EFFICIENT IMPLEMENTATIONS OF CONCURRENT SNAPSHOT OBJECTS

Charidimos Kiosterakis

Thesis submitted in partial fulfillment of the requirements for the Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Manolis G.H. Katevenis
Thesis Co-Advisor: Nikolaos Kallimanis

FEBRUARY 1, 2020
UNIVERSITY OF CRETE  
COMPUTER SCIENCE DEPARTMENT  

Efficient Implementations of Concurrent Snapshot Objects  

Thesis submitted by  
Charidimos Kiosterakis  
in partial fulfilment of the requirements for the  
Masters' of Science Degree in Computer Science  

Thesis Approval  

Author:  
Charidimos Kiosterakis  

Committee approvals:  

EMMANOUIL KATEVENIS  
Digitally signed by  
EMMANOUIL KATEVENIS  
Date: 2020.03.17 15:05:46 +02'00'  

Dr. Nikolaos D. Kallimantis  
Thesis Co - Advisor  
Manolis G. H. Katevenis  
Professor, Thesis Supervisor  

Evangelos Markatos  
Professor, Committee Memer  

Kostas Magoutis  
Associate Professor, Committee Memer  

Department approval:  
Antonis Argyros  
Professor, Director of Graduate Studies  

Heraklion, February 2020
Efficient Implementations of Concurrent Snapshot Objects

Abstract

During the last decade, there has been a rise in the use of multicore and many-core systems. Their application covers a huge range, from customer products like smartphones and laptops to servers that equip modern data centers. The multicore and many-core systems can provide better performance only if the user can use their potential, although this is not a trivial task since parallel programming is much harder than serial programming. In order to take the fullest advantage of the power of such a system, utilization of efficient implementations of concurrent data structures is needed.

A snapshot object is a concurrent data structure that has numerous applications in concurrent programming. Snapshots can be used to record the state of the system, so they can provide solutions to problems where an action should be taken when the global state of the system satisfies some conditions. Furthermore, snapshots have been widely used for the design and verification of distributed algorithms such as the construction of concurrent timestamps and randomized consensus.

A snapshot object is a concurrent data structure that consists of \( m \) components, each component storing a value from a given set. Processes can read/modify the state of the object by executing the operations \( \text{UPDATE} \) and \( \text{SCAN} \). An \( \text{UPDATE} \) operation gives processes the ability to change the value of a component of the snapshot, while the \( \text{SCAN} \) operation returns a “consistent” view of all the components of the object. In this thesis, two variants of concurrent snapshot objects are studied. The first one is the single-scanner snapshot object which can support only one active \( \text{SCAN} \) operation at any given time (whilst supporting many concurrent \( \text{UPDATE} \) operations). The second one is the multi-scanner snapshot object that can support multi concurrent \( \text{SCAN} \) operations at any given time. A variation of a multi-scanner snapshot object is a \( \lambda \)-scanner snapshot object, in which the latter can support only \( \lambda \) different \( \text{SCAN} \) operations to be active at any given time.

In this work, two wait-free, linearizable implementations of snapshot objects are presented. The first one is an implementation of a single-scanner snapshot object, and the second one is an implementation of a \( \lambda \)-scanner snapshot object. The operations that are supported by those implementations have a step complexity that doesn’t correlate with the number of active processes of the system \( n \), while the space and step complexity of the \( \lambda \)-scanner object is a function of \( \lambda \). Furthermore, our implementations have been designed to maintain a relatively low space complexity even if that means sacrificing some step complexity. The low space complexity that our implementations provide makes them more appealing in real system applications.

We first provide a simple implementation of a single-scanner snapshot object, called \( 1 - OPT \). This implementation uses \( O(m) \) \text{CAS} registers of unbounded size. The \( \text{UPDATE} \) has a step complexity of \( O(1) \), while the \( \text{SCAN} \) has a step complexity of \( O(m) \).

Then we present a \( \lambda \)-scanner snapshot object, called \( \lambda - OPT \) that is based on \( 1 - OPT \). When the number of \( \lambda \) is equal to the number of processes in the system, \( \lambda - OPT \) implements a multi-scanner object. This implementation uses \( O(\lambda m) \) \text{CAS} registers of unbounded size. The \( \text{UPDATE} \) implementation has a step complexity of \( O(\lambda) \), and the \( \text{SCAN} \) has a step complexity of \( O(\lambda m) \). This implementation is also wait-free and provides a trade-
off between the step/space complexity and the maximum number of \textit{SCAN} operations that can afford to be active on any given point in time.
Αποδοτικές Υλοποιήσεις Ατομικών Στιγμιοτύπων Κοινόχρηστης Μνήμης

Περίληψη

Την τελευταία δεκαετία έχει αυξηθεί αισθητά η χρήση των πολυπύρηνων συστημάτων, οι εφαρμογές των οποίων καλύπτουν ένα ευρύ φάσμα, όπως έξυπνα τηλέφωνα και υπολογιστές έως διακομιστές σε σύγχρονα πληροφοριακά κέντρα. Τα πολυπύρηνα συστήματα παρέχουν καλύτερες επιδόσεις μόνο υπό την προϋπόθεση ότι οι εφαρμογές τους μπορούν να χρησιμοποιήσουν τις δυνατότητες πολλαπλών υπολογιστικών πυρήνων. Ωστόσο αυτό δεν είναι τετριμμένο, μια που ο παράλληλος προγραμματισμός είναι αρκετά δυσκολότερος από τον σειριακό. Η χρήση αποδοτικών κοινόχρηστων δομών δεδομένων δίνει τη δυνατότητα πλήρους αξιοποίησης των πόρων ενός πολυπύρηνου συστήματος.

Μια διαδεδομένη κοινόχρηστη δομή δεδομένων είναι τα ατομικά στιγμιότυπα τα οποία και έχουν πληθώρα εφαρμογών στην επιστήμη των υπολογιστών. Τα ατομικά στιγμιότυπα μπορούν να χρησιμοποιηθούν για την επίλυση προβλημάτων, όπου συγκεκριμένες ενέργειες πρέπει να γίνουν όταν η κατάσταση του συστήματος ικανοποιεί κάποια συνθήκες. Επιπρόσθετα, η χρήση ατομικών στιγμιότυπων είναι ιδιαίτερα διαδεδομένη για την σχεδίαση αλλά και τον έλεγχο ιστοτήτας κοινόχρηστων δομών δεδομένων, όπως στην κατασκευή concurrent timestamps ή randomized consensus.

Ένα ατομικό στιγμίτυπο αποτελείται από συνιστώσες, οι οποίες αποθηκεύουν μια τιμή από ένα προκαθορισμένο σύνολο τιμών. Οι διεργασίες μπορούν να τροποποιούν ή/και να διαβάζουν τα αποθηκευμένα δεδομένα του αντικειμένου μέσα από την κλήση των διαδικασιών και. Η δίνει τη δυνατότητα στις διεργασίες να τροποποιήσουν τη τιμή μιας συνιστώσας θέτοντας σε αυτή μια τιμή από το προκαθορισμένο σύνολο τιμών. Από την άλλη, η επιστρέφει ένα συνεπές αντίγραφο των τιμών όλων των συνιστωσών του στιγμίτυπου. Στην παρούσα εργασία μελετάμε διάφορες κατηγορίες ατομικών στιγμιότυπων. Πρώτα μελετάμε τα single-scanner στιγμιότυπα, σε αυτές τις υλοποιήσεις μόνο μια ενεργή ανά πάσα χρονική στιγμή, ταυτόχρονα το σύστημα όμως υποστηρίζει τη συνύπαρξη πολλών αλλά και. Μια δεύτερη κατηγορία είναι τα multi-scanner στιγμιότυπα, όπου αυτά υποστηρίζουν ανά πάσα χρονική στιγμή και οσοδήποτε αλλά και. Όταν το είναι ίσο με το συνολικό τότε ένα multi-scanner ατομικο στιγμίτυπο είναι ταυτόσημο με ένα multi-scanner. Στην παρούσα εργασία, παρουσιάζονται δύο wait-free υλοποιήσεις ατομικών στιγμιότυπων. Η πρώτη αποτελεί υλοποίηση ενός single-scanner στιγμίτυπου ενώ η δεύτερη είναι υλοποίηση ενός λ-scanner στιγμίτυπου. Τόσο η χρονική τους πολυπλοκότητα όσο και η χωρική τους πολυπλοκότητα είναι γραμμική συνάρτηση του λ. Επιπρόσθετα, οι υλοποιήσεις μας έχουν σχετικά χαμηλή χωρική πολυπλοκότητα θυσιάζοντας μέρος της χρονικής πολυπλοκότητας. Η χαμηλή χωρική πολυπλοκότητα που παρέχουν οι υλοποιήσεις μας τις καθιστούν ελκυστικές σε εφαρμογές πραγματικών συστημάτων.
Αρχικά παρουσιάζεται μια απλή υλοποίηση ενός single-scanner στιγμότυπου, την οποία ονομάζουμε 1 – \textit{OPT}. Αυτή η υλοποίηση χρησιμοποιεί \(O(m)\) καταχωρητές. Η \textit{UPDATE} έχει χρονική πολυπλοκότητα \(O(1)\), ενώ η \textit{SCAN} \(O(m)\).

Τέλος παρουσιάζεται μια υλοποίηση ενός \(\lambda\)-scanner στιγμότυπου, την οποία ονομάζουμε \(\lambda – \textit{OPT}\). Αυτή η υλοποίηση χρησιμοποιεί \(O(\lambda m)\) καταχωρητές μη πεπερασμένου μεγέθους. Η χρονική πολυπλοκότητα της \textit{UPDATE} είναι \(O(\lambda)\) ενώ εκείνη της \textit{SCAN} είναι μόλις \(O(\lambda m)\).
Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επόπτη μου κ. Μανόλη Κατεβαΐνη, καθηγητή του τμήματος και εξαιρέτως ερευνητή του Ινστιτούτου Πληροφορικής, του οποίου η σταθερή καθοδήγησή αποτέλεσε κινητήρια δύναμη για την συγγραφή της παρούσας εργασίας.

Θέλω να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντά μου Νικόλαο Καλλιμάνη, ερευνητή του Ινστιτούτου Πληροφορικής. Η επιστημονική καθοδήγησή του, τόσο μέσω ιδεών όσο και μέσω σχολιών κατά την συγγραφή, συντέλεσε στην ολοκλήρωση αυτού του εγχειρήματος.

Επιπρόσθετα, ένα μεγάλο ευχαριστώ στην ερευνήτρια του Ινστιτούτου Πληροφορικής Λένα Κανέλλου, η οποία βοήθησε στην γραμματική αλλά και ποιοτική εκσφαλμάτωση του κειμένου αυτού.

Ως άλλο ευχαριστώ στην ερευνήτρια του Ινστιτούτου Πληροφορικής Λένα Κανέλλου, η οποία βοήθησε στην γραμματική αλλά και ποιοτική εκσφαλμάτωση του κειμένου αυτού.

Δεν θα μπορούσε να λείπει από τις ευχαριστίες αυτές η καθηγήτρια του τμήματος κα. Παναγιώτα Φατούρου, της οποίας οι συμβουλές αποδείχθηκαν ανεκτίμητες.

Ένα μεγάλο ευχαριστώ οφείλω στους φίλους μου, Αναστάσιο, Γεώργιο και Χριστίνα (και όλους τους υπόλοιπους που δεν αναφέρω συγκεκριμένα), για όλες τις συζητήσεις, για όλες τις φορές που διαβάζαμε μαζί, για όλες τις εκδηλώσεις φιλικής αγάπης, για την ασύγκριτη υποστήριξη, για όλες τις στιγμές.

