Poly-Logarithmic Adaptive Algorithms Require Unconditional Primitives

Hagit Attiya[*1] **and Arie Fouren**[†2]

1 **Department of Computer Science, Technion, Haifa 32000, Israel**
hagit@cs.technion.ac.il

2 **Faculty of Business Administration, Ono Academic College, Kiryat Ono, 5545173, Israel**
aporan@ono.ac.il

Abstract

This paper studies the step complexity of adaptive algorithms using primitives stronger than reads and writes. We first consider *unconditional* primitives, like fetch&inc, which modify the value of the register to which they are applied, regardless of its current value. Unconditional primitives admit snapshot algorithms with $O(\log k)$ step complexity, where k is the total or the point contention. These algorithms combine a renaming algorithm with a mechanism for propagating values so they can be quickly collected.

When only *conditional* primitives, e.g., compare&swap or LL/SC, are used (in addition to reads and writes), we show that any collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$. The lower bound applies for snapshot and renaming, both one-shot and long-lived. Note that there are snapshot algorithms whose step complexity is polylogarithmic in n using only reads and writes, but there are no adaptive algorithms whose step complexity is polylogarithmic in the contention, even when compare&swap and LL/SC are used.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases collect, atomic snapshot, renaming, fetch&inc, compare&swap

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.36

1 Introduction

Collecting up-to-date information from all processes is a key to coordination and synchronization, for example, for implementing *atomic snapshots* [1] or solving *renaming* [7]. A simple way to do so is to have an array where each process *stores* its latest value in the entry associated with its index, and to read this array in order to *collect* the values of all processes.

This scheme is an overkill when only a few processes participate in the algorithm: many entries are read from the array although they contain irrelevant information about processes not wishing to coordinate. Better performance is achieved when the step complexity depends only on the *total contention*, namely, the number of processes that participate in the algorithm. We say that such an algorithm is *adaptive to total contention*. Even better is an algorithm whose step complexity is adaptive to *point contention*, which is the maximal number of processes simultaneously executing the algorithm concurrently.

* This work is supported by the Israel Science Foundation (grant number 1749/14), and by Yad HaNadiv foundation.
† This work is supported by the Ono Academic College Research Fund.

Algorithm	Problem	Step Complexity	Contention	Primitives
Algorithm 1, using [22]	atomic snapshot	$O(\log(k))$	point	LL/SC, fetch&inc, bounded-fetch&dec
Algorithm 1, using fetch&inc	atomic snapshot	$O(\log(k))$	total	LL/SC, fetch&inc
Algorithm 2	atomic snapshot	$O(\min(k, \log n))$	point	LL/SC

Figure 1 Summary of upper bounds for long-lived atomic snapshots.

There has been significant progress in designing adaptive collect algorithms that only use reads and writes. These algorithms have step complexity that is at least linear in the total or point contention [3, 12, 2]. They have been used in algorithms for atomic and immediate snapshots, renaming and timestamping [9, 4, 10, 12, 11].

Much less is known about the complexity of adaptive collect using primitives stronger than reads and writes. A notable exception is the collect algorithm of Moir et al. [17], which uses test&set. This algorithm uses less memory than collect algorithms that use only reads and writes, but its step complexity is not much better, as it is linear in the point contention.

This paper studies the step complexity of adaptive algorithms using primitives stronger than reads and writes and investigates whether they can be used to obtain adaptive algorithms with poly-logarithmic step complexity.

We present a snapshot algorithm with $O(\log k)$ step complexity, where k is the total contention; the algorithm uses fetch&inc (as well as LL/SC). We also describe a snapshot algorithm with $O(\log k)$ step complexity, where k is point contention, using fetch&inc and bounded-fetch&dec primitives (and LL/SC). These algorithms combine a renaming algorithm for having processes obtain unique locations to store their values, with a mechanism for propagating these values up a tree ("bubbling" them up), so they can be quickly collected. (See Table 1.)

These algorithms use *unconditional* primitives, like fetch&inc and bounded-fetch&dec, which modify the value of the register to which they are applied regardless of the current value of the register. In contrast, *conditional* primitives [15], like compare&swap and LL/SC, modify the register only if it holds a specific value that depends on the input of the conditional operation. When only conditional primitives are used (in addition to reads and writes), we show that any collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$. Specifically, the bound applies for the sum of the step complexities of a pair of store and collect operations performed by some process on a one-shot collect object.

There is a trade-off between the contention k and the step complexity lower bound of collect. If the total contention k is in $\Theta(n)$, we prove that at least $O(\log k)$ steps are required to perform store and collect. We present an algorithm with $O(\min(k, \log n))$ step complexity, matching the lower bounds.

Clearly, the lower bound for collect immediately applies for snapshots. Moreover, the reduction from collect to renaming, described earlier, implies that the lower bound also applies to renaming, giving an alternative proof of the $\Omega(k)$ step complexity lower bound previously proved by Alistarh et al. [5]. Obviously, it also implies a $\Omega(k)$ lower bound for the long-lived versions of collect, snapshot and renaming, where k is point contention.

Related Work. The work of Fich, Hendler and Shavit [15] showed a separation (in terms of space complexity) between conditional and unconditional primitives. It proves that any

wait-free implementation of a large class of objects that includes counters, stacks, queues and snapshots from reads, writes and conditional primitives requires at least $\Omega(n)$ space. The same space is required for any starvation-free mutual exclusion implemented from conditional primitives. In contrast, using unconditional primitives (like fetch&add) allows to implement these objects and to solve mutual exclusion using only constant space. These results align with the separation in the terms of time complexity presented in our work.

There is an $\Omega(k)$ lower bound on the step complexity of adaptive *mutual exclusion* using reads, writes and compare&swap [21].[1] The lower bound requires that the total number of the processes $n = \Omega(k^{2^k})$, which is slightly higher than the requirement on n in our lower bound.

Alistarh et al. [5] show how to transform any adaptive wait-free renaming with sub-exponential name space $M(k)$ into adaptive mutual exclusion with $O(\log(M(k)) = o(k)$ additional steps. Using the $\Omega(k)$ lower bound on the step complexity of adaptive mutual exclusion [21], they derive an $\Omega(k)$ lower bound on the step complexity of adaptive renaming. This lower bound also requires $n = \Omega(k^{2^k})$.

