedule* does not contain processes in P. These definitions extend naturally to P*-only executions* and P*-free executions*. When $P = \{p\}$, we write p-only and p-free instead of $\{p\}$-only and $\{p\}$-free.

For a configuration C and a process p_i, Cp_i denotes the configuration that is a result of letting p_i take a single step from configuration C. If $\alpha = p_{i_1}, p_{i_2}, \ldots, p_{i_l}$ is a finite schedule, $C\alpha$ denotes the configuration that results from letting $p_{i_1}, p_{i_2}, \ldots, p_{i_l}$ take steps from configuration C, in order of their appearance in α; that is, $C\alpha = Cp_{i_1}p_{i_2}\ldots p_{i_l}$.

A process p_i *covers* register r in configuration C if p is about to write to r (according to its state in C).

When a set of processes P covers a set of registers R in configuration C, if it is possible to perform a *block write*: a sequence of $|R|$ consecutive write operations, one to each register in R, each by a different process in P. A block write to R fixes the values of all the registers in R. Because the order of steps in a block write does not change the resulting configuration, the configuration denoted by CP is well defined when $|P| = |R|$.

It is possible to restrict access to an object to limit which processes may apply which operations to it. For registers, we consider the following types of restrictions:

A register Y is *single-writer* if it is owned by one process, so that, in every execution, only its owner can write to Y. It is *c-write* if, at any configuration, at most c processes cover (i.e. are about to write to) Y. Note that those processes may be different in different executions, or at different configurations in the same execution. When $c = 1$, we say that the register is *exclusive-write*. Finally, a *concurrent-write* register does not restrict the way processes write to it.

We assume all registers are *multi-reader*, so that all processes may read from all registers. Throughout the paper, we assume that each process p_i owns one single-writer register, denoted r_i.

An *implementation* of an object of type X using r objects $Y_1, \ldots, Y_\ell$, provides, for every operation OP of X, a set of n procedures $F_1, \ldots, F_n$, one for each process. (Typically, the procedures are the same for all processes.) To execute OP on X, process p_i calls procedure F_i, which specifies the steps of p_i and, thus, the operations on $Y_1, \ldots, Y_\ell$. The worst-case number of steps performed by some process p_i executing procedure F_i is the *step complexity* of implementing OP.

An operation OP_i *precedes* operation OP_j (and OP_j *follows* operation OP_i) in an execution α, if the call to the procedure of OP_j appears in α after the return from the procedure of OP_i.

Our proofs rely on a weak liveness property, called *solo termination*, which guarantees for any process p_i and configuration C, p_i returns within a finite number of steps in a p_i-only execution from C.

Let α be a finite execution. Process p_i is *active* at the end of α if α includes a call of an implementation procedure without a matching return. The set of active processes at the end of α is denoted $active(\alpha)$.

The point contention at the end of α is $|active(\alpha)|$.

Suppose $\alpha = \beta\gamma$. The point contention during the interval γ is the maximum of $|active(\beta\gamma')|$ taken over all prefixes γ' of γ. It is the maximum number of processes that are simultaneously active at some point during γ and is denoted by $pointCont(\gamma)$. The *interval contention* during γ, $intCont(\gamma)$, is the number of different processes such that each is active at the end of $\beta\gamma'$ for some (possibly different) prefix γ' of γ. Finally, the *total contention* during γ, $totalCont(\gamma)$, is $intCont(\alpha)$, namely, the cardinality of the set of all processes that are active before and during γ.

Let $f : \mathbf{N} \to \mathbf{N}$ be an increasing function. An implementation is f*-adaptive* to total, interval, or point contention if the step complexity of each of its procedures is bounded from above by $f(k)$, where k is the total, interval, or point contention, respectively, during the interval the procedure is

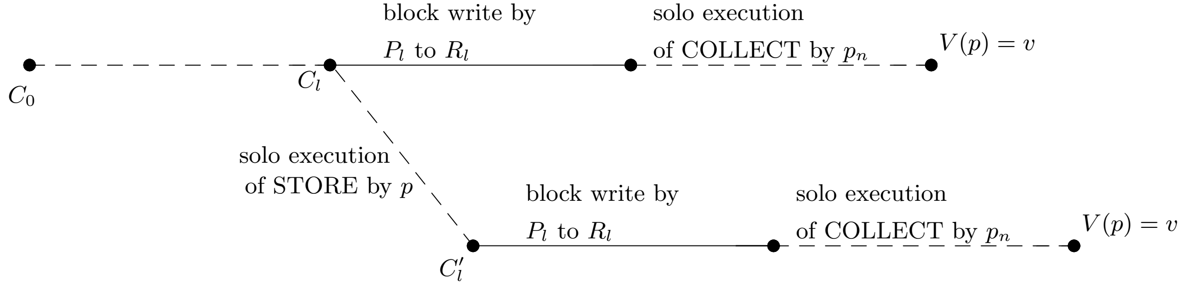

Figure 1: Illustration for Lemma 3.1

being performed (i.e., between the call and matching return of the procedure).