Τέλος θέλω να πω ότι ένα τεράστιο ευχαριστώ στους γονείς μου, οι οποίοι όλα αυτά τα χρόνια αποτελέσαν τους αρωγούς σε κάθε μου προσπάθεια, με ένα ζεστό πιάτο φαγητό, μια αγκαλιά, με αστείρευτη αγάπη και υπομονή.
Στοις γονείς μου,
Σε σαένα,
Σε εμένα
Table of Contents
1. Introduction........................................................................................................................... 1
2. Related Work........................................................................................................................ 4
3. Model ................................................................................................................................... 7
4. 1-OPT ................................................................................................................................... 10
   4.1. Description of 1-OPT .................................................................................................... 10
   4.2. Linearization of 1-OPT .............................................................................................. 13
   4.3. Step Complexity of 1-OPT........................................................................................ 21
   4.4. Space Complexity of 1-OPT ....................................................................................... 21
   4.5. A partial version of 1-OPT ....................................................................................... 22
5. λ-OPT ................................................................................................................................... 25
   5.1. Description of λ-OPT ................................................................................................. 25
   5.2. Linearization of λ-OPT .............................................................................................. 28
   5.3. Step Complexity of λ-OPT........................................................................................ 41
   5.4. Space Complexity of λ-OPT ....................................................................................... 42
   5.5. A partial version of λ-OPT ....................................................................................... 43
6. Discussion and Future Work............................................................................................... 46
   6.1. Discussion .................................................................................................................. 46
   6.2. Future Work ............................................................................................................... 46
7. Bibliography........................................................................................................................... 47
List of Tables

Table 1: Known single-scanner snapshot implementations .................................................. 4
Table 2: Known multi-scanner snapshot implementations .................................................. 5
Table 3: Commonly Used Notation \((1 - \text{OPT})\) ................................................................. 13
Table 4: Commonly Used Notation \((\lambda - \text{OPT})\) ............................................................ 29
List of Figures

Figure 1: Two $SC$ operations of $SC_{val[i]}^\epsilon$ with their corresponding $LL$ operations ............ 16
Figure 2: $SC_{seq}^S$ table precedes $SC_1^S$ .......................................................................................... 17
Figure 3: $SC_{seq}^S$ table precedes $SC_1^S$ .......................................................................................... 31
Figure 4: The two cases that are used to prove Lemma 4. ................................................................. 31
Figure 5: The value of seq is greater or equal than $v_{seq}^S + 2$ right before the end of S. ........ 32
Figure 6: There is only one successful $SC_{seq}^v$ ............................................................................. 32
Figure 7: The existence of a successful $SC_{seq}^v$ implies the existence of a successful $SC_{seq}^{seq-1}$ .... 33
Figure 8: The existence of a successful $SC_{seq}^v$ implies the existence of any successful $SC_{seq}^{seq}$ 33
Figure 9: There is exactly one successful $SC$ on seq that sets the value of seq to $v' + 2$. .... 34
Figure 10: Two $SC$ operations of $SC_{val[i]}^\epsilon$ with their corresponding $LL$ operations .......... 36
Figure 11: At $C_t^{val[i]}$ the value of values[i].value would be $t$ ......................................................... 37
1. Introduction

In the last decades, we can observe the inarguably expansive applications of computer science. Almost every device is becoming smart. Thus, almost every device requires to have an embedded multicore CPU. All this new equipment promises to solve more and more problems while performing increasingly complex jobs. In our days, more than ever, an application that does not use the many cores that are provided by the hardware is gradually becoming obsolete.

A job that is performed by a single process in \( t \) seconds could ideally be performed in \( t/n \) seconds if \( n \) processes concurrently try to complete the same job. Although this is not quite true, since a complex job may have some parts that can only be executed sequentially. If we want to be more precise, a complex job may not be fully parallelizable. The portion of it that can be fully parallelized is denoted by \( p \), thus, \( 1 - p \) is the portion that has to be executed by a single core. The portion of the complex job that has to be executed by a single process dictates the maximum speedup that we can get. More specifically, Amdahl’s Law [1] implies that the best speedup that can be achieved when using infinitely many concurrent processes is \( \frac{1}{1-p} \).

At the core of concurrent algorithms are the concurrent data structures. Compared to sequential data structures, the concurrent ones can simultaneously be accessed and/or modified by more than one process. Ideally, we would like to have the best concurrent implementation, in terms of space and step complexity, of any given data structure. However, this is not yet true since the design of those data structures is a complex task. The design of concurrent data structures, such as lists [2], queues [3], [4], stacks [5], [6] and even trees [7], [8] is a vastly explored topic through many publications. Snapshot is also a concurrent data structure which is explored in this work. A concurrent snapshot object consists of components which can be read and modified by any process.

Concurrent snapshot objects are used in numerous applications. Snapshot objects are used to provide a coherent “view” of the memory of a system. They are also used to design and validate various concurrent algorithms such as the construction of concurrent timestamps [9], approximate agreement [10], etc. The ideas that are used to design a snapshot object may be further developed in order to implement complex data structures [11]. Applications of snapshot objects also appear in sensor networks where snapshot implementations can be used to provide a consistent view of the state of the various sensors of the network.

Snapshots can be used to simulate, under certain circumstances, concurrent graphs, such as the graph that is provided in [12]. The graph data structure is widely used by many applications, such as the representation of transport networks [13], video-game design [14], automated design of digital circuits [15]. Since snapshot objects can be used to simulate graph objects, the study of snapshot objects is of major interest in these areas.

There are many implementations of snapshot objects. We study different implementations based on the progress guarantee that they provide. With the term progress, we refer to the ability of an operation to successfully terminate its execution, independently from other operations or failures of other processes in the system. In order to provide fault tolerance against process failure, it is important to provide algorithms with strong progress guarantees. We can divide the different implementations of any concurrent
data structure in three main categories based on the progress guarantees they provide, as follows:

1. **Blocking Implementations**: such implementations do not provide any progress guarantee and they may be affected by a process failure. To put it simply, a process failure may lead operations of other processes to never terminate their execution. Since a blocking implementation provides no progress guarantee, it is easier to design. Because of that, such implementations often use small registers with primitives that are easily provided by the hardware.

2. **Lock-Free Implementations**: such implementations are less vulnerable to process failure. Furthermore, they provide a system-wide progress guarantee, meaning that given sufficient time, at least one operation would be successfully applied. The stronger progress guarantee usually results in implementations that use bigger registers and more complex primitives than their blocking counterpart. Since these implementations provide a better progress guarantee, it is preferable in systems where processes may crash.

3. **Wait-Free Implementations**: in such implementations, an operation invoked by any process that does not fail returns a result after it executes a finite number of steps. Those implementations provide the strongest progress guarantee, thus, they are more complex to design. The higher difficulty in providing stronger guarantees leads to implementations that use more and bigger registers.

In this thesis, we study the implementations of snapshot objects and provide two algorithms that implement such an object in a wait-free manner. We first design such an implementation for a single-scanner snapshot object. In such an object, only one process is allowed to read the values of it and any process may request changes to the values of components. We then provide an implementation of \( \lambda \) - scanner snapshot object, where \( \lambda \) predefined processes may read the components of the object while any process may change the value of any component. Note that \( \lambda \) should be lower than or equal to \( n \) (\( n \) is the processes that are available in the system). In case the value of \( \lambda \) is set to be the same as \( n \), we have an implementation of a general snapshot object.

We decided to present a wait-free implementation since it provides a better progress guarantee than other implementations. We also want to explore the lower bound of such algorithms by providing an implementation that uses less space than other state-of-the-art implementations and provides a relatively small step complexity. We also present a trade-off in our \( \lambda \) - scanner snapshot implementation, since the increase of the \( \lambda \) value leads to a linear increase of the space and step complexity.

Our implementation can be modified to obtain a dynamic partial snapshot implementation (partial snapshot implementations are presented in Sections 4.5. and 5.5.). In a dynamic partial snapshot implementation, processes can execute modified SCAN operations called PARTIAL_SCAN that could result in obtaining just a part of the snapshot object dynamically. We use the term dynamic to refer to the fact that the components, which a PARTIAL_SCAN reads, can be defined at execution time. Notice that these advantages are important if we use our implementation of the concurrent snapshot object to provide a simulation of a concurrent graph object, such as the graph object presented in [12].

Our algorithm, \( \lambda \) - OPT, has a low space complexity (number of shared registers used) of \( O(\lambda m) \) where \( m \) is the number of the components of the snapshot object. The low
space complexity does not come with major compromises in terms of step complexity, since the step complexity of an \UPDATE operation is $O(\lambda)$, while that of a \SCAN operation is $O(\lambda m)$. The registers we use are of unbounded size, although the only unbounded value that they store is a sequence number. This is a common practice from many state-of-the-art implementations [16], [17]. The atomic primitive that the registers need to support is that of \CAS(\text{Compare And Swap}), although we present a version of the algorithm using \LL/\SC registers in order to be more comprehensive and easier to prove correct. An \LL/\SC register can be constructed by \CAS registers using known constructions [18], [19].

The rest of the Thesis is organized as follows. Chapter 2, the Related Work section, provides a brief comparison of our algorithm, with other state-of-the-art algorithms that solve similar problems. Chapter 3, the Model section, contains all theoretical definitions used in this work. Chapter 4, the $1 - \OPT$ section, presents an algorithm called $1 - \OPT$, which is a wait-free implementation of a single-scanner snapshot object. Chapter 5, the $\lambda - \OPT$ section, contains our second algorithm called $\lambda - \OPT$. Chapter 6, the Discussion and Future Work section, provides some more ideas about our implementation and some ideas that may lead to further study in this field. Chapter 7, the bibliography section, contains all the citations that are used in this thesis.
2. Related Work

In the past years, many wait-free implementations of single-scanner snapshot objects have been presented. Some of them use simple multi-read/write registers, while others use single-write multi-read registers and just a few of them use CAS or LL/SC registers. Our implementation uses LL/SC registers. The use of LL/SC (or CAS) registers provides a single-scanner implementation that can be easily modified to obtain an algorithm that can solve the multi-scanner version of the snapshot problem, as presented in Section 5.

In terms of register size, Fatourou & Kallimanis [16] provided two implementations of a single-scanner snapshot object that use unbounded registers, since they use sequence number to achieve synchronization between operations. Using a recycling technique leads to two implementations that use bounded size registers of $O(\log n)$ size that are also presented in [16]. Riany, et al. [17] presented an algorithm that also uses registers of unbounded size that are needed to hold sequence numbers, a technique that we also use in $1 - OPT$. Jayanti [20] presented two algorithms that use registers of finite size. In our work, we use unbounded size registers. More specifically the registers used in our implementation need to be of size $O(\log(s))$, where $s$ is the maximum number of SCANS in a given execution. Given an execution that does not contain too many SCAN operations, our implementation uses relatively small registers.

Our implementation uses $O(m)$ registers, where $m$ is the number of components of the snapshot object. The only implementation that we are aware of that uses fewer registers is the Checkmarking algorithm presented in [16]. Checkmarking algorithm only uses $m + 1$ registers. On the other hand Riany et al. [17] provides an implementation that only uses $n + 1$ registers, where $n$ is the number of processes. In case the number of processes is high enough, our algorithm uses a smaller amount of registers compared to that presented in [17].

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Regs type</th>
<th>Regs number</th>
<th>Regs size</th>
<th>SCAN</th>
<th>UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 - OPT$</td>
<td>LL/SC and SW r/w</td>
<td>$O(m)$</td>
<td>Unbounded</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$1 - OPT$ (partial)</td>
<td>LL/SC and SW r/w</td>
<td>$O(m)$</td>
<td>Unbounded</td>
<td>$O(r)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Checkmarking [16]</td>
<td>MW r/w</td>
<td>$m + 1$</td>
<td>Unbounded</td>
<td>$O(m^2)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>$T - OPT$ [16]</td>
<td>MW r/w</td>
<td>Unbounded</td>
<td>Unbounded</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>RT [16]</td>
<td>MW r/w</td>
<td>$O(mn)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>RT − OPT [16]</td>
<td>MW r/w</td>
<td>$O(mn)$</td>
<td>$O(\log n)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Kirousis et al. [21]</td>
<td>MW r/w</td>
<td>$O(mn)$</td>
<td>$O(m \log n)$</td>
<td>$O(mn)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Riany et al. [17]</td>
<td>SW r/w</td>
<td>$n + 1$</td>
<td>Unbounded</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Jayanti [20]</td>
<td>SW r/w</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Jayanti [20]</td>
<td>LL/SC and r/w</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

The time complexity of SCAN operations in our implementation is $O(m)$. This time complexity is the same as an implementations presented by Jayanti [20] and by some implementations provided by Fatourou & Kallimanis [16]. Jayanti presents an implementation with $O(n)$ step complexity of SCAN operations. The same time complexity is provided by the implementation of Riany et al. [17] and the RT algorithm provided in [16]. We can’t have a straightforward comparison with the step complexity of those implementations since in case $n$ is greater than $m$, our implementation provides a better step complexity. On the other hand, in case $n$ is smaller than $m$, the step complexity of those implementations is better than the one we provide in this work. Kirousis et al. [21] provided an implementation where SCAN operations have a step complexity of $O(mn)$. 