Another reduction from mutual exclusion to renaming can be obtained using unbalanced tournament tree presented in [8]. For sub-exponential name space $M(k)$ this transformation also requires $O(\log(M(k)) = o(k)$ additional steps. Combined with the $\Omega(k)$ lower bound for mutual exclusion [21], it also provides an $\Omega(k)$ step complexity lower bound for adaptive renaming.

There is a simple implementation of $O(k^2)$-renaming from lattice agreement, with $O(1)$ additional steps [16, Algorithm 11]. Together with the $\Omega(k)$ step complexity lower bound for adaptive renaming [5, 8] it implies an $\Omega(k)$ step complexity lower bound for adaptive lattice agreement (and therefore, for atomic and immediate snapshots).

We could not find a reduction with sublinear step complexity from these problems (mutual exclusion, renaming or snapshots) to adaptive store / collect, even with compare&swap and LL/SC. Without a way to deduce the lower bound for adaptive store-collect from the existing lower bounds for mutual exclusion or renaming, we had to directly prove the lower bound for adaptive collect presented in this paper.

2 The Computation Model

In the *wait-free asynchronous shared-memory* model, n *processes*, $p_0, \ldots, p_{n-1}$ communicate by applying *primitive operations* (in short, *primitives*) to shared memory registers [21]. A process is described as a state machine, with a set of (possibility infinite number of) *states*, one of which is a designated *initial state*, and a state transition function.

The executions of the system are sequences of events. In each *event*, based on its current state, a process applies a primitive to a shared memory register and then changes its state, according to the state transition function. At the beginning of the execution, all shared registers hold the value $\perp$. During an execution, no process ever changes the value of a shared register to $\perp$.

An event ϕ in which a process p applies a primitive op to register R is denoted by a triple $\langle p, R, op\rangle$. An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. There are no constraints on the interleaving of events by different processes, reflecting the assumption

[1] This lower bound holds also for LL/SC by using an implementation of LL/SC from compare&swap, with constant step complexity [19].

that processes are asynchronous. We denote by *next_event*(p_i, α) the next event a process p_i will perform if it is scheduled to take a step after an execution prefix α.

For an execution α and a set of processes P, $\alpha|_P$ is the sequence of all events in α by processes in P; $\alpha|_{\overline{P}}$ is the sequence of all events in α that are *not* by processes in P. If $P = \{p\}$, we write $\alpha|_p$ instead of $\alpha|_{\{p\}}$ and $\alpha|_{\overline{p}}$ instead of $\alpha|_{\overline{\{p\}}}$. An execution α is *P-only* if $\alpha = \alpha|_P$, and it is *P-free* execution if $\alpha = \alpha|_{\overline{P}}$. Two executions α and α' are *equivalent* with respect to P if $\alpha|_P = \alpha'|_P$.

We always assume the availability of read and write primitives. A read(R) primitive returns the current value of R and does not change its value. A write(v, R) operation sets the value of R to v, and does not return a value. We make no restrictions on which process can read from or write to each register, i.e., they are *multi-writer multi-reader*.

A fetch&inc on register R atomically increments the value of the register by 1 and returns the previous value. That is, if the value of R immediately before the invocation of fetch&inc (R) was v, then the primitive sets the value of R to $v + 1$ and returns v. Similarly, fetch&dec atomically decrements the value of the register by 1 and return the previous value. A bounded-fetch&dec primitive is similar to standard fetch&dec, except that if the value of register R is 0 before the primitive is applied to it, then the value of R remains unchanged.

LL on register R returns the current state of R. SC (R, v) invoked by a process p changes the state of R to v only if no other process has changed the value of R since the the latest execution of LL (R) by p. If the value of R is changed SC returns **true**, otherwise it returns **false**.

A compare&swap(R, v, u) primitive works as follows. If the register R holds the value v, then the state of R is changed to u and and **true** is returned (the compare&swap *succeeds*). Otherwise, the state of R remains unchanged and **false** is returned (the compare&swap *fails*).

An *implementation* of a high-level object provides algorithms for each high-level operation supported by the object. Some of the transitions are *requests*, invoking a high-level operation, or *responses* to a high-level operation. When a high-level operation is invoked, the process executes the algorithm associated with the operation, applying primitives to the shared registers, until a response is returned.

In a *well-formed* execution, a high-level operation is invoked only if there is a response to the previous high-level operation, that is, a process alternates between invocations and *matching* responses, beginning with an invocation. A well-formed execution α defines a partial order on operations: If the response of operation op_1 occurs in α before the invocation of operation op_2, then op_1 *precedes* op_2 and op_2 *follows* op_1. We say that op_1 and op_2 are *non-overlapping*.

We require implementations to be *linearizable* [18]. Roughly speaking, a linearizable object guarantees that there is a reordering of the object operations which satisfies the sequential specification of the object and respects the real-time order of non-overlapping operations among all the processes.

Adaptive Algorithms. Let α' be a finite prefix of an execution α. Process p_i performing a high-level operation *op* is *active* at the end of α', if α' includes an invocation of *op* without a return from *op*. The set of the processes active at the end of α' is denoted $active(\alpha')$. The *point contention* at the end of α', denoted $pointCont(\alpha')$, is $|active(\alpha')|$.

The *total contention* during α is the total number of processes active in α:

$$totalCont(\alpha) = \left| \bigcup_{\alpha' \text{ prefix of } \alpha} active(\alpha') \right| .$$

Assume that β is a finite interval β of α, i.e., $\alpha = \alpha_1\beta\alpha_2$. The point contention during β, denoted $pointCont(\beta)$, is the maximum contention in all prefixes $\alpha_1\beta'$ of $\alpha_1\beta$:

$$pointCont(\beta) = \max_{\alpha_1\beta' \text{ prefix of } \alpha_1\beta} pointCont(\alpha_1\beta') \ .$$

Consider an execution α of an algorithm A implementing a high-level operation op. For process p_i executing operation op_i, $step(A, \alpha, op_i)$ is the number of operations on shared registers p_i performs executing op_i in α. The step complexity of A in α, denoted $step(A, \alpha)$, is the maximum of $step(A, \alpha, op_i)$ over all operations op_i of all processes p_i.