A *collect algorithm* provides two operations: A STORE(*val*) by process p_i sets *val* to be the latest value for p_i. A COLLECT operation returns a *view*, a partial function V from the set of processes to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each process p_i. A COLLECT operation *cop* should not read from the future or miss a preceding STORE operation. Formally, the following validity properties holds for every process p_i:

– If $V(p_i) = \bot$, then no STORE operation by p_i precedes *cop*.

– If $V(p_i) = v \neq \bot$, then v is the value of a STORE operation *sop* of p_i that does not follow *cop*, and there is no STORE operation by p_i that follows *sop* and precedes *cop*.

3. SPACE LOWER BOUNDS FOR COLLECT

We bound the number of multi-writer registers needed for adaptive n-process collect using a covering argument. A typical covering argument constructs an execution in which an increasingly large set of processes cover an increasingly large set of registers. To increase the size of these sets, we let some process p_i execute on its own, in a way that must be observed by later operations. This is difficult when proving lower bounds for weak test&set [1], since operations may abort after they observe an operation being performed by another process. This requires the effects of p_i to be cleaned up, complicating the proof and increasing the contention. Deriving a contradiction for collect is simpler: process p_i just stores a new value, which must be observed by a later COLLECT operation, by the second validity property. The STORE operation by p_i must write to an uncovered register; otherwise, a block write can be used to remove all traces of the STORE. This register must be multi-writer because an f-adaptive COLLECT operation cannot check all single-writer registers. This allows us to increase the size of the set of covered multi-writer registers.

In all the executions we construct, processes $p_1, \ldots, p_{n-1}$ invoke only STORE operations, each with a different value. Process p_n only invokes a COLLECT operation cop_n in a p_n-only schedule, denoted $cop_n(C)$ when starting from configuration C; $|cop_n(C)|$ denotes the number of steps p_n takes. $S(C)$ is the set of processes whose single-writer registers are read by p_n during $cop_n(C)$.

Suppose a process p_l is not active in configurations C_1 and C_2, and let op_l be an operation applied by p_l. We say that C_1 and C_2 are *indistinguishable to* op_l, if the sequence of state transitions by p_l are the same in the application of op_l from C_1 and from C_2. We denote this by $C_1 \overset{op_l}{\approx} C_2$.

Lemma 3.1 is the key to the lower bounds in this section. It shows that a process performing a STORE must write to a multi-writer register, unless p_n reads its single-writer register during its COLLECT.

LEMMA 3.1. *Let α_l be a p_n-free execution that reaches a configuration C_l in which a set P_l of l processes covers a set R_l of l multi-writer registers, and $P_l \subseteq active(\alpha_l)$. Let $C = C_l P_l$. Then for every process $p \notin S(C) \cup active(\alpha_l) \cup \{p_n\}$, there is a p-only schedule β such that, in configuration $C_l\beta$, p covers a multi-writer register $Y \notin R_l$ that is read in $cop_n(C)$ and $C_l P_l \overset{cop_n}{\approx} C_l \beta P_l$.*

PROOF. Consider a p-only execution of a STORE operation from C_l, and let γ be the corresponding schedule (see Fig. 1). Let $C'_l = C_l\gamma$. If p does not write to a multi-writer register $Y \notin R_l$ that is read during $cop_n(C)$, then $C_l P_l \overset{cop_n}{\approx} C'_l P_l$, because the processes in P_l also cover the registers in R_l in configuration C'_l. Therefore p_n returns the same value for $V(p)$ in both $cop_n(C_l P_l)$ and $cop_n(C'_l P_l)$. This contradicts the validity property of collect.

Let β be the prefix of γ up to, but not including p's first write to a multi-writer register $Y \notin R_l$ that is read during $cop_n(C)$. Then in configuration $C_l\beta$, process p covers Y and $C_l P_l \overset{cop_n}{\approx} C_l \beta P_l$. □

3.1 Lower Bound on the Number of Multi-Writer Registers

We start with a simple proof that does not distinguish between types of multi-writer registers. In Section 3.2, we extend the proof to bound the number of concurrent-write registers, even when an unlimited number of c-write registers are available.

LEMMA 3.2. *Let P_l be a set of l processes. Let α_l be a P_l-only execution that reaches a configuration C_l in which P_l covers a set R_l of l multi-writer registers and let $C = C_l P_l$. If $n > |cop_n(C)| + l + 1$, then there exists a process $p_{l+1} \notin P_l \cup \{p_n\}$ and a finite p_{l+1}-only schedule β such that in configuration $C_l\beta$, p_{l+1} covers a multi-writer register not in R_l.*

PROOF. Note that $|S(C)| \leq |cop_n(C)|$, because the number of registers accessed in an operation is bounded by the total number of steps. Since $n > |cop_n(C)| + l + 1$, there is a process $p_{l+1} \notin S(C) \cup P_l \cup \{p_n\}$. By Lemma 3.1, there is a