4
The time complexity of \textit{UPDATE} operations in our implementation is $O(1)$, which is on par with all the different implementations that are reviewed in this section.

Table 1 summarizes known single-scanner snapshot implementations. In this table, we also include the partial version of $1 - \text{OPT}$ that is presented in 4.5. To the best of our knowledge, $1 - \text{OPT}$ is the only partial single-scanner implementation of a snapshot object. The partial version of $1 - \text{OPT}$ has the same characteristics as $1 - \text{OPT}$ except that the step complexity of a \textit{SCAN} operation is reduced to $O(r)$, where $r$ is the amount of components the \textit{SCAN} operation wants to read. Since $r \leq m$, the time complexity we provide for \textit{SCAN} operations of the partial version of $1 - \text{OPT}$ is lower than the non-partial version of $1 - \text{OPT}$.

Regarding multi-scanner snapshot implementations, to the best of our knowledge, $\lambda - \text{OPT}$ provides the first $\lambda - \text{scanner}$ snapshot implementation and thus, we compare $\lambda - \text{OPT}$ with other multi-scanner algorithms. In Table 2 we present the basic characteristics of each implementation that is reviewed in this section. In order to compare our algorithm with other implementations, we suppose that the value of $\lambda$ is relatively small, although not equal to one. In this case, a single-scanner snapshot implementation cannot be used instead of a $\lambda - \text{scanner}$ or a multi-scanner snapshot implementation to simulate a snapshot object.

Our implementation, $\lambda - \text{OPT}$ uses LL/SC registers, which is common in implementations of multi-scanner snapshot objects. Only the implementation of Attiya, Herlihy & Rachman make use of a different type of register and this is a dynamic Test&Set register. Our implementation only uses $O(\lambda m)$ registers, and in cases where $\lambda$ is a relatively small value, the number of registers used can be reduced to $O(m)$, which is on par with all the implementations that are reviewed in this section. Only Riany et al. have presented an implementation that uses $O(n^2)$ registers. This implementation uses more registers if $n^2$ is greater than $m$. The step complexity of \textit{SCAN} and \textit{UPDATE} operations of $\lambda - \text{OPT}$ is on par with the other state-of-the-art implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>P/Non-P</th>
<th>Regs Type</th>
<th>Regs Number</th>
<th>\textit{SCAN}</th>
<th>\textit{UPDATE}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda - \text{OPT}$</td>
<td>non-p</td>
<td>LL/SC and SW r/w</td>
<td>$O(\lambda m)$</td>
<td>$O(\lambda m)$</td>
<td>$O(\lambda)$</td>
</tr>
<tr>
<td>$\lambda - \text{OPT}$ (partial)</td>
<td>partial</td>
<td>LL/SC and SW r/w</td>
<td>$O(\lambda m)$</td>
<td>$O(\lambda r)$</td>
<td>$O(\lambda)$</td>
</tr>
<tr>
<td>Attiya, Herlihy &amp; Rachman [22]</td>
<td>non-partial</td>
<td>dynamic Test&amp;Set</td>
<td>$\infty$</td>
<td>$O(n\log^4 n)$</td>
<td>$O(n\log^4 n)$</td>
</tr>
<tr>
<td>Fatourou &amp; Kallimanis [23]</td>
<td>non-partial</td>
<td>CAS &amp; r/w regs</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Jayanti [20]</td>
<td>non-p</td>
<td>CAS or LL/SC &amp; r/w regs</td>
<td>$O(mn^2)$</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Jayanti [24]</td>
<td>non-p</td>
<td>CAS or LL/SC &amp; r/w regs</td>
<td>$O(mn^2)$</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Riany et al. [17]</td>
<td>non-p</td>
<td>CAS or LL/SC &amp; Fetch&amp;Inc &amp; r/w regs</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>\textit{Implied Algorithm by} [12]</td>
<td>partial</td>
<td>CAS or LL/SC &amp; r/w regs</td>
<td>$O(n + m)$</td>
<td>$O(k)$</td>
<td>$O(k)$</td>
</tr>
<tr>
<td>D. Imbs and M. Raynal [25]</td>
<td>partial</td>
<td>LL/SC and SW r/w</td>
<td>$O(n)$</td>
<td>$O(nr)$</td>
<td>$O(r_n n)$</td>
</tr>
<tr>
<td>Attiya Hagit, Guerraoui &amp; Rachid, Ruppert Eric [26]</td>
<td>partial</td>
<td>CAS and SW r/w</td>
<td>$O(n + m)$</td>
<td>$O(r^2)$</td>
<td>$O(r_n n_{\text{max}}^2)$</td>
</tr>
</tbody>
</table>

Note that the implementation presented by Fatourou & Kallimanis may have the best characteristics but uses very big registers. More specifically, this implementation requires registers that contain a vector of $m$ values as well as a sequence number. Our implementation
uses smaller registers. More specifically, we need registers that can store two integer values and a sequence number.

In case the value of $\lambda$ is equal to $n$, $\lambda - \text{OPT}$ provides an implementation of a multi-scanner snapshot object that uses a smaller amount of registers compared to the implementations of Attiya, Herlihy & Rachman, Jayanti and Riany et al.

Kallimanis and Kanellou [12] presented a wait-free implementation of a graph object. This implementation can be modified to simulate a snapshot object, which supports partial SCAN operations. We consider each component of the adjacency matrix presented in [12] to be a component of a snapshot object. This algorithm manages to implement UPDATE and SCAN operations with step complexity of $O(k)$, where $k$ is the number of active processes in a given execution. It also maintains a low space complexity of $O(n + m)$ but the registers used are of unbounded size. In essence, the algorithm needs registers that can contain $O(n)$ integer values, where half of those values are unbounded. This algorithm cannot be easily compared with partial $\lambda - \text{OPT}$, since we provide step and space complexities that are relative to the value $\lambda$ and $r$, where Kallimanis and Kanellou provided an implementation that has step complexities relative to $k$ and its space complexity is $O(n + m)$.

Imbs and Raynal [25] provide two implementations of a partial snapshot object. We compare our work with their second implementation that uses $LL/SC$ registers since the first implementation uses simpler registers, but has a higher space complexity. The space complexity of their second algorithm is $O(n)$. Our implementation has a lower space complexity when it is used for small snapshot objects and we only allow a small number of processes to invoke SCAN operations. The algorithm presented by Imbs and Raynal provides a step complexity to SCAN operations of $O(nr)$, which is higher than our implementation in any case, since $\lambda \leq n$. Moreover, they provide UPDATE operations that have $O(r'n)$ step complexity, where $r'$ is a value that is relative to the helping mechanism the UPDATE operations provide. Finally, the implementation of Imbs and Raynal provides a new idea about the helping mechanism since they implement the “write first, help later” technique in their work.

Attiya, Guerraoui and Ruppert [26] provided a partial snapshot algorithm that uses $O(m + n)$ CAS registers. The UPDATE operations of this implementation have a step complexity of $O(r^2)$. Notice that our implementation provides a better step complexity when $\lambda \leq r$. A step complexity of $O(\overline{c_s} r_{\text{max}}^2)$ is provided in [26], where $\overline{c_s}$ is the number of active SCAN operations, whose execution interval overlaps with the execution interval of $S$, and $r_{\text{max}}$ is the maximum number of components that any SCAN operation may read in any given execution.
3. Model

We consider a system consisting of \( n \) uniquely distinguishable processes modeled as sequential state machines. The processes are asynchronous and communicate through shared data structures called base objects. A base object has a state, and it provides a set of primitives, through which the object state can be accessed and/or modified.

- A Single – Write Multi – Read register \( R \) of a process \( p \) is a shared object that stores a value from a set and supports the primitives:
  - \( Write(R, v) \) that writes the value \( v \) in \( R \), and can only be invoked by process \( p \). This primitive returns a positive acknowledgment \( \text{ack} \).
  - \( Read(R) \) that returns the value of \( R \), and can be invoked by any process.
- A Multi – Write Multi – Read register \( R \) or more simple Multi Read/Write register \( R \), is a shared object that stores a value from a set and that supports the primitives:
  - \( Write(R, v) \) that writes the value \( v \) in \( R \), and can be invoked by any process.
    It returns a positive acknowledgment \( \text{ack} \).
  - \( Read(R) \) that returns the value of \( R \), and can be invoked by any process.
- An LL/SC register \( R \) is a shared object that stores a value from a set and supports the primitives:
  - \( LL(R) \) which returns the value of \( R \), and can be invoked by any process
  - \( SC(R, v) \) which can be executed by a process \( p \) only after the execution of an \( LL(R) \) primitive by the same process. An \( SC(R, v) \) writes the value \( v \) in \( R \) only if the state of \( R \) hasn’t changed since \( p \) executed the last \( LL(R) \). If \( SC(R, v) \) successfully updates the value of \( R \) with \( v \) then we say that this primitive is executed successfully and it returns a positive acknowledgment \( \text{ack} \); otherwise, it is executed unsuccessfully and returns a negative acknowledgment \( \text{nack} \).
- An LL/SC – Write register \( R \) is a shared object that stores a value from a set. It supports the same primitives as an \( LL/SC \) register \( R \) and in addition, the primitive \( Write(R, v) \) that writes the value \( v \) in \( R \), can be executed by any process and responds by returning a positive acknowledgment \( \text{ack} \).

A shared object is a data structure that can be accessed and/or modified by processes in the system. Each shared object provides a set of operations. Any process can access and/or modify the shared object by invoking operations that are supported by it.

An implementation of a shared object uses base objects to store the state of the shared object and provides a set of algorithms that use the base objects to implement each
operation of the shared object. An operation consists of an invocation by some process and terminates by returning a response to the process that invoked it.

Similar to each base object, each process also has an internal state. A configuration $C$ of the system is a vector that contains the state of each of the $n$ processes and the state of each of the base objects at some point in time. In essence, a configuration describes the state of the system at some point in time. In an initial configuration, the processes and base objects are in an initial state. We denote an initial configuration by $C_0$.

A step taken by a process consists either of a primitive to some base object or the response to that primitive. Operation invocation by some process and operation response to some process are also considered steps. Each step is executed atomically.

An execution $a$ is a (possibly infinite) sequence $C_0, e_1, C_1, e_2, C_2 ...$, alternating between configurations and steps, starting from some initial configuration $C_0$, where each $C_k, k > 0$, results from applying step $e_k$ to configuration $C_{k-1}$. If $C$ is a configuration that is present in $a$ we write $C \in a$. An execution interval of a given execution $a$ is a subsequence of $a$ which starts with some configuration $C_k$ and ends with some configuration $C_l$ (where $0 \leq k < l$). An execution interval of an operation $op$ is an execution interval with its first configuration being the one right after the step where $op$ was invoked and last being the one right after the step where $op$ responded.

Given an execution $a$, we say that a configuration $C_k$ precedes $C_l$ if $k < l$. Similarly, we say that step $e_k$ precedes step $e_l$ if $k < l$. We say that a configuration $C_k$ precedes the step $e_l$ in $a$, if $k < l$. On the other hand, we say that the step $e_l$ precedes $C_k$ in $a$ if $l \leq k$. We furthermore say that $op$ precedes $op'$ if the step where $op$ responds precedes the step where $op'$ is invoked. Given two execution intervals $I, I'$ of $a$, we say that $I$ precedes $I'$ if any configuration $C$ contained in $I$ precedes any configuration $C'$ contained in $I'$.