Consider a bounded function $\mathrm{S} : \mathcal{N} \mapsto \mathcal{N}$. An algorithm implementing operation op is S-*adaptive to total contention* if for every execution α and every operation op_i with interval β_i, $step(A, \alpha) \leq \mathrm{S}(totalCont(\alpha))$. That is, the step complexity of the algorithm in any execution is bounded by a function of the total contention during the execution. An algorithm implementing operation op is S-*adaptive to point contention* if for every execution α and every operation op_i with interval β_i, $step(A, \alpha, op_i) \leq \mathrm{S}(pointCont(\beta_i))$. That is, the step complexity of an operation op_i with interval β_i is bounded by a function of the point contention during β_i.

Since contention is bounded by n, an operation op_i of p_i terminates within a bounded number of steps of p_i, regardless of the behavior of other processes; that is, adaptive algorithms are *wait-free*.

3 Problems Studied in this Paper

Collect. A solution for the *collect* problem should define algorithms for two operations – store and collect. Intuitively, a store(val) operation of p_i declares val as the latest value for p_i, and a collect operation returns a *view* containing the latest values stored by active processes. A *view* is a set of process-value pairs, $V = \{\langle p_{i_1}, v_{i_1}\rangle, \ldots\}$, without repetitions of processes. $V(p_j)$ refers to v_j, if $\langle p_j, v_j\rangle \in V$, and to $\perp$ otherwise.

A collect operation cop returns a view V such that the following holds for every process p_j:

Validity: If $V(p_j) = \perp$, then no store operation of p_j precedes cop; if $V(p_j) = v \neq \perp$ then v is the value of a store operation sop of p_j that does not follow cop, and there is no other store operation sop' of p_j that follows sop and precedes cop.

That is, cop does not read from the future or miss a preceding store operation.

Moreover, if a collect operation op follows another collect operation cop', then cop should return a view which is more up-to-date. To capture this notion, we define a partial order on views: $V_1 \preceq V_2$, if for every process p_i such that $\langle p_i, v_i^1\rangle \in V_1$, we have $\langle p_i, v_i^2\rangle \in V_2$, and v_i^2 is written in a store operation of p_i that follows or is equal to a store operation of p_i which writes v_i^1. Using this definition, we formulate the property of the collect problem as follows:

Regularity: Assume a collect operation cop by p_i returns V_1, and a collect operation cop' by p_j returns V_2. If cop precedes cop', then $V_1 \preceq V_2$.

Atomic Snapshots. The *atomic snapshot* problem [1] extends the collect problem by requiring views to look instantaneous. We assume a combined upscan operation, which updates a new value and atomically collects a view. The returned views should satisfy the following conditions:

Validity: If an upscan operation op returns a view V, and precedes an upscan operation op', then V does not include the value written by op'.[2]

Self-inclusion: The view returned by the ℓth upscan operation of p_j includes the ℓth value written by p_j.

Comparability: If V_1 and V_2 are the views returned by two upscan operations, then either $V_1 \preceq V_2$ or $V_2 \preceq V_1$.

The *lattice agreement* problem is a special case of atomic snapshots, in which a process performs the algorithm at most once, writing its own identifier and collecting a view which contains identifiers of participating processes. The returned views should satisfy the validity, self-inclusion and comparability properties of the atomic snapshot.

Immediate Snapshots. The *immediate snapshot* problem [14] is an extension of the atomic snapshot problem; it supports a combined im-upscan operation, which updates a new value and returns a view. In addition to the validity, self-inclusion, and comparability properties of the atomic snapshot problem, returned views should satisfy the next condition:

Immediacy: If the view returned by some im-upscan operation, V_1, includes the value written in the ℓth im-upscan of p_j which returns the view V_2, then $V_2 \preceq V_1$.

M-Renaming. In the *long-lived M-renaming* problem, processes $p_1, \ldots, p_n$ with unique names from the range $\{0, \ldots, N-1\}$ repeatedly acquire and release distinct names in the range $\{0, \ldots, M-1\}$. The range $\{0 \ldots N-1\}$ is the *initial name space*, and the range $\{0 \ldots M-1\}$ is the *final name space*. A solution supplies two procedures: getName returning a *new name*, and releaseName; p_i alternates between invoking $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$, starting with $\mathsf{getName}_i$.

For the long-lived renaming problem we redefine the notion of an active process. Process p_i is *active* at the end of execution prefix α', if α' includes an invocation of $\mathsf{getName}_i$ without a return from the matching $\mathsf{releaseName}_i$. A long-lived renaming algorithm should guarantee *uniqueness* of new names: Active processes hold distinct names at the end of α'.

A renaming algorithm has a name space *adaptive to point contention*, if there is a function M, such that the name obtained in an interval β of getName is in the range $\{1, \ldots, \mathrm{M}(\mathit{pointCont}(\beta))\}$. The name space is adaptive to *total* contention, if the new names are in the range $\{1, \ldots, \mathrm{M}(\mathit{totalCont}(\beta))\}$.

One-shot M-renaming is a special case of long-lived renaming. The processes start with unique names from the range $\{0, \ldots, N-1\}$ and are required to choose distinct names in the range $\{0 \ldots M-1\}$, where $M < N$.

4 Sub-linear Adaptive Algorithms for Atomic Snapshots

4.1 Atomic Snapshots Using Renaming

This section presents a modular construction of atomic snapshots using renaming. The algorithm uses an unbalanced binary tree, consisting of a sequence of complete binary trees of growing sizes, connected as shown in Figure 2. (This tree structure was also used in [6, 22].)

In Algorithm 1, a process starts by acquiring a name i using an adaptive renaming algorithm (Line 1.2). Then it exclusively accesses the i-th leaf of the unbalanced binary tree,

[2] Typically, this condition trivially holds and we do not prove it below.