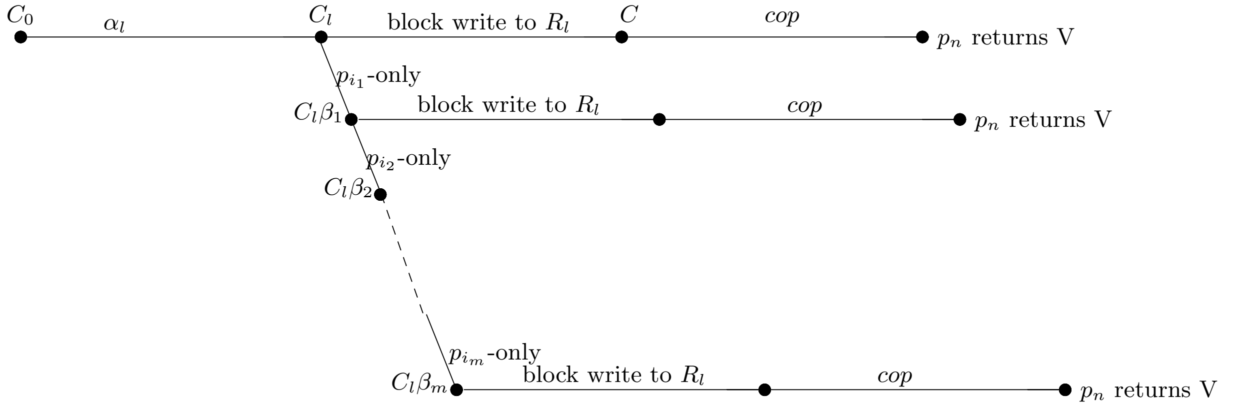

Figure 2: Illustration for Lemma 3.5.

p_{l+1}-only schedule, β, such that, in configuration $C_l\beta$, p_{l+1} covers a multi-writer register $Y \notin R_l$. $\square$

Applying Lemma 3.2 inductively yields the next lemma. Note that the total contention increases by one in each application of the lemma, allowing the collect operation by p_n to take more steps.

Lemma 3.3. *Consider any n-process collect algorithm that is f-adaptive to total contention. For any l such that $n > f(l) + l$, there is a set P_l of l processes and a P_l-only execution α_l that reaches a configuration C_l in which P_l covers a set R_l of l multi-writer registers.*

Proof. By induction on l. The lemma holds for the base case $l = 0$, at the initial configuration.

Consider an f-adaptive collect algorithm. Suppose that, as the induction hypothesis, there is a set of l processes P_l and a P_l-only execution α_l that reaches a configuration C_l in which P_l covers a set R_l of l multi-writer registers. Because $active(\alpha_l) = P_l$ and the algorithm is f-adaptive to total contention, it follows that $|cop_n(C_lP_l)| \le f(l+1)$.

If $n > f(l+1) + l + 1$, Lemma 3.2 implies that there is a process $p_{l+1} \notin P_l \cup \{p_n\}$ and a p_{l+1}-only schedule β, such that in configuration $C_l\beta$, p_{l+1} covers a multi-writer register not in R_l. The lemma follows for $\alpha_{l+1} = \alpha_l\beta$ and $P_{l+1} = P_l \cup \{p_{l+1}\}$. $\square$

Apply Lemma 3.3 with $l = \max\{i|n > f(i) + i\}$. Then any n-process collect algorithm that is f-adaptive to total contention uses at least l multi-writer registers. The lower bound of [23] implies that for an f-adaptive collect algorithm, $f(k) \in \Omega(k)$. This implies that $l \in \Omega(f^{-1}(n))$, and yields our first main result.

Theorem 3.4. *An n-process collect algorithm that is f-adaptive to total contention requires $\Omega(f^{-1}(n))$ multi-writer registers.*

3.2 Lower Bounds on the Number of Concurrent-Write Registers

This section extends Theorem 3.4 to bound the number of concurrent-write registers required for collect, even when an unlimited number of exclusive-write registers are available. We prove somewhat stronger results: we bound the number of concurrent-write registers required for collect, even when c-write registers are available. Let $T(C)$ denote the set of c-write registers that p_n reads during $cop_n(C)$. Recall that $S(C)$ denotes the set of single-writer registers that p_n reads during $cop_n(C)$.

Lemma 3.5. *Let α_l be an execution that reaches a configuration C_l in which a set P_l of l processes cover a set R_l of l concurrent-write registers, and $active(\alpha_l) = P_l$. Let $C = C_lP_l$. For every set Q of $m = c \cdot |T(C)| + 1$ processes disjoint from $S(C) \cup P_l \cup \{p_n\}$, there is a Q-only schedule γ such that, in configuration $C_l\gamma$, some process $q \in Q$ covers a concurrent-write register not in R_l and $active(\alpha_l\gamma) = P_l \cup \{q\}$.*

Proof. Suppose $Q = \{p_{i_1}, \ldots, p_{i_m}\}$. Let R' be the set of multi-writer registers not in R_l that are read by p_n in $cop_n(C_lP_l)$.

Let $j \in \{0, \ldots, m\}$ and let $P'_j = \{p_{i_1}, \ldots, p_{i_j}\}$. We claim there is a P'_j-only schedule β_j such that $C_lP_l \overset{cop_n}{\approx} C_l\beta_jP_l$ and, in configuration $C_l\beta_j$, the processes in P'_j all cover multi-writer registers in R'.