Given an execution $a$, we say that operation $op$ is active at a configuration $C$ if $C$ is contained inside the execution interval of $op$, otherwise, $op$ is inactive at $C$. An operation $op$ is called concurrent with an operation $op'$ in execution $a$ if there is at least one configuration $C \in a$, such that both $op$ and $op'$ are active in $C$. An execution $a$ is called sequential if in any given $C \in a$ there is at most one active $op$. An execution $a$ that is not sequential is called concurrent.

Executions $a$ and $a'$ are equivalent if they contain the same operations and only those operations are invoked in both of them by the same process, which in turn have the same responses in $a$ and $a'$.

An execution $a$ is linearizable if it is possible to assign a linearization point, inside the execution interval of each operation $op$ in $a$, so that the response of $op$ in $a$ is the same as its response would be in the equivalent sequential execution that would result from performing the operations in $a$ sequentially, following the order of their linearization points. An implementation of a shared object is linearizable if all executions it produces are linearizable.

A process $p$ may fail by crashing. In this case, there is a configuration $C \in a$, such that there is no step $e \in a$ that follows $C$ and is executed by $p$. To put it simply, $p$ takes no more steps after a certain point in time. Sometimes we may abuse notation and say that an
operation $op$ crashed, meaning that the process $p$ executing it failed by crashing while executing $op$.

Let $O$ be a shared object and $IM$ an implementation of this object. Let $a$ be any infinite execution produced by $IM$. We will say that $IM$ is \textit{wait-free} if any operation $op$, that does not crash in $a$, responds after a finite amount of steps. The maximum number of those steps is called \textit{step complexity} of $op$.

A \textit{snapshot} $S$ is a shared object that consists of $m$ components, each taking values from a set, that provides the following two primitives:

- $SCAN()$ which returns a vector of size $m$, containing the values of $m$ components of the object.
- $UPDATE(i, v)$ which writes the non \texttt{NULL} value $v$ on the $i$-th component of the object.

A \textit{partial snapshot} $S$ is a shared object that consists of $m$ distinct components denoted by $c_0, c_1, ..., c_{m-1}$, each taking values from a set, that provides the following two primitives:

- $SCAN(A)$ which, given a set $A$ that contains integer values ranging from 0 to $m - 1$, returns for each $i \in A$ the value of the component $c_i$.
- $UPDATE(i, v)$ which writes the non \texttt{NULL} value $v$ on $c_i$.

A snapshot implementation is \textit{single-scanner} if in any execution $a$ produced by the implementation there is no $C \in a$ in which there are more than one active $SCAN$ operations. Similarly, a snapshot implementation is \textit{partial-scanner} if in any execution $a$ produced by the implementation there is no $C \in a$ in which there are more than $\lambda$ active $SCAN$ operations.
4. 1-OPT

4.1. Description of 1-OPT

Our algorithm 1–OPT implements a single-scanner snapshot object. In this implementation, all processes can invoke UPDATE operations on any component of the snapshot object. 1–OPT uses a shared integer variable called seq in order to provide sequence numbers to operations. Each applied operation gets a sequence number by reading the shared variable seq. We often refer to that as the sequence number of the operation. The general idea behind the sequence number is that an operation op that is applied with a smaller sequence number than that of another operation op’ is considered to be applied before op’.

We use a shared table, which stores the values of the snapshot object. Each component of the snapshot object can store two values, one that is the current value of it and another one that is the value that an UPDATE operation wants to write on this component, we call this value as the announced value. The process that executes the SCAN operations has a unique data structure called pre_values where it stores a previous value and a sequence number of each component of the snapshot object. Since we apply a helping mechanism, any UPDATE and SCAN operation can read and modify the components of this data structure regardless of their process id.

---

Algorithm 1. Data Structures of 1-OPT

```c
1. struct value_struct {
2.  val    value;
3.  int seq;
4.  val proposed_value;
5. };

6. struct pre_value_struct {
7.  val value;
8.  int seq;
9. };

10. shared int seq;

11. shared value_struct values[0..m-1]=
    [NULL,NULL,NULL>,...,<NULL,NULL,NULL>];

12. shared pre_value_struct pre_values[0..m-1]=
    [NULL,NULL>,...,<NULL,NULL>];

13. private int view[0..m-1]=[NULL,NULL,...,NULL,NULL];
```

An UPDATE operation U on i–th component executed by process p first tries to announce the new value that it wants to store on the i–th component of the snapshot. This is achieved by trying to write on the announce value of the i–th component. Afterwards, it tries to copy the value of the i–th component of the snapshot to pre_values data structure if needed. Then it tries to update the value of the i–th component of the snapshot using a local copy of seq as its sequence number. If the announcement was successful, then the UPDATE operation ends its execution after the abovementioned last step. Otherwise, it repeats all previous steps for one last time. Doing so will make sure that an UPDATE
operation (may or may not be the same as $U$) on the $i -$th component of the snapshot object is applied and furthermore linearized inside the execution interval of $U$.

A $SCAN$ operation increases the value of $seq$ by one. The value of $seq$ right after the increase is the sequence number of this $SCAN$ operation. $UPDATE$ operations that have been applied with a greater or equal sequence number than that of the sequence number of this $SCAN$, are not “visible” from this operation (recall that operations are considered to be applied in increasing order based on their assigned sequence number). Afterwards, for each component of the snapshot object it does the following steps:

- It tries to copy the value of this component to pre_values data structure if the sequence number of the component is lower than that of the sequence number of the corresponding $SCAN$.
- It tries to apply an announced $UPDATE$ to this component of the snapshot object.

Finally, $SCAN$ returns its copy of the snapshot object.

Algorithm 2. $UPDATE$ and $SCAN$ implementations of 1-OPT

```c
14. void UPDATE(int j, int value){
15.   int i;
16.   struct value_struct up_value, cur_value;
17.   for (i=0; i<2; i++){
18.     cur_value=LL(values[j]);
19.     up_value=cur_value;
20.     up_value.proposed_value=value;
21.     if (cur_value.proposed_value==NULL){
22.       if (SC(values[j],up_value)){
23.         ApplyUpdate(j);
24.         break;
25.       }
26.     }
27.   }
28.   ApplyUpdate(j);
29. }
30. pointer SCAN(){
31.   int j;
32.   struct value_struct v1;
33.   struct pre_value_struct v2;
34.   seq=seq+1;
35.   for (j=0;j<m;j++){
36.     ApplyUpdate(j);
37.     v1=values[j];
38.     v2=pre_values[j];
39.     if (v1.seq<seq){
40.       view[j]=v1.value;
41.     }else{
42.       view[j]=v2.value;
43.     }
44.   }
45.   return view[0..m-1];
46. }
```
The data structures of the algorithm are shown in Algorithm 1. \(1 - OPT\) employs the shared Read – Write register \(seq\), which takes integer values and its initial value is 0. Each operation read \(seq\) in order to be assigned a sequence number. Only SCAN operations can increase the value of \(seq\) by one. Since in any given configuration there is only one active SCAN operation in our implementation, there is no need for this register to be more complex.

\(Single - OPT\) uses a shared table called \(values\) consisting of \(m\) structs. Each struct of \(values\) is stored in an \(LL/SC\) register and any process can execute \(LL\) and \(SC\) operations on each of them. The \(i-th\) component of the snapshot object is stored in the \(i-th\) struct of the \(values\) data structure, this struct is denoted \(values[i]\) and its type is \(valuestruct\). Each of those structs contains the following three values:

- A \(val\) variable called \(value\) which stores the value of the \(i-th\) component of the snapshot object that is simulated by \(1 - OPT\). The value of a component takes integer values.
- An int variable called \(seq\), which stores the sequence number of the last \(UPDATE\) operation that has been applied to the \(i-th\) component of the snapshot. This is also referred to as the sequence number of the \(i-th\) component.
- A \(val\) variable called \(proposed_value\) which stores the value that the announced \(UPDATE\) operation wants to apply on the \(i-th\) component of the snapshot.

**Algorithm 3. ApplyUpdate** implementation of 1-OPT

47. void ApplyUpdate(int j){
48. struct value_struct cur_value;
49. struct pre_value_struct cur_pre_value, proposed_pre_value
50. cur_value=LL(values[j]);
51. cur_seq=seq;
52. for (t=0; t<2; t++){
53.   cur_pre_value=LL(pre_values[j]);
54.   cur_value=values[j];
55.   if (cur_value.seq<seq){
56.     proposed_pre_value.seq=cur_value.seq;
57.     proposed_pre_value.value=cur_value.value;
58.     SC(pre_values[j],proposed_pre_value);
59.   }
60. }
61. if (cur_value.proposed_value!=NULL){
62.   cur_value.value=cur_value.proposed_value;
63.   cur_value.seq=cur_seq;
64.   cur_value.proposed_value=NULL;
65.   SC(values[j],cur_value);
66. }
67. }

The process that performs SCAN operations uses a shared table called \(pre_values\). This table consists of \(m\) structs that are stored in an \(LL/SC\) register and any process can execute \(LL\) and \(SC\) operations on them. The \(i-th\) struct of \(pre_values\) table is denoted by \(pre_values[i]\) and it contains a previous value of the \(i-th\) component. In other words, it contains the most recent value of the \(i-th\) component of the snapshot object that has a
sequence number smaller than that of the last invoked SCAN operation. The pre_values[i] is a struct of type pre_value_struct and it contains the following two variables:

- A val variable called value, which stores a copy of the value of the $i-th$ component of the snapshot.
- An int variable called seq, which stores the sequence number of the corresponding component. This sequence number is always smaller than that of the SCAN executed by process $p$.

4.2. Linearization of 1-OPT

We proceed with basic observations about SCAN operations in order to present their linearization points. In Table 3, we have the most commonly used notation in the process of assigning linearization points to $1-OPT$.

Table 3: Commonly Used Notation ($1-OPT$)

| $a$         | Any execution of $S-OPT$. |
| $U$         | An UPDATE operation inside $a$. |
| $S$         | A SCAN operation inside $a$. |
| $C$         | A configuration in $a$. |
| seq$_C$     | The value of seq at configuration $C$. |
| seq$_S$     | The value that $S$ stores in seq |
| seq$_U$     | The value of seq right after the invocation of $U$. |
| $w_v^{seq}$ | The write that stores the value $v$ to seq. |
| $SC_{i}^{val[i]}$ | The $i-th$ successful SC on values$[i]$. |
| $SC_{ann,U}^{val[i]}$ | The successful SC of type announce that $U$ executes. |
| $LL_{ann,U}^{val[i]}$ | The matching LL of $SC_{ann,U}^{val[i]}$. |
| $SC_{app,U}^{val[i]}$ | The successful SC of type apply that is executed after $SC_{ann,U}^{val[i]}$ and inside the execution interval of $U$. |
| $LL_{app,U}^{val[i]}$ | The matching LL of $LL_{ann,U}^{val[i]}$. |
| seq$_U$     | The value that $SC_{app,U}^{val[i]}$ stores in values$[i]$. seq. Essentially, seq$_U$ is the sequence number with which $U$ is applied. |

It is trivial to see that the value of seq is only modified by line 34, thus the following observation also stands true:

**Observation 1:** Only SCAN operations can modify the value of seq.

By inspection of the pseudocode of SCAN, only line 34 executes a write operation on the register that the value seq is stored (remind that only SCAN operations modify seq). Furthermore, a SCAN operation that responds in $a$, executes line 34, thus increasing the value of seq by one. Since our algorithm provides a solution to the single-scanner version of the snapshot problem, we do not support concurrent SCAN operations (there is no configuration in $a$ that more than one SCAN operations are active). Thus, the following observation stands true:

**Observation 2:** Given an execution $a$ of $1-OPT$, any SCAN operation that responds in $a$ increases exactly one time the value of seq. Furthermore, no SCAN operation decreases the value of seq.
In order to continue with our proof, we denote with $w_{seq}^{seq}$ the write that is executed by any \textit{SCAN} operation and stores the value $v$ to $seq$. We prove the following lemma that is used to linearize any \textit{SCAN} operation that responds in $a$.