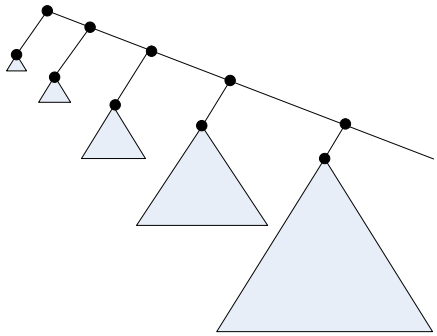

Figure 2 Unbalanced binary tree used in Algorithm 1.

writes its information in the register associated with the leaf. Then, it climbs up to the root, updating the views associated with the nodes on the path. (This part of the algorithm is similar to the f-array implementation of Jayanti [20].) After updating the root, the process releases the acquired name.

The next lemma shows that the sequence of the views stored in a node is monotonically increasing:

▶ **Lemma 1** (Comparability). *Let $V_1, V_2, \ldots$ be the sequence of the views written into v.subtree-View of some node v in Line 1.3, 1.17 or 1.17. Then $V_1 \preceq V_2 \preceq \ldots$.*

Proof. If node v is a leaf, then define a sequence of execution intervals $\gamma_1, \gamma_2, \ldots$ as follows: γ_i starts in the ith time a process p acquires the name associated with leaf v (Line 1.2), and it ends when p releases this name (Line 1.6). By the uniqueness of renaming, the intervals $\gamma_1, \gamma_2, \ldots$ do not overlap.

If v is an inner node, let $\gamma_1, \gamma_2, \ldots$ be the execution intervals corresponding to the pairs of successful LL/SC primitives that write views $V_1, V_2, \ldots$ into *v.subtree-View*. That is, for each successful pair SC_i, the interval γ_i starts with the corresponding LL (Line 1.16 or 1.18) and ends with the corresponding SC (Line 1.17 or 1.19). Clearly, the intervals $\gamma_1, \gamma_2, \ldots$ do not overlap.

In both cases, processes access node v in non-overlapping intervals $\gamma_1, \gamma_2, \ldots$. Therefore, a view V_i written by a process p into *v.subtree-View* in interval γ_i, (in Line 1.3 if v is a leaf, or in Line 1.17 or 1.19 if v is an inner node) is read after that by a process q in the next interval γ_{i+1} (in Line 1.16 or 1.18). Therefore, it is included in the view V_{i+1} written by q into *v.subtree-View* (in Line 1.3, Line 1.17 or 1.19). This implies that $V_i \preceq V_{i+1}$, and the lemma follows by induction on i. ◀

The following lemma states that the views returned by scan operations satisfy the self-inclusion property of atomic snapshot.

▶ **Lemma 2** (Self-inclusion). *Assume p_i performs* update(id_i, val_i)*, and then* scan*() that returns view V_i. Then $val_i \in V_i$.*

Proof. We show by induction on the sequence of the nodes visited by p_i during refresh that $val_i \in v.subtree\text{-}View$ after p_i leaves node v (Line 1.21).

If node v is a leaf, then p_i appends val_i to *v.subtree-View* in Line 1.3. Therefore, $val_i \in v.subtree\text{-}View$ after p_i leaves v.

Algorithm 1 Adaptive atomic snapshot algorithm using renaming

Type:
node :
subtree-View: view of the values stored in the subtree of the node, initially $\emptyset$
left-child: pointer to left child node
right-child: pointer to right child node
parent: pointer to the parent node
Global variables:
T: unbalanced binary tree of nodes
Local variables:
id : process id
val : value to be written; for simplicity we assume increasing numbers

1: **procedure** update(*id*, *val*)
2: i = acquire-name() ▷ using any adaptive long-lived renaming algorithm
3: $v = i$-th leaf of the unbalanced tree T ▷ start at the bottom of the unbalanced tree
4: $v.subtree\text{-}View$ = merge($v.subtree\text{-}View$, $\{\langle id, val\rangle\}$) ▷ update your value in the leaf
5: refresh(v) ▷ start from the current leaf v and ascend back to the root
▷ updating the views in the nodes along the path
6: release-name(i) ▷ release the name associated with the i-th leaf
7: **end procedure**

8: **procedure** scan
9: return($root.subtree\text{-}View$) ▷ return the view from the root node
10: **end procedure**

11: **procedure** merge(views V_1, V_2, ...) ▷ return the view of the latest processors' values
12: return($\{\langle p_i, v_i\rangle \mid v_i = \max(V_1(p_i), V_2(p_i), \ldots)\}$)
13: **end procedure**

14: **procedure** refresh(node v) ▷ start at leaf v and ascend back to the root
15: **while** $v \neq root$ **do** ▷ updating the views in the nodes along the path
16: $view$ = LL($v.subtree\text{-}View$)
▷ merge the views stored v and in both its children and try to store it in $v.subtree\text{-}View$
17: **if** ¬ SC($v.subtree\text{-}View$, merge($view$, $v.right\text{-}child.View$,$v.left\text{-}child.View$) **then**
18: $view$ = LL($v.subtree\text{-}View$) ▷ if the previous store failed, try once more
19: SC($v.subtree\text{-}View$, merge($view$, $v.right\text{-}child.View$,$v.left\text{-}child.View$))
20: **end if**
21: $v = v.parent$ ▷ climb up to the patent node even if both updates failed
22: **end while**
23: **end procedure**

For the induction step, assume that $val_i \in$ *subtree-View* in the left child or the right child of the current node v. Process p_i reads these views before it attempts to write the merged view into *v.subtree-View* with SC (Line 1.17 or 1.19). If one of the SC primitives succeeds, then $val_i \in$ *v.subtree-View* after p_i leaves node v.

If both of these SC primitives fail, then there is a successful pair LL_j and SC_j by some process p_j such that LL_j starts after the first LL of p_i and SC_j ends before the second SC of p_i. Process p_j reads *v.left-child.subtree-View* and *v.right-child.subtree-View* (one of them containing val_i) between LL_j and SC_j. Therefore, SC_j writes a view containing val_i into *v.subtree-View*. By Lemma 1, *v.subtree-View* is monotonically increasing. Therefore, $val_i \in$ *v.subtree-View* after p_i leaves node v.