This is vacuously true for $j = 0$, taking β_0 to be the empty schedule. So suppose that $j > 0$ and β_{j-1} is a P'_{j-1}-only schedule such that $C_lP_l \overset{cop_n}{\approx} C_l\beta_{j-1}P_l$ and, in configuration $C_l\beta_{j-1}$, the processes in P'_{j-1} all cover multi-writer registers in R'.

Let β be the p_{i_j}-only schedule guaranteed by Lemma 3.1 such that, in the execution of β starting from configuration $C_l\beta_{j-1}$, process p_{i_j} covers a multi-writer register R' and $C_l\beta_{j-1}P_l \overset{cop_n}{\approx} C_l\beta_{j-1}\beta P_l$. Thus p_n reads the same registers when performing cop_n starting from configuration C_lP_l or from configuration $C_l\beta_jP_l$. Hence the claim is true for j, taking $\beta_j = \beta_{j-1}\beta$. (See Fig. 2.)

Since at most c processes may concurrently cover a register in $T(C)$ and $|Q| = c \cdot |T(C)| + 1$, there exists $j \in \{1, \ldots, m\}$ such that p_{i_j} does not cover a c-write register at $C_l\beta_m$. Thus p_{i_j} covers a concurrent-write register in R'. Let k be the minimum such j. Let γ denote the schedule $\beta_k\gamma_1 \cdots \gamma_{k-1}$, where γ_j is a p_{i_j}-only schedule that ends when p_{i_j} first becomes inactive in the execution starting from C_l. Then, at configuration $C\gamma$, process p_{i_k} covers a concurrent-write register not in R_l and $active(\alpha_l\gamma) = P_l \cup \{p_{i_k}\}$. $\square$

We use this lemma to derive a space lower bound for collect algorithms that are adaptive to point contention.

LEMMA 3.6. *Consider an n-process collect algorithm that is f-adaptive to point contention. For any l such that $n > c \cdot f(l)+l$ there is an execution α_l that reaches a configuration C_l in which a set P_l of l processes cover a set of l concurrent-write registers R_l and active$(\alpha_l) = P_l$.*

PROOF. The proof is by induction on l, with a simple base case $l = 0$ at the initial configuration.

Consider a collect algorithm that is f-adaptive to point contention. By the induction hypothesis there is an execution α_{l-1} that reaches a configuration C_{l-1}, in which a set P_{l-1} of $l-1$ processes cover a set of $l-1$ concurrent-write registers R_{l-1} and *active*$(\alpha_{l-1}) = P_{l-1}$. Because the algorithm is f-adaptive to point contention, $|cop_n(C_{l-1}P_{l-1})| \leq f(l)$.

If $n > c \cdot f(l) + l \geq c \cdot |cop_n(C_{l-1}P_{l-1})| + (l-1) + 1$, Lemma 3.5 implies that there is a schedule γ such that at the end of $C_{l-1}\gamma$, some process $p_l \notin P_{l-1}$ covers a concurrent-write register $r \notin R_{l-1}$ and *active*$(\alpha_{l-1}\gamma) = P_{l-1} \cup \{p_l\}$. Then the claim is true for l, since $\alpha_l = \alpha_{l-1}\gamma$ reaches a configuration C_l, in which a set $P_l = P_{l-1} \cup \{p_l\}$ of l processes cover a set of l concurrent-write registers $R_l = R_{l-1} \cup \{r\}$ and *active*$(\alpha_l) = P_l$. □

Applying Lemma 3.6 with $l = \max\{i | n > c \cdot f(i) + i\}$ and solving the inequality yields the next theorem.

THEOREM 3.7. *An n-process collect algorithm that is f-adaptive to point contention requires $\Omega(f^{-1}(\frac{n}{c}))$ concurrent-write registers, even if it uses an unlimited number of c-write registers.*

When $c = 1$, Theorem 3.7 implies that an n-process collect algorithm that is f-adaptive to point contention requires $\Omega(f^{-1}(n))$ concurrent-write registers, even if it uses an unlimited number of exclusive-write registers.

The point contention at the end of execution α_l (constructed in the proof of Lemma 3.6) is l. Unfortunately, the total contention during α_l is much higher. We bound it using the function $g : \mathbf{N} \to \mathbf{N}$ which grows much more quickly than f and is defined as follows:

$$g(l) = \begin{cases} 0 & \text{if } l = 0 \\ c \cdot f(g(l-1)+1) + l & \text{if } l > 0. \end{cases}$$

LEMMA 3.8. *Consider a collect algorithm that is f-adaptive to total contention. For any l such that $n > g(l)$ there exists a $\{p_1 \dots p_{g(l)}\}$-only execution α_l that reaches a configuration C_l, in which a set R_l of l processes cover a set R_l of l concurrent-write registers, and active$(\alpha_l) = P_l$.*

PROOF. The proof is by induction on l, with a simple base case $l = 0$ at the initial configuration.