**Lemma 1:** Let $S$ be a \textit{SCAN} operation that responds in $a$. Let also $seq_S$ be the value of $seq$ right after the execution of line 34 by $S$. Inside the execution interval of $S$, there is exactly one write operation that writes the value $seq_S$ to $seq$, we denote this write operation with $w_{seq_S}^{seq}$. Then, the value of $seq$ right after the invocation of $S$ is equal to $seq_S - 1$.

**Proof:** $S - OPT$ is a single-scanner implementation of a snapshot object, thus inside the execution interval of $S$ there is no other active \textit{SCAN} operation. Observation 1 implies that only \textit{SCAN} operations can modify the value of $seq$. Right after the execution of line 34 by $S$, the value of $seq$ is equal to $seq_S$ and Observation 2 implies that $S$ increases by one the value of $seq$ exactly one time inside its execution interval. So there would be exactly one step that writes the value $seq_S$ to $seq$ inside the execution interval of $S$. Furthermore, the value of $seq$ right before the execution of line 34 would be $seq_S - 1$. It follows, that right after the invocation of $S$ the value of $seq$ would be $seq_S - 1$, since the value of $seq$ only increases by one a single time inside the execution interval of $S$. Thus, Lemma 1 stands true. \hfill \blacksquare

We continue with the assignment of linearization point to any given \textit{SCAN} operation that responds in $a$. We linearize any such \textit{SCAN} at the configuration right after $w_{seq_S}^{seq}$.

**Lemma 2:** The linearization point of any \textit{SCAN} operation that responds in $a$ is inside its execution interval.

**Proof:** Let $S$ be any \textit{SCAN} operation that we have assigned a linearization point. $S$ is linearized right after $w_{seq_S}^{seq}$ and Lemma 1 implies that $w_{seq_S}^{seq}$ resides inside the execution interval of $S$. Thus, the linearization point of $S$ is inside its execution interval. It follows that Lemma 2 stands true. \hfill \blacksquare

In order to assign linearization points to any \textit{UPDATE} that responds in $a$. We need to first state some observations.

**Observation 3:** Any \textit{UPDATE} operation can execute at maximum one successful \textit{SC} of line 22.

**Observation 4:** Any register of the values array can only be modified by the execution of lines 22 or 65.

**Observation 5:** Let $j$ be any component of the snapshot object, then the value of $values[j]$ is modified only by lines 22 and 65.

We will now study the successful \textit{SC} operations on any register of the values array. Given any integer value $j$, such that $m > j \geq 0$, we denote with $SC_i^{val[j]}$ the $i$-th successful \textit{SC} on $values[j]$ register. Thus, we denote by $SC^{val[j]} = SC_0^{val[j]}, SC_1^{val[j]}, ..., SC_t^{val[j]}, ...$ the sequence of successful \textit{SC} on $values[j]$ register. There is a chance in some executions and for some value $j$, the sequence of successful \textit{SC} on $values[j]$ to be empty, so $SC_0^{val[j]} = \emptyset$.

**Lemma 3:** Given an execution $a$ and an integer value $j$ such that $m > j \geq 0$ and let $SC_0^{val[j]}, SC_1^{val[j]}, ..., SC_t^{val[j]}, ...$ be a non-empty sequence of \textit{SC} on $values[j]$. Then for any $SC_t^{val[j]}$ ($t$ is an integer value) that is term of the abovementioned sequence, it follows that if
Induction Step: If \( t \mod 2 = 0 \) then the \( S_{t}^{\text{val}}[j] \) is a successful \( SC \) of line 22 and is of type announce, otherwise \( S_{t}^{\text{val}}[j] \) is a successful \( SC \) of line 65 and is of type apply.

**Proof:** We will prove this statement using mathematical induction on the amount \( t \) of successful \( SC \) instructions in \( a \).

**Induction Base:** \( S_{0}^{\text{val}}[j] \) is a successful \( SC \) of line 22 and \( S_{1}^{\text{val}}[j] \) is a successful \( SC \) of type apply. (Note that this is true as long as \( S_{t}^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \). On the other hand \( S_{0}^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \) since the \( SC^{\text{val}}[j] \) is a non-empty sequence)

**Proof of Induction Base:** By inspection of the pseudocode line 11 it follows that at the initial configuration of \( a \), the value of \( \text{value}[j] \). \( \text{proposed} \_\text{value} \) is set to be \( \text{NULL} \). Observation 4 implies that the value of \( \text{value}[j] \) can only be modified by \( SC \) operations of line 22 and 65. By inspection of the pseudocode (lines 50, 61, 65), it follows that the \( SC \) of line 65 can only be executed in case that the if condition of line 61 is evaluated to true. If an \( SC \) of line 65 is the first \( SC \) of the \( SC^{\text{val}}[j] \) then its matching \( LL \) should read the preset values of \( \text{values}[j] \). In that case the if condition of line 61 would not be evaluated to true. Thus, \( S_{0}^{\text{val}}[j] \) cannot be an \( SC \) of type apply. On the other hand by inspection of lines 11, 18-22 it follows that \( S_{0}^{\text{val}}[j] \) is an \( SC \) of type announce. We will now prove that if \( S_{1}^{\text{val}}[j] \) is a term of the \( SC^{\text{val}}[j] \) then it should be an \( SC \) of type apply. Since \( S_{0}^{\text{val}}[j] \) is an \( SC \) of type announce and by inspection of the pseudocode (lines 18-22), it follows that \( S_{0}^{\text{val}}[j] \) sets the value of \( \text{values}[j] \). \( \text{proposed} \_\text{value} \) to be a non \( \text{NULL} \) value. Observation 5 implies that \( \text{values}[j] \) can only be modified by successful \( SC \) operations of line 22 or 65. It follows that between \( S_{0}^{\text{val}}[j], S_{1}^{\text{val}}[j] \) the value of \( \text{value}[j] \) remains the same. Let \( op \) be the operation that executes \( S_{1}^{\text{val}}[j] \). Since \( S_{1}^{\text{val}}[j] \) is a successful \( SC \) on \( \text{values}[j] \) it follows that its matching \( LL \) should follow the execution of \( S_{0}^{\text{val}}[j] \). Thus, the matching \( LL \) of \( S_{1}^{\text{val}}[j] \) should read the value that \( S_{0}^{\text{val}}[j] \) stored in \( \text{values}[j] \). It follows, (by inspection of the pseudocode, lines 18-22) that the if condition of line 21 is evaluated to false by \( op \) (if ever executed). Thus, \( op \) never executes line 22 and \( S_{1}^{\text{val}}[j] \) cannot be an \( SC \) of type announce. On the other hand, by inspection of the pseudocode (lines 50, 61, 65) it follows that the if condition of line 61 is evaluated to true by \( op \) and \( op \) successfully executes line 65. Thus, \( S_{1}^{\text{val}}[j] \) is an \( SC \) of type apply.

**Induction Hypothesis:** If \( S_{t}^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \) then \( S_{t}^{\text{val}}[j] \) is an \( SC \) of type announce if \( mod 2 = 0 \), else \( S_{t}^{\text{val}}[j] \) is an \( SC \) of type apply.

**Induction Step:** If \( S_{t}^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \) and \( S_{t+1}^{\text{val}}[j] \) is also a term of \( SC^{\text{val}}[j] \) then \( S_{t+1}^{\text{val}}[j] \) is an \( SC \) of type announce if \( (t + 1) \mod 2 = 0 \), else \( S_{t}^{\text{val}}[j] \) is an \( SC \) of type apply.

**Proof of Induction Step:** We proceed with case analysis.

- **Case 1:** Let \( t \mod 2 = 0 \), then \( (t + 1) \mod 2 = 1 \). Since \( t \mod 2 = 0 \), by our induction hypothesis, it follows that \( S_{t}^{\text{val}}[j] \) is an \( SC \) of type apply. By definition, between \( S_{t}^{\text{val}}[j] \) and \( S_{t+1}^{\text{val}}[j] \) there is no successful \( SC \) on \( \text{values}[j] \). By inspection of the pseudocode (lines 18-22), it follows that right after the execution of \( S_{t}^{\text{val}}[j] \), the value of \( \text{values}[j] \). \( \text{proposed} \_\text{value} \) is not equal to \( \text{NULL} \). Since \( S_{t+1}^{\text{val}}[j] \) is a
successful \( SC \), it follows that its matching \( LL \) operation follows \( SC_{t+1}^{val[i]} \), as shown in Figure 1. The matching \( LL \) of \( SC_{t+1}^{val[i]} \) reads the value that \( SC_t^{val[i]} \) stored in \( values[j] \). By inspection of the pseudocode (lines 18-22), it follows that the matching \( LL \) of \( SC_{t+1}^{val[i]} \) cannot be an \( LL \) of line 18, since in that case the if condition of line 21 would be evaluated to false, preventing the execution of \( SC \) of type announce. Thus, the matching \( LL \) of \( SC_{t+1}^{val[i]} \) is an \( LL \) of line 50. By inspection of the pseudocode (lines 50 and 61-65), it follows that the if condition of line 61 is evaluated to true and thus the \( SC \) of line 65 is executed successfully. It follows that \( SC_{t+1}^{val[i]} \) is an \( SC \) of type apply.

- **Case 2:** Let \( t \mod 2 = 1 \), then \((t + 1) \mod 2 = 0 \). Since \( t \mod 2 = 1 \), by our induction hypothesis, it follows that \( SC_t^{val[i]} \) is an \( SC \) of type apply. Between \( SC_t^{val[i]} \) and \( SC_{t+1}^{val[i]} \) there is no successful \( SC \) on \( values[j] \), so the value of \( values[j] \) does not change in this execution interval. By inspection of the pseudocode (lines 62-65), it follows that right after the execution of \( SC_t^{val[i]} \), the value of \( values[j] \), \( proposed\_value \) is NULL. Using the same argument as in case 1 it follows that \( SC_{t+1}^{val[i]} \) is an \( SC \) of type announce.

\[
\begin{align*}
\text{LL}_t^{val[i]} & \quad SC_t^{val[i]} & \quad \text{LL}_{t+1}^{val[i]} & \quad SC_{t+1}^{val[i]}
\end{align*}
\]

*Figure 1: Two \( SC \) operations of \( SC_{t+1}^{val[i]} \) with their corresponding \( LL \) operations.*

In order to assign linearization points to \( UPDATE \) operations, we study the \( UPDATE \) operations that successfully execute an \( SC \) of line 22 by proposing the following definition and prove lemma 4.

Let \( U \) be any \( UPDATE \) operation in \( a \). We denote by \( C^{st}_U \) the configuration right after the invocation of \( U \). The value of \( seq \) at \( C^{st}_U \) is denoted by \( seq^{st}_U \).

**Lemma 4:** Let \( U \) be an \( UPDATE \) operation that wants to store the value \( t \) to the \( i - th \) component such that: (a) \( U \) successfully executes an \( SC \) of line 22, denoted by \( SC_{\text{ann},U}^{val[i]} \) (b) inside the execution interval of \( U \) and after \( SC_{\text{ann},U}^{val[i]} \) at least one \( SC \) of line 65 we denote the first such \( SC \) by \( SC_{\text{app},U}^{val[i]} \). \( SC_{\text{app},U}^{val[i]} \) is executed by an operation \( op \) that may, or may not be the same as \( U \). At the configuration right after the execution of \( SC_{\text{app},U}^{val[i]} \), denoted with \( C_{\text{app},U}^{val[i]} \), the value of \( values[i] \). \( value \) is \( t \) and the value of \( values[i] \). \( seq \) is greater or equal to \( seq^{st}_U \).