Process p_i completes refresh after leaving the root. Therefore, $val_i \in$ *root.subtree-View* after p_i returns from update. By Lemma 1, *root.subtree-View* is monotonically increasing. Therefore, the view V_i returned by the following scan operation contains val_i. ◀

By Lemmas 1 and 2 we have that Algorithm 1 implements a long-lived atomic snapshot. Let $M(k)$ be the size of the name space of the adaptive renaming algorithm used in the algorithm. The distance from the $M(k)$-th leaf of the unbalanced tree to the root is $O(\log M(k))$. In each node on the path from the leaf to the root in update, a process performs a constant number of steps. Therefore, the step complexity of the resulting snapshot algorithm is $f(k) + O(\log M(k))$, where $f(k)$ is the step complexity of the adaptive renaming algorithm.

A simple way to do renaming in Line 1 is by applying fetch&inc to a shared register. When a process executes update for the first time, it performs fetch&inc to get a name, and then it uses this name in all the following update operations. The names obtained in this way are in $0, \ldots, k-1$, where k is the total contention. Then the depth of the leaf acquired is $O(\log k)$, and therefore the step complexity of the algorithm is $O(\log k)$, where k is the total contention. The algorithm uses only LL/SC and fetch&inc.

Alternatively, we can use the long-lived adaptive k-renaming of Moir and Anderson [22, Fig. 8]. The step complexity of this algorithm is $O(\log k)$, where k is the point contention. However, it uses bounded-fetch&dec in addition to the more standard LL/SC and fetch&inc. Using this algorithm gives a $O(\log(k))$ long-lived atomic snapshot, where k is point contention, using LL/SC, fetch&inc and bounded-fetch&dec.

4.2 Atomic Snapshot with Conditional Primitives

What is the best step complexity we can achieve with only conditional primitives? When the number of participants is high ($k \sim n$), at least $\Omega(\log n)$ steps are required to perform collect (and therefore atomic snapshot) [13]. This section presents a snapshot algorithm with $O(\min(k, \log n))$ step complexity using only reads, writes and LL/SC. In the next section we prove that at least $\Omega(k)$ steps are required when contention is low ($k \in O(\log \log n)$), and at least $\Omega(\log k)$ steps is required when contention is high ($k \in \Theta(n)$). The algorithm matches the both lower bounds.

Algorithm 2 is a modification of Algorithm 1. It uses an unbalanced tree with the same structure, but the first $\log n$ leaves of the tree are reserved for processes that obtain new names in $\log n$-*restricted* adaptive k-renaming. This restricted renaming algorithm guarantees that if the total (or point) contention is less than $\log n$, then all the processes get new names in range $0, \ldots, k-1$. If the contention is higher than $\log n$, then a process p_i either gets a name in range $0, \ldots, \log n$ or a special **fail** value. If p_i gets **fail** then it accesses the unbalanced tree using its original name id_i, starting at leaf $\log n + id_i$.

Algorithm 2 Adaptive atomic snapshot with $O(\min(k, \log n))$ step complexity, using reads, writes and LL/SC.

```
1:  procedure update(id, val)
2:      i = acquire-name()              ▷ using long-lived k-renaming restricted to log n names
3:      if i == failed then
4:          i = log n + id_i            ▷ if failed to get a name ≤ log n, use its original name + log n
5:      end if
6:      v = i-th leaf of the unbalanced tree T ▷ start at the bottom of the unbalanced tree
7:      v.subtree-View = merge(v.subtree-View, {val})  ▷ update your value in leaf's subtree
        view
8:      refresh(v)                      ▷ start from the current leaf v and ascend back to the root
                                        ▷ updating the views in the nodes along the path
9:      if i < log n then
10:         release-name(i)             ▷ release the name if it was acquired by the adaptive renaming
11:     end if
12: end procedure
```

The correctness of the algorithm follows by the uniqueness of the new names and by Lemmas 1 and 2. We can get a restricted renaming adaptive to point contention by using a sequence of $\log n$ LL/SC variables. A process sequentially accesses these variables until it succeeds to acquire one of them. If the process fails in all $\log n$ variable, it returns **failed**. This is essentially the k-renaming algorithm using test&set [22, Theorem 4], restricted to the first $\log n$ names. The step complexity of the renaming is $O(k)$, adaptive to point contention. Therefore, if $k < \log n$, each process gets a name $\leq k$ in $O(k)$ steps, and accesses a leaf at depth $O(\log k)$. Therefore the total number of steps is $O(k)$. If $k \geq \log n$, then a process accesses the tree using its original name at a leaf on depth $O(\log n)$. Thus the total complexity is $O(\log n)$. This implies that Algorithm 2 correctly implements atomic snapshot with $O(\min(k, \log n))$ step complexity, where k is the point contention.

5 Lower Bounds on Adaptive Collect with Conditional Primitives

This section proves a trade-off between the total contention and the step complexity of adaptive one-time collect using conditional primitives. For low contention, $k \in O(\log \log n)$, at least $\Omega(k)$ steps are required to complete an update operation followed by a collect. If the contention is high, $k \in \Theta(n)$, then at least $\Omega(\log k)$ steps are required to complete this pair of operations. Algorithm 2 presented in Section 4.2 solves the atomic snapshot problem using only conditional primitives (writes, reads and LL/SC) with $O(\min(k, \log n))$ step complexity, thus matching both lower bounds.

To prove the lower bounds, we construct an execution in which each active process performs a store followed by a collect, and show that at least one process performs the required number of steps. In the execution, the active processes are divided to *visible* and *invisible*. Intuitively, an *invisible* process may be removed from the execution without affecting the steps of the others. Formally, process p is *invisible* after an execution prefix α if α and $\alpha|_{\overline{p}}$ are equivalent with respect to any process in $\text{active}(\alpha) - \{p\}$. The set of the processes invisible after α is denoted $\mathit{invisible}(\alpha)$.

The construction proceeds in rounds. In each round, every process that is still invisible after the previous round performs its next computational event. After constructing the new

round, we keep in the execution all the invisible processes, and some of the processes that became visible; the rest are deleted retroactively.

Provided that the initial number of processes is sufficiently high (as stated below), we inductively build r rounds of execution so that at least two processes p and q remain invisible after the last round. By the validity property of collect, it is impossible that two processes complete their store and collect operations without being aware of each other. This implies that at least one of them does not complete its collect in r rounds, implying that it takes at least r steps.