By the induction hypothesis, there is a $\{p_1 \dots p_{g(l-1)}\}$-only execution α_{l-1} that reaches a configuration C_{l-1} in which a set P_{l-1} of $l-1$ processes cover a set R_{l-1} of $l-1$ concurrent-write registers and *active*$(\alpha_{l-1}) = P_{l-1}$. Because the algorithm is f-adaptive to total contention, $|cop_n(C_{l-1}P_{l-1})| \leq f(g(l-1)+1)$.

Since $n > g(l) = c \cdot f(g(l-1)+1)+l \geq c \cdot |cop_n(C_{l-1}P_{l-1})| + (l-1)+1$, Lemma 3.5 implies that there is a $(\{p_1 \dots p_{g(l)}\} \setminus P_{l-1})$-only schedule β, such that in configuration $C_{l-1}\beta$, some process p_l covers a concurrent-write register not in R_{l-1} and *active*$(\alpha_{l-1}\beta) = P_{l-1} \cup \{p_l\}$. Thus, the lemma holds since $\alpha_l = \alpha_{l-1}\beta$ is a $\{p_1 \dots p_{g(l)}\}$-only execution that reaches a configuration C_l, in which a set $P_l = P_{l-1} \cup \{p_l\}$ of l processes cover a set R_l of l concurrent-write registers, and *active*$(\alpha_{l-1}\beta) = P_l$. □

Applying Lemma 3.8 with $l = \max\{i | n > g(i)\}$ implies that an n-process collect algorithm that is f-adaptive to total contention, requires at least $g^{-1}(n) - 1$ concurrent-write registers.

THEOREM 3.9. *An n-process collect algorithm that is f-adaptive to total contention, requires at least $g^{-1}(n) - 1$ concurrent-write registers, even if it uses an unlimited number of c-write registers.*

4. THE CONTENTION-FREE STEP COMPLEXITY OF WEAK TEST&SET AND COLLECT

This section studies the contention-free step complexity of one-shot collect and weak test&set objects, implemented using only c-write registers. The *contention-free step complexity* of an algorithm is the maximum step complexity of a single process p running in a p-only execution from the initial configuration.

A one-shot *weak test&set* object supports a test&set operation that can either *succeed* (in which case we say the operation *wins* the object or that it *owns* the object) or *fail*. In every execution, at most one test&set operation succeeds. A test&set operation by p_i must succeed in a p_i-only execution.

A long-lived weak test&set object also supports a reset operation, which can be invoked by a process owning the object to release it. As for one-shot weak test&set, at most one process can own the object at any configuration. If no process owns the test&set object and there are no pending operations on the object at a configuration C, then a p_i-only execution of a test&set operation starting from C must succeed.

It is easy to implement (long-lived) weak test&set using collect: a test&set operation by p executes STORE(1), and then executes COLLECT to obtain a view V. If, for some process $q \neq p$, $V(q) \neq \perp$, then the operation fails and p executes STORE($\perp$); otherwise, the operation succeeds. A reset operation executes STORE($\perp$).

4.1 Weak Test&Set Algorithms

There is a one-shot weak test&set implementation with $O(\log_c n)$ step complexity, using $O(n/c)$ c-write registers. The algorithm uses a complete c-ary tree of depth $\lceil \log_c n \rceil$ as a tournament tree. Every process is assigned to a different leaf and there is a *splitter* [25, 7] assigned to each internal node. To perform a weak test&set operation, a process traverses the nodes on the path from its leaf to the root. Only if p_i wins a node, does it continue to the node's parent. The operation succeeds if p_i wins at the root of the tree; otherwise, the operation fails. Note that since the number of processes trying to win the splitter at any node is bounded by c, the splitter implemention [25] uses only two c-write registers.

There is a similar one-shot weak test&set implementation with $O(\log n)$ step complexity, using $O(n)$ dynamic single-writer registers. We use a complete binary tree of depth $\lceil \log_2 n \rceil$ as a tournament tree, but replace the splitter at

each internal node with two Boolean flags, one for each child. These flags are initially *false*. When a process comes to a node from a child, it sets the flag for that child to *true* and reads the other flag. If the other flag is *false*, the process wins this node and proceeds to its parent. Otherwise, the process fails.

A long-lived version of this algorithm uses exclusive-write instead of dynamic single-writer registers; if a process fails to win some node, it undoes all its writes in reverse order (i.e. writes *false* to all the registers it wrote *true* to, in reverse order). A reset operation also writes *false* to all the registers to which it wrote *true*, in reverse order.

4.2 Lower Bounds

We now give lower bounds on the contention-free step complexity of one-shot collect and weak test&set.

Lemma 3.5 can be used to derive a lower bound on the latency and, hence, the contention-free step complexity of one-shot collect, when concurrent-write registers are not used.

THEOREM 4.1. *An n-process collect which uses only c-write registers has latency $\Omega(\frac{n}{c})$.*

PROOF. Let C be the initial configuration. To obtain a contradiction, suppose that p_n accesses fewer than $\frac{n-1}{c}$ different registers in $cop_n(C)$. Then $|S(C)| + |T(C)| < \frac{n-1}{c}$. Thus, there is a set Q of $c \cdot |T(C)| + 1$ processes disjoint from $S(C) \cup \{p_n\}$. By Lemma 3.5, with $l = 0$, $\alpha_l = \epsilon$, $P_l = \varnothing$, and $R_l = \varnothing$, there is a Q-only schedule γ such that, in configuration $C_0\gamma$, some process $q \in Q$ covers a concurrent-write register. This is a contradiction. □

Theorem 4.1 implies that c-write registers do not help in reducing the contention-free step complexity of collect. Recall from Section 4.1 that even exclusive-write registers suffice for faster implementations of one-shot weak test&set. This implies a gap in the contention-free step complexity of one-shot collect and one-shot weak test&set, when using only c-write registers.