**Proof:** By inspection of the pseudocode (lines 18-22), it follows that at \( C_{\text{ann},U}^{val[i]} \) the value of \( values[i] \), \( proposed\_value \) will be \( t \). Observation 5 implies that the struct \( values[i] \) can only be modified by the execution of lines 22 and 65. Lemma 3 implies that an \( SC \) of line 22 cannot be successfully executed between \( SC_{\text{ann},U}^{val[i]} \) and \( SC_{\text{app},U}^{val[i]} \). Furthermore, \( SC_{\text{app},U}^{val[i]} \) is the first \( SC \) of line 65 that follows \( SC_{\text{ann},U}^{val[i]} \). Thus, between \( SC_{\text{ann},U}^{val[i]} \) and \( SC_{\text{app},U}^{val[i]} \), there is no successful \( SC \) on \( values[i] \). Let \( LL_{\text{app},U}^{val[i]} \) be the matching \( LL \) operation of \( SC_{\text{app},U}^{val[i]} \) executed by \( op \). Given
that $SC_{app, U}^{val[i]}$ is executed successfully $LL_{app, U}^{val[i]}$ should follows the execution of $SC_{ann, U}^{val[i]}$. It follows that $LL_{app, U}^{val[i]}$ reads the value that $SC_{ann, U}^{val[i]}$ stores in $values[i]$ (as shown in Figure 2). Right after $LL_{app, U}^{val[i]}$, op reads the value of seq (line 51), we denote this value with $seq_U$. The value $seq_U$ is greater or equal than $seq_{st}$, this is true since inside a the value of seq never decreases (Observations 1 and 2). By inspection of the pseudocode (lines 50, 51 and 61-65) it follows that op evaluates the if condition of line 61 to true and with the execution of $SC_{app, U}^{val[i]}$ sets the value of $values[i].value$ to $t$ and the value of $values[i].seq$ to the value of seq that op reads with the execution of line 51. So lemma 4 stands true.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{At $C^i_{val[i]}$ the value of $values[i].value$ would be $t$.}
\end{figure}

Lemma 4 introduces the notation $seq_U$ that refers to the value that $SC_{app, U}^{val[i]}$ stores to $values[i].seq$. The processes that executed the $SC_{app, U}^{val[i]}$ read the value of seq with the execution of line 51 and then stores this value to $values[i].seq$ without modifying it. Thus, right before the abovementioned read of line 51, the value of seq is equal to $seq_U$. Furthermore, Observation 1 implies that only SCAN operations can modify the value of seq. Thus the following observation stands true.

**Observation 6:** Let a be any execution of $\lambda - OPT$. Let also U be an UPDATE operation that wants to store the value $t$ to the $i-th$ component such that: (a) U successfully executes an SC of line 22, denoted by $SC_{ann, U}^{val[i]}$ (b) inside the execution interval of U and after $SC_{ann, U}^{val[i]}$ at least one SC of line 65 is successfully executed. Then, inside a there is a SCAN operation that stores the value $seq_U$ to seq before the response of the execution of U.

We now assign a linearization point to any UPDATE operation U that successfully executes an SC of line 22 ($SC_{ann, U}^{val[i]}$) and inside its execution interval but after $SC_{ann, U}^{val[i]}$, there is a successful execution of an SC of line 65. We distinguish the following two cases:

- **Case 1:** Right after the execution of $SC_{app, U}^{val[i]}$, the value of seq is equal to $seq_U$. U is linearized right after the execution of $SC_{app, U}^{val[i]}$.
- **Case 2:** Right after the execution of $SC_{app, U}^{val[i]}$, the value of seq is greater than $seq_U$. U is linearized at $w_{seq_U+1}^{seq}$. Recall that $w_{seq_U+1}^{seq}$ is the write operation of line 34 that sets the value of seq to $seq_U + 1$. Since in case 2 the value of seq right after the execution of $SC_{app, U}^{val[i]}$ is greater or equal than $seq_U$, it should be greater or equal to $seq_U + 1$ (Observation 2 implies that the value of seq is increased only by one each time). Thus, before the execution of $SC_{app, U}^{val[i]}$ a SCAN operation set the value of seq to $seq_U + 1$.

Finally, we assign linearization points to any UPDATE operation U that respond in a but do not execute a successful SC of line 22.
• **Case 3**: Inside the execution interval of $U$ an *UPDATE* operation, which is assigned a linearization point using case 1 or case 2, is linearized. Let $U'$ be the first such operation. We assign the linearization point of $U$ right before the linearization point of $U'$ (We show in lemma 8 that there is at least one *UPDATE* operation that is linearized inside the execution interval of $U$. We prove in lemma 9 that the linearization point of $U$ resides inside the execution interval of $U$).

If an *UPDATE* operation is linearized at the same configuration as another *SCAN* operation, then the *UPDATE* operation is linearized before the *SCAN* operation.

We now show that the linearization point that is assigned in *UPDATE* operations that belong to case 1 or case 2 resides inside their execution interval.

**Lemma 5**: Let also $U$ be an *UPDATE* operation that wants to store the value $t$ to the $i$–th component such that: (a) $U$ successfully executes an *SC* of line 22, denoted by $SC_{val[i]}^{val[i]}$, inside the execution interval of $U$ and after $SC_{val[i]}^{val[i]}$, at least one *SC* of line 65 is successfully executed by some operation $op$ ($op$ may, or may not be the same as $U$). The linearization point of $U$ resides inside its execution interval.

**Proof**: If $U$ is linearized right after the execution of $SC_{app,U}^{val[i]}$, then its linearization point belongs to the execution interval of $U$. If $U$ is linearized at $w_{seq_{U}+1}$ then, we will show that $w_{seq_{U}+1}$ is inside the execution interval of $U$. The value $seq_{U}$ is the value of $seq$ that $op$ reads after the execution of line 51, we denote this operation with $r_{U}^{seq}$. $r_{U}^{seq}$ follows the execution of $LL_{app,U}^{val[i]}$ and precedes the execution of $SC_{app,U}^{val[i]}$. Both $LL_{app,U}^{val[i]}$ and $SC_{app,U}^{val[i]}$ reside inside the execution interval of $U$. Thus, the execution of $r_{U}^{seq}$ is inside the execution interval of $U$. Although, right after $SC_{app,U}^{val[i]}$, the value of $seq$ is greater or equal than $seq_{U}$ ($U$ is linearized following the case 2). Since $seq_{U} < seq_{U} + 1$ and $seq_{U} + 1$ is smaller or equal than the value of $seq$ right after $SC_{app,U}^{val[i]}$, it follows, that somewhere between the $r_{U}^{seq}$ and $SC_{app,U}^{val[i]}$ there was a *SCAN* operation that set the number of $seq$ to $seq_{U} + 1$. Thus, $w_{seq_{U}+1}$ is inside the execution interval of $U$. It follows that lemma 5 stands true in any given case.

The proof of lemma 5 implies that any *UPDATE* operation $U$ that is linearized based on the abovementioned assignment of linearization points has its linearization point inside its execution interval and somewhere in between $LL_{app,U}^{val[i]}$ and the configuration right after $SC_{app,U}^{val[i]}$.

**Observation 7**: Let also $U$ be an *UPDATE* operation that wants to store the value $t$ to the $i$–th component such that: (a) $U$ successfully executes an *SC* of line 22, denoted by $SC_{app,U}^{val[i]}$ (b) inside the execution interval of $U$ and after $SC_{val[i]}^{val[i]}$, at least one *SC* of line 65 is successfully executed by some operation $op$ ($op$ may, or may not be the same as $U$). $U$ has its linearization point somewhere between $LL_{app,U}^{val[i]}$ and the configuration right after $SC_{app,U}^{val[i]}$.

We now continue through proving the following lemma.

**Lemma 6**: Let $U$ be an *UPDATE* operation that wants to store the value $t$ to the $i$–th component such that: (a) $U$ successfully executes an *SC* of line 22, denoted by $SC_{app,U}^{val[i]}$ (b) outside the execution interval of $U$, after $SC_{val[i]}^{val[i]}$ at least one *SC* of line 65 is successfully
executed by some operation \( op \) (\( op \) may, or may not be the same as \( U \)). Then, inside the execution interval of \( U \) and after \( SC^{\text{val}}_{\text{ann}, U} \), at least one \( SC \) of line 65 is successfully executed by some operation \( op' \) (\( op' \) may, or may not be the same as \( U \)).

**Proof:** Let’s assume by contradiction that there is an \( UPDATE \) operation \( U \) such that statements (a) and (b) stand true, but there is no operation \( op' \) that successfully executes an \( SC \) of line 65 inside the execution interval of \( U \) and after \( SC^{\text{val}}_{\text{ann}, U} \). Right after \( SC^{\text{val}}_{\text{ann}, U} \) the value of \( \text{values}[i].\text{proposed\_value} \) is equal to \( t \). Inside the execution interval of \( U \) and after \( SC^{\text{val}}_{\text{ann}, U} \) there is no successful execution of an \( SC \) of line 65. Although there is an \( SC \) of line 65 successfully executed after \( SC^{\text{val}}_{\text{ann}, U} \), we denote the first such \( SC \) operation as \( SC_1 \).

Between \( SC^{\text{val}}_{\text{ann}, U} \) and \( SC_1 \) there is no successful \( SC \) on \( \text{values}[i] \). By inspection of the pseudocode (lines 22, 23) it follows that \( U \) invokes an \( \text{ApplyUpdate} \) function after \( SC^{\text{val}}_{\text{ann}, U} \). By inspection of the pseudocode (lines 50 and 61-66), it follows that the if condition of line 61 will be evaluated to true by \( U \) (\( LL \) of line 50 reads the value that \( SC^{\text{val}}_{\text{ann}, U} \) stored in \( \text{values}[i] \)). Thus, an \( SC \) of line 65 is executed by \( U \) inside its execution interval. The abovementioned \( SC \) operation is successful, which is a contradiction. Thus, lemma 6 stands true in any case. 

We now study \( UPDATE \) operations that are linearized using case 3.

**Lemma 7:** Let \( U \) be an \( UPDATE \) operation that wants to update the \( i-th \) component of the snapshot object with the value \( t \) such that: (a) \( U \) responds in a but, (b) \( U \) does not execute a successful \( SC \) of line 22. Then, inside the execution interval of \( U \), there is some operation \( U' \) that executes \( LL^{\text{val}}_{\text{app}, U} \), and some operation \( op \) that executes \( SC^{\text{val}}_{\text{ann}, U} \).

**Proof:** Let’s assume by contradiction that there is no \( UPDATE \) operation \( U' \) that executes \( LL^{\text{val}}_{\text{app}, U} \), and an operation \( op \) that executes \( SC^{\text{val}}_{\text{ann}, U} \) inside the execution interval of \( U \). Let’s study the first iteration of the loop of lines 17-28. Since \( U \) does not execute successfully an \( SC \) of line 22 it follows that either the if condition of line 21 is evaluated to false or the execution was evaluated to true but the execution of line 22 was unsuccessful.

- Let’s study the case where the execution of line 21 is evaluated to false. By inspection of the pseudocode (lines 18 and 21), it follows that right after the execution of line 18 by \( U \), the value of \( \text{values}[j].\text{proposed\_value} \) is not \( \text{NULL} \). Since only \( SC \) operations of type announce write a value that is not \( \text{NULL} \) in \( \text{values}[j].\text{proposed\_value} \), it follows that, the last successful \( SC \) on \( \text{values}[j] \) was an \( SC \) of type announce. Let’s denote this \( SC \) operation as \( SC^{U}_{\text{ann}, 1} \), note that this \( SC \) operation may or may not be inside the execution interval of \( U \). \( U \) executes line 23 and afterwards lines 50 and 61-66 are executed by \( U \). Following the same reasoning, it follows that after the execution of the \( LL \) of line 18 and before the execution of line 66 an \( SC \) of type apply is successfully executed. We denote this \( SC \) by \( SC^{U}_{\text{app}, 1} \).
- The case where the execution of line 21 is evaluated to true is similar.