Choosing the number of processes which are allowed to become visible in each round, we can trade the level of contention, k, and the number of rounds, r. If we allow at most one process to become visible, we get an execution with very low contention, $k = O(\log \log n)$, but the number of rounds is linear in k. Increasing the number of processes that become visible increases the contention, but decreases the length of the execution. If we allow a constant fraction of processes to become visible, then we get an execution with high contention $k = \Theta(n)$, but the number of rounds decreases to $\log k$.

The extension by one round relies on the fact that only conditional primitives are used. Instead of defining conditional primitives formally, the lower bound proof uses a more refined classification of primitives, according to their *transparency*, defined as follows.

▶ **Definition 3.** Suppose that after an execution prefix α there is a variable v such that $value(v, \alpha) = \perp$ and there is a subset $P = \{p_1, \ldots, p_k\}$ of invisible processes whose next events $\phi_1, \ldots, \phi_k$ apply the same primitive Op to the same variable v:

$$P = \{p_i | p_i \in invisible(\alpha) \wedge next\_event(p_i, \alpha) = \langle p_i, v, Op \rangle\} .$$

We say that primitive Op is $(k - m)$-*transparent* (for some $m \leq k$), if there is a permutation π of the next events $\phi_1, \ldots, \phi_k$ such that after execution $\alpha\pi$ at most m processes from P become visible, and the rest of the process in $invisible(\alpha)$ remain invisible.

More formally, define W to be the subset of processes P that become visible after $\alpha\ \pi$:

$$W = \{p_i | p_i \in P \wedge p_i \in visible(\alpha\ \pi)\} .$$

The primitive Op is $(k - m)$-transparent if $|W| \leq m$ and every $p_i \in invisible(\alpha) - W$ is in $invisible(\alpha\ \pi)$.

Appendix A shows that read, compare&swap and LL/SC are k-transparent, and that write is $(k - 1)$-transparent.

Suppose that the algorithms uses a constant number of primitive types, $Op_1, Op_2, \ldots, Op_t$. Assume, without loss of generality, that each process applies primitives cyclically in this order during its execution. That is, in its i-th step the process performs a primitive of type $Op_{i \bmod t}$. Any algorithm may be modified in this way by introducing "dummy" primitives of the required type. This increases the step complexity of the algorithm by a constant factor, since the algorithm uses a constant number of primitive types, and does not affect the asymptotic step complexity.

For completeness of the explanation, we state Turán's Theorem [23] used in the induction step of the lower bound proof (Lemma 5).

▶ **Theorem 4** (Turán). *Let $G(V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. If an average degree of G is d, then $G(V, E)$ has an independent set with at least $\lceil |V|/(d + 1) \rceil$ vertices.*

The next lemma provides the induction step for the lower bound proof.

▶ **Lemma 5.** *Suppose that there is an execution α_r containing r rounds such that $|invisible(\alpha_r)| = m_r$. Then for any w, $1 \leq w \leq m_r/2$, there is an execution α_{r+1} containing $r+1$ rounds, such that after α_{r+1} the number of visible processes $|visible(\alpha_{r+1})| = |visible(\alpha_r)| + w$, and the number of invisible processes $m_{r+1} = |invisible(\alpha_{r+1})| \geq 2\sqrt{\frac{m_r \cdot w}{3}} - 2w$.*

Proof. We show how to extend α_r with one round $(r+1)$. In round $r+1$, each process $p_i \in invisible(\alpha_r)$, $1 \leq i \leq k$, executes its next event $\phi_i = next\_event(p_i, \alpha_r)$. Define round $r+1$ as the sequence of these events $\pi_{r+1} = \phi_1, \phi_2, \ldots, \phi_k$, and define a new execution $\alpha'_{r+1} = \alpha_r \ \pi_{r+1}$.

By assumption, all the next events after α_r apply the same primitive in round $r+1$.

In order to keep many processes invisible, we should take care of two things. First, we need to ensure that the events of round $r+1$ do not conflict with events performed in the previous rounds. Otherwise, if process p in round $r+1$ overwrites a variable previously written by another process q, then p can not be further deleted from the execution without making q visible. We eliminate this kind of conflicts using Turàn's Theorem. Next, we eliminate conflicts between primitives in round $r+1$, using the fact they are k- or $(k-1)$-transparent.

Eliminating conflicts with previous rounds. Consider a *visibility* graph $G(V, E)$, with vertices V corresponding to the processes in $invisible(\alpha_r)$. If in round $r+1$, a process p_i accesses one of the variables previously changed by process p_j, then there is an edge $p_i \to p_j \in E$. In round $r+1$, process p_i accesses at most one memory location, and therefore $|E| \leq |V|$ and the average degree of G is $d = 2|E|/|V| \leq 2$. By Turàn's Theorem, G has an independent set $V' \subseteq V$ with at least $\lceil |V|/(d+1) \rceil = \lceil |V|/3 \rceil$ vertices. We leave the processes corresponding to V' in the execution, and delete all the other invisible processes $V - V'$. That is, we define $\alpha''_{r+1} = \alpha'_{r+1}|_{V'}$. Note that in the execution α''_{r+1}, the processes in V' access only variables which were not changed by other processes. Therefore, there are no conflicts between the primitives of round $r+1$ and the primitives of rounds $1, \ldots, r$.

Eliminating conflicts between events in round $r+1$. Let $M(V')$ the set of variables accessed by the processes in V'. Since $|V'|$ processes access $|M(V')|$ different variables, the average number of processes that access the same variable is $\frac{|V'|}{|M(V')|}$. We order the variables in $M(V')$ by the number of processes accessing them, and choose the w variables $\{v_1, \ldots, v_w\} \in M(V')$ that are accessed by the largest number of processes. In round $r+1$ we keep all the processes accessing the variables $\{v_1, \ldots, v_w\}$, and exactly one process for each variable $M(V') - \{v_1, \ldots, v_w\}$. Let V' be the set of these processes (see Figure 3). We define $\alpha_{r+1} = \alpha''_r|_{V''}$.