Next, we prove a lower bound of $\Omega(\frac{\log n}{\log c + \log\log n})$ on the number of steps performed in a solo execution of a weak test&set operation when there are no concurrent-write registers.

For the proof, it is helpful to limit attention to a restricted class of executions. A *t-round* execution is a P-only execution for some subset of processes P such that each process takes at most t steps and the processes in P take steps in rounds. This means that no process in P takes its $(k+1)$'st step before any process in P takes its k'th step. We also require that, in a given round, all reads occur before all writes.

An execution starting from some configuration C is *independent*, if for every process p, each time p reads from a register, either that register was not previously written to, or p was the last process to write to that register. Intuitively, a process cannot distinguish an independent execution from a solo execution. Note that any p-only execution is independent.

A process can be erased from an execution by removing all its steps. Note that, in an independent execution, this does not affect the steps of the other processes.

LEMMA 4.2. *Consider any algorithm for n processes that uses only c-write registers. For every non-negative integer t, there is a Q_t-only t-round independent execution α_t, where Q_t is a set of at least $\frac{n2^t t!}{c^t(2t)!}$ processes.*

PROOF. The proof is by induction on t. The base case $t = 0$ holds with Q_0 being the set of all processes and α_0 being the empty execution. Furthermore, if $c^t(2t)!/2^t t! \geq n$, then α_t can be any solo execution of length t.

Let $t \geq 0$ and suppose $c^{t+1}(2(t+1))!/2^{t+1}(t+1)! < n$. Assume that, as an induction hypothesis, there is a Q_t-only t-round independent execution, α_t, where Q_t is a set of at least $\frac{n2^t t!}{c^t(2t)!}$ processes. Consider the next step performed by each process in Q_t (immediately after α_t).

Let $G(Q_t, E)$ be the undirected graph, where $\{p, q\} \in E$ if and only if $p \neq q$ and the next step by one of these processes reads a register written by the other during α_t. Since at most c processes may write to a register concurrently, each register is written to by at most ct different processes during α_t.

Thus, $|E| \leq ct|Q_t|$. By Turan's Theorem [12], there exists an independent set Q_{t+1} in $G(Q_t, E)$ such that $|Q_{t+1}| \geq \frac{|Q_t|^2}{|Q_t|+2|E|} \geq \frac{|Q_t|}{1+2ct} \geq \frac{|Q_t|}{c(1+2t)} \geq \frac{n2^t t!}{c^t(2t)!c(2t+1)} = n\frac{2^{t+1}(t+1)!}{c^{t+1}(2(t+1))!}$.

Let α'_t be obtained from α_t by erasing all processes in $Q_t \setminus Q_{t+1}$. Let α_{t+1} be any Q_{t+1}-only execution obtained by extending α'_t with the next step of each process in Q_{t+1} (if it exists), so that all reads in the last round precede all writes.

By the induction hypothesis, each read in α'_t is from a register that either was not previously written or was last written by the same process. By construction, each read in the last round of of α_{t+1} has the same property. Thus α_{t+1} is a $(t+1)$-round independent execution. □

Consider any positive integer t such that $c^t(2t)!/2^t t! < n$ and suppose there is a weak test&set algorithm that halts within t steps in every solo execution. By Lemma 4.2, there is a Q_t-only t-round independent execution α_t, where Q_t contains at least two processes. Say $p_i, p_j \in Q_t$. Since p_i wins the weak test&set in a t-step p_i-only execution, and p_j wins in a t-step p_j-only execution, both win in α_t. This contradicts the correctness of the weak test&set algorithm.

By Stirling's approximation, if $c^t(2t)!/2^t t! \geq n$, then $t \in \Omega(\frac{\log n}{\log c + \log\log n})$.

THEOREM 4.3. *An n-process one-shot weak test&set object, which does not use concurrent-write registers, has a solo execution with $\Omega(\frac{\log n}{\log c + \log\log n})$ steps.*

When only dynamic single-writer registers are used for implementing a one-shot weak test&set object, the proof can be modified to show the following lower bound, which implies that the algorithm in Section 4.1 is optimal:

THEOREM 4.4. *An n-process weak test&set object, which uses only dynamic single-writer registers, has a solo execution with $\Omega(\log n)$ steps.*

5. DISCUSSION

This paper proves lower bounds on the memory requirements and the contention-free step complexity of collect and related problems. The lower bounds indicate that significant contention for the memory is needed in order to achieve adaptivity.