Following the same reasoning for the second loop of lines 17-28, it follows that, after \( SC^{U}_{\text{app}, 1} \) and inside the execution interval of \( U \) there is an \( SC \) of type announce denoted by \( SC^{U}_{\text{ann}, 2} \). Furthermore, there is a successful \( SC \) operation of type apply, denoted by \( SC^{U}_{\text{app}, 2} \), that is inside the execution interval of \( U \) and follows the execution of \( SC^{U}_{\text{ann}, 2} \). Since \( SC^{U}_{\text{ann}, 2} \) is an \( SC \) of type announce executed inside the execution interval of \( U \) it follows that \( SC^{U}_{\text{ann}, 2} \) is
executed by some process $U'$. So $U'$ executed $SC_{val[i]}^{ann,U'}$ inside the execution interval of $U$. Lemma 6 implies that, since the execution of $SC_{val[i]}^{ann,U'}$ is followed by a successful $SC$ execution of type apply ($SC_{app,2}^U$) it follows that $U'$ is an $UPDATE$ operation that is described in lemma 4. The first successful $SC$ of type apply that follows $SC_{val[i]}^{ann,U'}$ is the $SC_{val[i]}^{ann,U'}$. Thus, $SC_{app,2}^U$ may or may not be the same as $SC_{app,U'}$ but if $SC_{app,U'}$ is the same with $SC_{app,2}^U$ then $SC_{val[i]}^{ann,U'}$ precedes $SC_{app,U'}^U$. The corresponding $LL$ operation of $SC_{app,U'}^U$ precedes $SC_{val[i]}^{ann,U'}$ and follows $SC_{val[i]}^{ann,U'}$. Thus, $LL_{app,U'}$ and $SC_{val[i]}^{ann,U'}$ are inside the execution interval of $U$, which is a contradiction. Lemma 6 stands true in any case. ■

We now prove that inside the execution interval of an $UPDATE$ operation that is linearized using the case 3 another $UPDATE$ operation is linearized.

**Lemma 8**: Let $U$ be an $UPDATE$ operation that wants to update the $i - th$ component of the snapshot object with the value $t$ such that: (a) $U$ responds in $a$ but, (b) $U$ does not execute a successful $SC$ of line 22. Then, inside the execution interval of $U$ another $UPDATE$ operation, denoted $U''$, is linearized.

**Proof**: Lemma 7 implies that inside the execution interval of $U$ there is some operation $U'$ that executes $LL_{val[i]}^{ann,U'}$, and some operation op that executes $SC_{val[i]}^{val[i]}$. $U'$ is linearized somewhere in between $LL_{val[i]}^{val[i]}$ and the configuration right after $SC_{val[i]}^{val[i]}$ (Observation 7). Thus Lemma 8 stands true. ■

We now prove that the linearization point of an $UPDATE$ of case 3 is inside its execution interval.

**Lemma 9**: Let $U$ be an $UPDATE$ operation that wants to update the $i - th$ component of the snapshot object with the value $t$ such that: (a) $U$ responds in $a$ but, (b) $U$ does not execute a successful $SC$ of line 22. Let $U'$ be the first $UPDATE$ operation that is linearized inside the execution interval of $U$. We assign the linearization point of $U$ right before the linearization point of $U'$. The linearization point of $U$ resides inside the execution interval of $U$.

**Proof**: Lemma 8 implies that there is at least one $UPDATE$ operation that is linearized inside the execution interval of $U$. We denote the first such $UPDATE$ operation with $U'$. Since $U$ is linearized right before $U'$, it follows that the linearization point of $U$ resides in its execution interval. ■

In order to prove that we assigned a linearization point on every operation that responds in $a$ we need to prove the following lemma.

**Lemma 10**: Let $U$ be any $UPDATE$ operation that wants to change the value of the $i - th$ component, of the snapshot object, and responds in $a$, then a linearization point is assigned to $U$.

**Proof**: $U$ either executed a successful $SC$ of type announce or it did not. Let’s first study the case where $U$ didn’t execute a successful $SC$ of type announce. Since $U$ responds in $a$, $U$ belongs to a case 3 $UPDATE$ and is linearized as such. Let’s now study the case where $U$ successfully executed an $SC$ of type announce. If, after $SC_{val[i]}^{val[i]}$, there is a successful $SC$ of type apply, then Lemma 6 implies that $U$ is an $UPDATE$ operation of case 1 or case 2. If after $SC_{val[i]}^{val[i]}$, there is no successful $SC$ of type apply, we will prove that at least $U$ executes an $SC$
of type apply and that should be successful which would be a contradiction. \( SC_{\text{val}[i]} \) is the last successful \( SC \) operation that changes the value of \( \text{values}[i] \). By inspection of the pseudocode (lines 22, 23) it follows that \( U \) invokes an \( ApplyUpdate \) function after \( SC_{\text{val}[i]} \). By inspection of the pseudocode (lines 50 and 61-66), it follows that the if condition of line 61 will be evaluated to true by \( U \) (LL of line 50 reads the value that \( SC_{\text{val}[i]} \) stored in \( \text{values}[i] \)). So, an \( SC \) of line 65 is executed by \( U \) inside its execution interval. The abovementioned \( SC \) operation is successful, which is a contradiction. Thus, Lemma 10 always stands true.

4.3. Step Complexity of 1-OPT

The step complexity of any operation of \( 1 - OPT \) is measured by the number of accesses that are executed in shared registers, inside its execution interval.

We start with the worst-case analysis of \( ApplyUpdate \).

1. In lines 48-51 only an \( LL \) operation is performed at line 50 and a read of shared variable \( \text{seq} \) (line 51).
2. Lines 52-60 contain a loop that is executed at maximum two times. In each iteration of this loop, there are executed at maximum two \( LL/SC \) operations (the \( LL \) of line 53 and the \( SC \) of line 58) and one read of line 54.
3. Lines 61-66 contain just a single \( SC \) operation (line 65).

Thus, \( ApplyUpdate \) executes \( O(1) \) shared memory accesses.

We now proceed with the worst-case analysis of the step complexity of any \( UPDATE \). The loop of lines 17-28 can be executed two times at maximum and contains an \( LL \) (line 18), an \( SC \) (line 22) and two invocations of \( ApplyUpdate \) (lines 23 and 27). We previously proved that any \( ApplyUpdate \) executes \( O(1) \) shared memory accesses. It follows that any \( UPDATE \) operation executes \( O(1) \) shared memory accesses.

We can finally proceed with the worst-case analysis of the step complexity of any \( SCAN \).

1. A write operation on the shared value \( \text{seq} \) is executed on line 34.
2. Lines 35-44 contain a loop that is executed exactly \( m \) times. In each iteration of the loop an invocation of \( ApplyUpdate \) is executed (line 36) and two read operations (lines 37 and 38) are performed.

It follows that any \( SCAN \) operation executes \( O(m) \) shared memory accesses.

4.4. Space Complexity of 1-OPT

The space complexity of \( 1 - OPT \) algorithm is measured through counting the number of shared registers that are needed for its implementation. The implementation of \( 1 - OPT \) deploys three different shared objects:

1. A shared integer variable called \( \text{seq} \) which is stored in a multi-read/write register.
2. A shared table called \( \text{values} \) that is consisted of \( m \) \( LL/SC \) registers of unbounded size.
3. A shared table called \( \text{pre.values} \) that is consisted of \( m \) \( LL/SC \) registers of unbounded size.

Thus, our implementation deploys \( 2m LL/SC \) unbounded registers and \( 1 r/w \) register of unbounded size. It follows that the space complexity of our algorithm is \( O(m) \).
The implementation of $1-OPT$ presented in this work uses $LL/SC$ registers of unbounded size (one sequence number and two integer values). Although registers should be unbounded it can be proven that they need to have a size of $O(\log(s))$, where $s$ is the maximum number of $SCANS$ in a given execution. Thus, in executions that the maximum number of $SCAN$ operation is not too big, $1-OPT$ may use bounded registers.

**Theorem 1:** $1-OPT$ is a wait-free linearizable concurrent single-scanner snapshot implementation that uses $O(m)$ registers, provides $O(1)$ step complexity to any $UPDATE$ operation and $O(m)$ to any $SCAN$ operation.

### 4.5. A partial version of $1$-$OPT$

We now present a modified version of $1-OPT$ that implements a partial snapshot object. The data structures used in this modified version of $1-OPT$ remain the same and as shown in Algorithm 4. Furthermore, the pseudocode of the $UPDATE$ operation and the $ApplyUpdate$ function remain the same as shown in Algorithms 5 and 7. A new function is introduced called $Read$ (Algorithm 6). This function is invoked by $SCAN$ operations in order to read the values of the snapshot object.

A $SCAN$ operation firstly increases the value of $seq$ shared variable by one and then executes a for loop. For each integer $j$ that is contained in $A$ ($A$ is the set that contains the different components of the graph a $SCAN$ operation wants to read), the $SCAN$ operation tries to help an $UPDATE$ operation that wants to update the value of $c_j$ component by invoking the $ApplyUpdate$ function. Afterwards, it reads the value of $c_j$ by invoking the $Read$ function.

---

**Algorithm 4. Data Structures of 1-OPT (partial version)**

1. struct value_struct {
   2.   val    value;
   3.   int   seq;
   4.   tval  proposed_value;
   5. };

6. struct pre_value_struct {
   7.   val    value;
   8.   int   seq;
   9.   }

10. shared int seq;

11. shared value_struct values[0..m-1]=
    [<NULL,NULL,NULL>,...,<NULL,NULL,NULL>];

12. shared pre_value_struct pre_values[0..m-1]=
    [<NULL,NULL>,...,<NULL,NULL>];

13. private int view[0..m-1]=[NULL,NULL,...,NULL,NULL];

The only modification in this version of $1-OPT$ is that the $SCAN$ operations do not read every component of the snapshot object, they only read the components of set $A$. The sketch of proof of the partial version of $1-OPT$ is outside of the scope of this work.
Both partial $1 - \text{OPT}$ and non-partial $1 - \text{OPT}$ provide the same step complexity to \textit{UPDATE} operations of $O(1)$ and have the same space complexity of $O(m)$. Although, partial $1 - \text{OPT}$ provides a step complexity to \textit{SCAN} operations of $O(r)$ where $r$ is the number of elements contained in $A$. Simpler said $r$ is the number of different components that the \textit{SCAN} operation reads. The step complexity that non-partial $1 - \text{OPT}$ provides to \textit{SCAN} operation is $O(m)$. Since $r \leq m$ the step complexity of partial $1 - \text{OPT}$ is lower compared to the step complexity of non-partial version of $1 - \text{OPT}$.

\begin{algorithm}
\caption{\textit{UPDATE} and \textit{SCAN} implementations of 1-OPT (partial)}

\begin{algorithmic}
\State 14. \textbf{void UPDATE}(int $j$, int value){
\State 15. \hspace{1em} int $i$;
\State 16. \hspace{1em} struct value_struct up_value, cur_value;
\State 17. \hspace{1em} for ($i=0$; $i<2$; $i++$){
\State 18. \hspace{2em} cur_value=LL(values[$j$]);
\State 19. \hspace{2em} up_value=cur_value;
\State 20. \hspace{2em} up_value.proposed_value=value;
\State 21. \hspace{2em} if (cur_value.proposed_value==NULL){
\State 22. \hspace{3em} if (SC(values[$j$],up_value)){
\State 23. \hspace{4em} ApplyUpdate($j$);
\State 24. \hspace{3em} break;
\State 25. \hspace{2em} }
\State 26. \hspace{1em} }\hspace{1em} ApplyUpdate($j$);
\State 27. \hspace{1em} }
\State 28. \hspace{1em} }
\State 29. \hspace{1em}

\State 30. \textbf{void SCAN}(A){
\State 31. \hspace{1em} seq=seq+1;
\State 32. \hspace{1em} for each $j$ in A{
\State 33. \hspace{2em} ApplyUpdate($j$);
\State 34. \hspace{2em} Read($j$);
\State 35. \hspace{1em} }
\State 36. \hspace{1em}
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{\textit{Read} implementation of 1-OPT (partial)}

\begin{algorithmic}
\State 37. \textbf{int Read}($j$){
\State 38. \hspace{1em} struct value_struct v1;
\State 39. \hspace{1em} struct pre_value_struct v2;
\State 40. \hspace{1em} v1=values[$j$];
\State 41. \hspace{1em} v2=pre_values[$j$];
\State 42. \hspace{1em} if ($v1$.seq<$seq$){
\State 43. \hspace{2em} view[$j$]=v1.value;
\State 44. \hspace{1em} }else{
\State 45. \hspace{2em} view[$j$]=v2.value;
\State 46. \hspace{1em} }
\State 47. \hspace{1em} Return view[$j$];
\State 48. }
\end{algorithmic}
\end{algorithm}
Algorithm 7. \textit{ApplyUpdate} implementation of 1-OPT (partial)

49. void ApplyUpdate(int j){
50.     struct value_struct cur_value;
51.     struct pre_value_struct cur_pre_value,proposed_pre_value
52.     cur_value=LL(values[j]);
53.     cur_seq=seq;
54.     for (t=0; t<2; t++){
55.         cur_pre_value=LL(pre_values[j]);
56.         cur_value=values[j];
57.         if (cur_value.seq<seq){
58.             proposed_pre_value.seq=cur_value.seq;
59.             proposed_pre_value.value=cur_value.value;
60.             SC(pre_values[j],proposed_pre_value);
61.         }
62.     }
63.     if (cur_value.proposed_value!=NULL){
64.         cur_value.value=cur_value.proposed_value;
65.         cur_value.seq=cur_seq;
66.         cur_value.proposed_value=NULL;
67.         SC(values[j],cur_value);
68.     }
69. }

24
5. λ-OPT

5.1. Description of λ-OPT

Our algorithm $\lambda - OPT$ implements a $\lambda - SCANNER$ snapshot object. All processes can invoke $UPDATE$ operations on any component of the snapshot object. We only allow a predefined amount of $\lambda$ processes to invoke $SCAN$ operations. $\lambda - OPT$ uses a shared integer variable called $seq$. Each applied operation gets a sequence number by reading the shared variable $seq$. We often refer to that as the sequence number of the operation. The general idea behind the sequence number is that an operation $op$ with a smaller sequence number than that of another operation $op'$ is considered to be applied before $op'$.