For each variable $v_i \in \{v_1, \ldots, v_w\}$, we choose exactly one process $p(v_i) \in V''$ that accesses this variable. In round $r+1$, we schedule the process $p(v_i)$ after all the other processes accessing v_i. Thus, in round $r+1$, process $p(v_i)$ covers all the other processes accessing the variable v_i, $p(v_i)$ becomes visible and is stopped. The rest of the processes $V'' - \{p(v_1), \ldots, p(v_w)\}$ remain invisible after round $r+1$. Below we bound from below the number of processes that can be kept invisible after this round.

Note that the number of processes that access each variable $v_i \in \{v_1, \ldots, v_w\}$ is more than the average $\frac{|V'|}{M(V')}$. Therefore, the number of invisible processes is

$$|V''| - w \geq \frac{|V'|}{|M(V')|} \cdot w + (|M(V')| - w) - w = \frac{|V'|}{|M(V')|} \cdot w + |M(V')| - 2w \ .$$

For simplicity of notation, denote $|M(V')| = x$. Using this notation, the number of processes that are invisible after round $r+1$ is $m_{r+1} \geq \frac{|V'|}{x} \cdot w + x - 2w$.

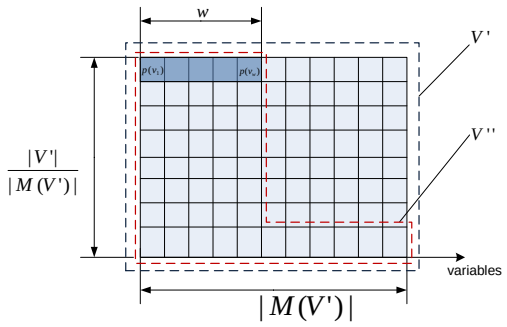

Figure 3 Induction step in the proof of Lemma 5.

Differentiating by x and equating to 0, we get $m'_{r+1}(x) = -\frac{|V'|w}{x^2} + 1 = 0$, implying $x = \sqrt{|V'|w}$. Therefore, the minimal value of m_{r+1} is

$$\frac{|V'|}{\sqrt{|V'|w}} \cdot w + \sqrt{|V'|w} - 2w = 2\sqrt{|V'|w} - 2w \ .$$

Substituting $|V'| = m_r/3$, we get $m_{r+1} \geq 2\sqrt{\frac{m_r \cdot w}{3}} - 2w$. ◀

Different values of w lead to different trade-offs between the total contention and the number of rounds in the execution. For two extreme values of w, we have lower bounds that match the upper bounds:

When $w = 1$, Lemma 5 implies that the number of invisible processes after round $r+1$ is $m_{r+1} \geq 2\sqrt{\frac{m_r}{3}} - 2 \geq \sqrt{\frac{m_r}{3}}$. To prove the lower bound, we need that after r rounds there are at least two invisible processes, i.e., $m_r = 2$. Solving this recurrence, we get that $m_0 = 2^{2^r} 3^{2^r - 1} = \frac{6^{2^r}}{3}$. That is, the total number of the processes in the system should be $n \geq m_0 \in \Omega(6^{2^r})$, or $r = O(\log \log n)$.

In each round of the execution α_r, at most $w = 1$ processes become visible and stopped, and two processes remain invisible after α_r. The rest of processes are deleted from the execution. Therefore the total contention in α_r is $k = r + 2 = \Omega(\log \log n)$. Thus we have an execution with contention $k \in O(\log \log n)$, in which a pair of store and collect operations take at least $\Omega(k)$ steps.

When $w = \frac{|V'|}{2} = m_r/6$, Lemma 5 implies that the number of invisible processes we have after round $r+1$ is $m_{r+1} \geq 2\sqrt{\frac{m_r \cdot w}{3}} - 2w = 2\sqrt{\frac{m_r \cdot m_r/6}{3}} - 2 \cdot m_r/6 = \frac{m_r}{3(\sqrt{2}+1)}$. We need that after r rounds there are at least two invisible processes, i.e., $m_r = 2$. Solving these recurrences, leads to $m_0 = 2\left(3\left(\sqrt{2}-1\right)\right)^r$. Thus, the total number of processes in the system is $n \geq m_0 = 2\left(3\left(\sqrt{2}-1\right)\right)^r$ and hence, $r = \Theta(\log n)$.

In each round r at most $w = m_r/6$ processes become visible and are stopped. Therefore, the total contention in the execution α_r is $k = \sum_i^r \frac{m_i}{6} = \frac{1}{6}\sum_i^r \frac{m_0}{(3(\sqrt{2}+1))^i} = \Theta(m_0) = \Theta(n)$. Thus we have an execution with contention $k \in \Theta(n)$, in which a pair of store and collect operations take at least $r = \Omega(\log k)$ steps.

Thus, we have the following theorem:

▶ **Theorem 6.** *Any implementation of one-time* collect *using $(k-1)$-transparent primitives has an execution of $k+2$ processes in which a pair of* store *and* collect *operations takes at least (a) $\Omega(k)$ steps, provided $k \in O(\log \log n)$, and (b) $\Omega(\log k)$ steps, provided $k \in \Theta(n)$.*

The reduction from collect to $M(k)$-renaming presented in Section 4.1 requires $O(\log M(k))$ additional steps. This implies a linear lower bound on the step complexity of adaptive one-shot renaming with sub-exponential name space $M(k) = 2^{o(k)}$, giving an alternative proof for the lower bound of Alistarh et al. [5].

▶ **Theorem 7.** *An adaptive implementation of one-shot $M(k)$-renaming with sub-exponential name space $M(k) = 2^{o(k)}$ using $(k-1)$-transparent primitives requires at least $\Omega(k)$ steps in an execution with total contention $k \in O(\log \log n)$.*

6 Summary and Open Problems

We have shown that *unconditional* primitives, like fetch&inc, allow snapshot algorithms with $O(\log k)$ step complexity, where k is the total or the point contention. In contrast, when only *conditional* primitives, like compare&swap, are used, any snapshot or collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$.

We also give an adaptive algorithm whose step complexity is in $O(\min\{k, \log n\})$. The algorithm has a linear step complexity $O(k)$ when contention is low (this is optimal for $(k \in O(\log \log n))$, and logarithmic step complexity $O(\log n)$ when contention is high (this is optimal for $k \approx n$). An immediate question is to understand the complexity of adaptive algorithms in the intermediate range, when the contention is in $o(\log n)$, but still growing faster than $\log \log n$.