Our lower bounds are on the number of multi-writer or concurrent-write registers *used* in a single execution of a

collect algorithm. They match, for example, the number of multi-writer registers used by the linear time collect algorithm in [9]. Observe, however, that algorithms typically allocate the registers used in all possible executions. For example, the algorithm in [9] allocates an exponential number of multi-writer registers. Lower bounds on the number of registers used, as proved in this paper, clearly imply lower bounds on the number of allocated registers. Unfortunately, lower bounds derived in this manner are far away from the number of registers allocated by known algorithms, and new techniques seem to be needed in order to derive optimal bounds.

Several variants of the collect problem have appeared in the literature [2, 3, 8]. Our proofs do not require a *regularity* property among collect operations, and our lower bounds hold for a weak variant of the collect problem called *gather* [2].

Implementing weak test&set can be significantly easier than implementing one-shot collect. For example, a splitter implements one-shot weak test&set in a constant number of steps, using only two concurrent-write registers. Thus, the space lower bounds in Section 3 do not extend to one-shot weak test&set. Furthermore, the memory lower bounds do not apply to weak test&set nor do they apply to the *renaming* problem. Extending our proofs to derive lower bounds on the number of multi-writer and concurrent-write registers needed for these problems is an interesting research direction.

The contention-free step lower bounds obviously apply to *obstruction free* [21, 22] implementations of such objects.

Acknowledgements. We thank David Hay for helpful comments. Faith Ellen Fich was supported by the Natural Sciences and Engineering Research Council of Canada and Sun Microsystems.

6. REFERENCES

[1] Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived & adaptive objects (extended abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. ACM Press, 2000.

[2] 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.

[3] 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.

[4] M. K. Aguilera, B. Englert, and E. Gafni. Uniform solvability with a finite number of mwmr registers. In *Proceedings of the 17th International Conference on Distributed Computing*, pages 16–29. Springer-Verlag, 2003.

[5] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Conference on Distributed Computing*, Oct. 2000.

[6] J. H. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, 2002.

[7] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[8] H. Attiya and A. Fouren. Algorithms adaptive to point contention. *J. ACM*, 50(4):444–468, July 2003.

[9] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[10] H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[11] H. Attiya and I. Zach. Fully adaptive algorithms for atomic and immediate snapshots. www.cs.technion.ac.il/~hagit/pubs/AZ03.pdf, Oct. 2003.

[12] C. Berge. *Graphs and Hypergraphs*. North Holland Publishing Company, 1973.

[13] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, Dec. 1993.

[14] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory systems. *J. ACM*, 44(6):779–805, Nov. 1997.

[15] P. Fatourou, F. Fich, and E. Ruppert. A tight time lower bound for space-optimal implementations of multi-writer snapshots. In *Proceedings of the 35th ACM Symposium on Theory of Computing*, pages 259–268. ACM Press, 2003.

[16] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[17] F. E. Fich. The complexity of computation on the parallel random access machine. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 843–899. Morgan-Kaufmann, 1993.

[18] F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2–3):121–163, 2003.

[19] D. Hendler and N. Shavit. Operation-valency and the cost of coordination. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 84–91, 2003.

[20] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[21] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[23] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[24] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook*

of Theoretical Computer Science, volume Volume A: Algorithms and Complexity, pages 869–942. 1990.

[25] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.

APPENDIX

A. LOWER BOUNDS FOR SENSITIVE OBJECTS

The key for the lower bounds proved in Section 3 and for the proof of Theorem 4.1 is the fact that just before the block write, it is possible to let a process store a new value, so that p_n must collect this new value. Here, we generalize the result in three ways. First, the algorithm can be *nondeterministic*, for example, randomized, provided it has the *nondeterministic solo termination* property [16]. Second, the algorithm can use *historyless* objects [16], not just registers. Finally, the lower bounds holds for any *sensitive* object, in particular, increment, fetch&add, modulo-b counter (for $b \geq 2n$), LL/SC bit, and b-valued compare&swap (for $b \geq n$).

Modelling Nondeterministic Algorithms. The state machine of a process can be nondeterministic, so there may be more than one possible p-only execution from a configuration C. We require the *nondeterministic solo termination* property [16, 23]: for every process p and every configuration C, there is a p-only execution from C in which p completes its procedure within a finite number of steps. A nondeterministic algorithm is f*-adaptive* if there is a p-only execution from C in which p completes its procedure within $f(k)$ steps.

Historyless Objects. Let $op(\sigma, Y)$ be the state of object Y that results if operation op is applied to Y when it is in state σ. An operation op is *trivial* if its application does not change the state, that is, $op(\sigma, Y) = \sigma$. Operation op' *overwrites* operation op if applying op and then op' results in the same state as simply applying op', that is $op'(op(\sigma, Y), Y) = op'(\sigma, Y)$. An object type is *historyless* [16, 23], if all its non-trivial operations overwrite one another. Examples of historyless objects include registers, test&set objects, and swap registers.

A process p_i *covers* a historyless object Y in configuration C if p_i is about to apply a non-trivial operation on to Y.

A historyless object Y is *owner-access* if only one process can apply non-trivial operations to Y in every execution; otherwise, Y is *multi-access*. Y is c*-access* if at any configuration, at most c processes cover Y. Y is *concurrent-access* if there is no restriction on the processes that may apply non-trivial operations to Y.

Sensitive Objects. Intuitively, an object is sensitive if every process can always invoke a sequence of operations that must be noticed by later operations. For example, a collect object is sensitive since any process can invoke a STORE operation that should be observed by a later COLLECT operation; on the other hand, weak test&set is not sensitive, since a process not owning the object cannot change its state.

Formally, an object is *sensitive* if, for any p_n-free execution $\alpha\delta$ where no process appears more than once in δ, there is an operation sop_n by p_n such that, for every process $p_l \neq p_n$ that does not appear in δ, there is a sequence of operations and a corresponding schedule $\gamma \in \{p_l\}^*$, such that p_n returns a different result when executing sop_n after $\alpha\delta$ and after $\alpha\gamma\delta$.

For example, increment, fetch&add, modulo-b counter (for $b \geq 2n$), LL/SC bit, and b-valued compare&swap (for $b \geq n$) are all sensitive objects.

We start by proving the analogue of Lemma 3.1:

LEMMA A.1. *Let α_l be a p_n-free execution that reaches a configuration C_l, in which a set P_l of l processes cover a set R_l of l multi-access historyless objects, and $P_l \subseteq active(\alpha_l)$. Let $C = C_l P_l$. Then for every process $p \notin S(C) \cup active(\alpha_l) \cup \{p_n\}$, there is a p-only schedule β such that, in configuration $C_l\beta$, p covers a multi-access historyless object $Y \notin R_l$ that is accessed in $sop_n(C)$, and $C_l P_l \overset{sop_n}{\approx} C_l \beta P_l$.*

PROOF. Let $\alpha = \alpha_l$ and let δ be a block write by the processes in P_l. Recall that the object is sensitive and note that each process appears at most once in δ and that p does not appear in δ. Consider the p-only execution of operations from C_l, guaranteed by the sensitiveness of the object, and let γ be the corresponding schedule. Let $C'_l = C_l\gamma$. If p does not apply a non-trivial operation to a multi-access historyless object $Y \notin R_l$ that is accessed during $sop_n(C)$, then $\alpha\delta C_l P_l \overset{sop_n}{\approx} C'_l P_l \alpha\gamma\delta$, because the processes in P_l also cover the historyless objects in R_l in configuration C'_l. Therefore p_n returns the same result in both $cop_n(C_l P_l)$ and $cop_n(C'_l P_l)$, contradicting the fact that the object is sensitive.

Let β be the prefix of γ up to, but not including p's first application of a non-trivial operation to a multi-access historyless object $Y \notin R_l$ that is accessed during $cop_n(C)$. Then in configuration $C_l\beta$, process p covers Y and $C_l P_l \overset{sop_n}{\approx} C_l \beta P_l$. □

Repeating the proofs in Section 3, substituting Lemma 3.1 by Lemma A.1, and cop_n by sop_n, gives the same space complexity lower bounds for any sensitive object.

THEOREM A.2. *An f-adaptive implementation of a sensitive object for n processes requires $\Omega(f^{-1}(n))$ multi-access historyless objects.*

THEOREM A.3. *An f-adaptive implementation of a sensitive object for n processes that adapts to point contention requires $\Omega(f^{-1}(\frac{n}{c}))$ concurrent-access historyless objects, even if an unlimited number of c-access historyless objects is available.*

THEOREM A.4. *An f-adaptive implementation of a sensitive object for n processes that adapts to total contention requires at least $g^{-1}(n) - 1$ concurrent-access historyless objects, even if an unlimited number of c-access historyless objects is available.*

THEOREM A.5. *An n-process implementation of a sensitive object, which does not use concurrent-write registers, has latency $\Omega(\frac{n}{c})$.*

Jayanti et al. [23] define *perturbable* objects. The difference between perturbable objects and sensitive objects is that for the former, the perturbing execution (γ) has to exist for some process not in $\{p_n\} \cup \delta$ while for the latter,

γ has to exist for every process not in $\{p_n\} \cup \delta$. Clearly, every sensitive object is also perturbable. The increment, fetch&add, modulo-b counter (for $b \geq 2n$), LL/SC bit, and b-valued compare&swap (for $b \geq n$) objects are shown to be perturbable [23]. Careful examination of the proofs reveal that they show that the objects are in fact, sensitive.

To understand why a stronger definition is needed, consider the *prefix collect* problem, which is a variant of the collect problem. Let l be the minimum id of a process that takes no step in a finite execution α'. If all processes participate in α', let $l = n + 1$. Then a COLLECT operation starting after α' has to return only the values stored by processes $p_1, \ldots, p_{l-1}$.

There is a simple adaptive prefix collect algorithm using only single-writer registers: a STORE operation by p_i writes the value to r_i, which is initially $\perp$. A COLLECT operation reads $r_1, r_2, \ldots$ until it reads $\perp$, namely, until it reaches a register of a process that has not started yet. The operation returns the sequence of values it collected.

There is an object based on the prefix collect problem that can be implemented in a similar manner. Because this object is perturbable, it follows that there are perturbable objects which have adaptive implementations without using multi-writer registers.