It is interesting to investigate whether non-standard *bounded* unconditional primitives, like bounded-fetch&dec, are needed in order to get adaptive algorithm with sublinear step complexity as a function of point contention. We believe that sublinear algorithms adaptive to point contention require unconditional primitives that prevent "wrapping around", like bounded-fetch&dec.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.
2. Yehuda Afek and Yaron De Levie. Efficient adaptive collect algorithms. *Distributed Computing*, 20(3):221–238, 2007.
3. Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 262–272. IEEE, 1999.
4. Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 71–80. ACM, 2000.
5. Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014. doi:10.1145/2597630.
6. James Aspnes, Hagit Attiya, and Keren Censoe-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of ACM*, 59:2:2–2:24, 2012.
7. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
8. Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.
9. Hagit Attiya and Arie Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Distributed Computing*, pages 149–163. Springer, 2000.

10 Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002. doi:10.1137/S0097539700366000.

11 Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003. doi:10.1145/792538.792541.

12 Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

13 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 217–226. ACM, 2008.

14 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC'93, pages 91–100, 1993.

15 Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, pages 80–87, 2004.

16 Arie Fouren. *Adaptive Wait-Free Algorithms for Asynchronous Shared-Memory Systems*. PhD thesis, Technion, 2001.

17 Maurice Herlihy, Victor Luchangco, and Mark Moir. Space-and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78:260–280, 2003.

18 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

19 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 285–294. ACM, 2003.

20 Prasad Jayanty. f-arrays: Implementaion and applications. In *Proceedings of the 21th Annual Symposium on Princeples of Distributed Computing (PODC)*, pages 270–279, New York, 2002. ACM.

21 Yong-Jik Kim and James H Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

22 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995.

23 P. Turan. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

A Transparent Primitives

▶ **Claim 8.**

(a) *the* write *primitive is* $(k-1)$*-transparent;*

(b) *the* read *primitive is* k*-transparent;*

(c) *the* compare&swap *primitive is* k*-transparent;*

(d) *the* LL/SC *prmitives are* k*-transparent.*

Proof. According to definition 3, suppose that after an execution prefix α there is a variable v such that $value(\alpha) = \perp$ and there is a subset $P = \{p_1, \ldots, p_k\} \subseteq invisible(\alpha)$ whose next events $\phi_1, \ldots, \phi_k$ contain the same primitive Op on the variable v:

$$P = \{p_i | p_i \in invisible(\alpha) \wedge next\_event(p_i, \alpha) = \langle p_i, v, Op \rangle\}$$

(a) *Op* **is a** write**.** Let $\pi = \langle \phi_1, \ldots, \phi_k \rangle$. In events $\phi_1, \ldots, \phi_k$ processes P do not read any information, therefore processes $\mathit{invisible}(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. All the write events $\phi_1, \ldots, \phi_{k-1}$ are overwritten by the following writes, thus these events are undetectable, and therefore processes $p_1, \ldots, p_{k-1}$ remain invisible also after $\alpha \circ \pi$. (The only process that becomes visible after $\alpha \circ \pi$ is p_k that performs the last write event in π. Since this event overwrites the previous values written to v, it can not be deleted from the execution undetectably). By definition 3 this implies that write is $(k-1)$-transparent.

(a) *Op* **is a** read**.** Define a permutation $\pi = \langle \phi_1, \ldots, \phi_k \rangle$. Since events ϕ_i read from an empty variable v and do not change values of other variables, there is no information flow between the processes in $\mathit{invisible}(\alpha)$, and therefore all the processes invisible after α remain invisible also after $\alpha \circ \pi$. According to definition 3 this imply that read is k-transparent.

(c) *Op* **is a** compare&swap We define the permutation π of the events $\phi_1, \ldots, \phi_k$ as follows. As mentioned in the introduction, we assume that no process attempts to write $\perp$. First, we schedule primitives $\mathsf{CAS}(u, w)$ where $u \neq \perp$. By semantics of CAS these primitives fail and do not change the value of v. Then we schedule the remaining primitives $\mathsf{CAS}(\perp, w)$, where $w \neq \perp$. By atomicity of CAS, only the first of these primitives reads $v = \perp$ and succeeds, while the rest read $v = w \neq \perp$ and fail.

Since none of the events π performed by processes P reads any value written previously by another process, therefore all the processes in $\mathit{invisible}(\alpha) - P$ remain invisible after $\alpha \circ \pi$. The $k-1$ unsuccessful CAS primitives do not change the value of v. Since a successful CAS (that changes the value of v from $\perp$ to w) does not overwrites any value previously written to v by another process, all the events π by processes P can be removed from the execution without affecting the rest of the invisible processes $\mathit{invisible}(\alpha) - P$. Therefore all, the processes P remain invisible after $\alpha \circ \pi$, implying $\mathit{invisible}(\alpha \circ \pi) = \mathit{invisible}(\alpha)$. By definition 3, CAS is k-transparent.

(d) *Op* **is an** LL**.** Define $\pi = \phi_1, \ldots, \phi_k$. All these primitives read $\perp$ from v, thus and they do not get any information written previously by other invisible processes. Therefore all the processes $\mathit{invisible}(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. Since LL primitives do not overwrite any value previously written to v by other processes, all the events π by processes P can be removed from the execution without affecting the steps of other invisible processes. Therefore, $\mathit{invisible}(\alpha \circ \pi) = \mathit{invisible}(\alpha)$. By Definition 3, LL is k-transparent.

Op is an SCI. Let $\pi = \phi_1, \ldots, \phi_k$. By the semantics of LL/SC, only the first SC, corresponding to event ϕ_1 succeeds, and the rest fail. Since SC does not read any value previously written by another process to v, the processes $\mathit{invisible}(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. The $k-1$ unsuccessful primitives do not change the value of v, and the single successful
$SC(w)$, $w \neq \perp$ does not overwrite any value written previously by another process. Therefore, all the computational events π can be removed from the execution without affecting the values of other invisible processes. Thus, all the processes P remain invisible after $\alpha \circ \pi$. By Definition 3, SC is k-transparent. ◀