We use a shared table data structure that stores the values of the snapshot object. Each component of the snapshot object can store two values, one that is the current value of it and another one that is the announced value, simpler said this is the value that an $UPDATE$ operation wants to write on this component. A $SCAN$ operation executed by process $p$ has a unique data structure called $prevalues_p$ where it stores a previous value and a sequence of each component of the snapshot object. Since we apply a helping mechanism any $UPDATE$ and $SCAN$ operation can read and modify the components of this data structure regardless of their process id.

An $UPDATE$ operation $U$ on $i-th$ component executed by process $p$ first tries to announce the new value that it wants to store on the $i-th$ component of the snapshot. This is achieved by trying to write on the announce value of the $i-th$ component of the snapshot object. Afterwards, it tries to copy the value of the $i-th$ component of the snapshot to every $prevalues_p$ data structure if needed. Then it tries to $UPDATE$ the value of the $i-th$ component of the snapshot using a local copy of $seq$ as its sequence number. If the announcement was successful, then the $UPDATE$ operation ends its execution after the abovementioned last step. Otherwise, it repeats all previous steps for one last time. Doing so will make sure that an $UPDATE$ operation (may or may not be the same as $U$) on the $i-th$ component of the snapshot object is applied and furthermore linearized inside the execution interval of $U$.

A $SCAN$ operation uses the helping technique to acquire a sequence number, which in turn defines the $UPDATE$ operations that are visible to this $SCAN$. $UPDATE$ operations that have been applied with a greater or equal sequence number than that of the sequence number of this $SCAN$, are not visible from this operation. Furthermore, each $SCAN$ operation tries to increase the value of $seq$. Afterwards, for each component of the snapshot object it does the following steps:

- Tries to copy the value of this component to every $prevalues_p$ data structure that is used by $SCAN$ operations, if the sequence number of the component is lower than that of the sequence number of the corresponding $SCAN$.
- Tries to apply an announced $UPDATE$ to this component of the snapshot object.

It returns its copy of the snapshot object.

We now describe the data structures that are used in $\lambda - OPT$. The data structures of the algorithm are shown in Algorithm 8.
Our algorithm, $\lambda - OPT$, employs a shared $LL/SC$ register $seq$, which takes integer values. Each operation accesses $seq$ in order to be assigned a sequence number. Only $SCAN$ operations can increase the value of $seq$ by one, and they try to do so 3 times while helping themselves and other $SCAN$ operations with the assignment of their sequence numbers.

Algorithm 8. Data Structures of $\lambda$-OPT

1. struct valuestruct {
2.    weightval value;
3.    int seq;
4.    weightval proposed_value;
5. }
6. struct pre_valuestruct {
7.    weightval value;
8.    int seq;
9. }
10. struct scanstruct{
11.    int seq;
12.    boolean write_enable;
13. }
14. shared int seq;
15. shared valuestruct values[0..m-1]=
16.     [<NULL,NULL,NULL>,...,<NULL,NULL,NULL>];
17. shared pre_valuestruct pre_values[0..l-1][0..m-1]=
18.     [<NULL,NULL>,...,<NULL,NULL>];
19. shared scanstruct s_table[0..l-1]=
20.     [<NULL,0>,<NULL,0>,...,<NULL,0>];
21. private int view[0..m-1]=[NULL,NULL,...,NULL,NULL];

$\lambda - OPT$ uses a shared table called $values$ consisting of $m$ structs. Each struct of $values$ is stored in an $LL/SC$ register and any process can execute $LL$ and $SC$ operations on each of them. The $i - th$ component of the snapshot object is stored in the $i - th$ struct of the $values$ data structure, this struct is denoted $values[i]$ and its type is $valuestruct$. Each of those structs contains the following three values:

- A $weightval$ variable called $value$ which stores the weight of the $i - th$ component of the snapshot object that is simulated by $\lambda - OPT$. The weight of a component takes integer values.
- An int variable called $seq$, which stores the sequence number of the last $UPDATE$ operation that has been applied to the $i - th$ component of the snapshot. Also referred to as the sequence number of the $i - th$ component.
- A $weightval$ variable called $proposed_value$ which stores the value that the announced $UPDATE$ operation wants to apply on the $i - th$ component of the snapshot.
Algorithm 9. *UPDATE* and *SCAN* implementations of λ-OPT

```c
19. void UPDATE(int j, int value){
20.  struct valuestruct up_value, cur_value;
21.  for (i=0; i<2; i++){
22.    cur_value=LL(values[j]);
23.    up_value=cur_value;
24.    up_value.proposed_value=value;
25.    if (cur_value.proposed_value==NULL){
26.      if (SC(values[j],up_value)){
27.        ApplyUpdate(j);
28.        break;
29.      }
30.    }
31.  }
32.  ApplyUpdate(j);
33.}

34. pointer SCAN(){
35.  s_table[p_id]={1,seq};
36.  for (i=0; i<3; i++){
37.    cur_seq=LL(seq);
38.    for (j=0; j<λ; j++){
39.      cur_s_table=LL(s_table[j]);
40.      if(cur_s_table.seq<seq+2 && cur_s_table.write_enable==1){
41.        cur_s_table.write_enable=0;
42.        cur_s_table.seq=seq+2;
43.        SC(s_table[j],cur_s_table);
44.      }
45.    }
46.    SC(seq,cur_seq+1);
47.  }  
48.  for (j=0; j<m; j++){
49.    ApplyUpdate(j);
50.    v1=values[j];
51.    v2=pre_values[p_id][j];
52.    if (v1.seq<s_table[p_id].seq){
53.      view[j]=v1.value;
54.    } else{
55.      view[j]=v2.value;
56.    }
57.  }
58.  return view[0..m-1];
59.}
```

Any process $p$ that is eligible to invoke *SCAN* operations uses a shared table called _values_[p]. vals. This table consists of $m$ structs that are stored in an LL/SC register and any process can execute LL and SC operations on them. The $i-th$ struct of pre_values_[p]. vals table is denoted by pre_values_[p]. vals[i] and it contains the value of the $i-th$ component, of the snapshot object that is visible by the last *SCAN* operation invoked by $p$. In other words, it contains the most recent value of the $i-th$ component of the snapshot object that has a sequence number smaller than that of the last invoked *SCAN* operation by $p$. The
is a struct of type `pre_valuestruct` and it contains the following two variables:

- A `weightval` variable called `value`, which stores a copy of the weight of the `i-th` component of the snapshot.
- An `int` variable called `seq`, which stores the sequence number of the corresponding component. This sequence number is always smaller than that of the `SCAN` executed by process `p`.

Furthermore, to assign a sequence number to any `SCAN` operation, `λ-OPT` uses a shared table of `λ` components called `s_table`. Each component of the table is a struct of type `scanstruct` and is stored in an `LL/SC write` register, and any process can execute all the primitives of this register. The `p-th` component of the `s_table` is denoted by `s_table[p]` and contains the sequence number of the last `SCAN` operation invoked by `p`. More specifically, `s_table[p]` contains the following two values:

- An `int` variable called `seq` which contains the sequence number of the `SCAN` operation invoked by process `p`.
- A `bool` variable called `write_enable` which is `TRUE` when the sequence number of the corresponding `SCAN` is not yet been assigned otherwise, it is `FALSE`.

Algorithm 10. **ApplyUpdate** implementation of `λ-OPT`

```c
61. void ApplyUpdate(int j){
62.   struct value_struct cur_value;
63.   struct pre_value_struct cur_pre_value,proposed_pre_value;
64.   cur_value=LL(values[j]);
65.   cur_seq=seq;
66.   for (i=0; i<λ; i++){
67.     for (t=0; t<2; t++){
68.       cur_pre_value=LL(pre_values[i][j]);
69.       cur_value=values[j];
70.       if (cur_value.seq<s_table[j].seq){
71.         proposed_pre_value.seq=cur_value.seq;
72.         proposed_pre_value.value=cur_value.value;
73.         SC(pre_values[i][j],proposed_pre_value);
74.       }
75.     }
76.   }
77.   if (cur_value.proposed_value!=NULL){
78.     cur_value.value=cur_value.proposed_value;
79.     cur_value.seq=cur_seq;
80.     cur_value.proposed_value=NULL;
81.     SC(values[j],cur_value);
82.   }
83. }
```

5.2. Linearization of `λ-OPT`

In Table 4, we have some commonly used notation in the process of assigning linearization points to operations of `λ-OPT`.
Let $a$ be any execution of $\lambda - OPT$. In order to assign linearization points to any SCAN operation that responds inside $a$, we need to prove some useful observations and lemmas. Let’s first consider Observation 1.

The value of shared variable $seq$ is only modified by a successful $SC$ of line 46. Lines 37 and 46 imply that the $SC$ of line 46 can only increase the value of $seq$ by one. So we have the following observation:

**Observation 1:** Let $C, C'$ be configurations in $a$, and let also $C'$ follows $C$. The value of $seq$ in $C'$ is greater or equal than that of $seq$ in $C$.

Let us continue with the following lemma:

**Lemma 1:** Let $S$ be a $SCAN$ operation that responds inside $a$, executed by process $p$. Inside the execution interval of $S$ there is at least one successful $SC$ on $s_table[p]$ (line 43) that follows the execution of line 35 by $S$, this $SC$ may or may not executed by $S$.

**Proof:** Let’s assume by contradiction that there is no successfully executed $SC$ on $s_table[p]$ (line 43) by any process inside the execution interval of $S$. On the first iteration of the for loop of lines 44-55 and the $p-th$ iteration of the nested loop of lines 36-47, $S$ executes the $LL$ operation on $s_table[p]$ of line 39. Between the write of line 35 and the $LL$ of line 39 there is no successful $SC$ on $s_table[p]$ (by our hypothesis). Since only $S$ can execute a write operation on $s_table[p]$, it is trivial that the $LL$ of line 39 on $s_table[p]$ reads the value that $S$ stored in it with the write of line 35. Let $< 1, x >$ be the value that $S$ stored on $s_table[p]$ after the execution of line 35, where $x$ is the value of $seq$ that $S$ reads at line 35. We now prove that on the first loop of lines 36-47 and the $p-th$ loop of lines 38-45, $S$ evaluates the if condition of line 40 to true. Let $C$ be the configuration right before the step that evaluates the abovementioned if condition, we denote by $seq_C$ the value of $seq$ at $C$. Observation 1 implies that at $C$ it holds that $x \leq seq_C < seq_C + 2$, since $x$ is the value of $seq$ in a previous configuration than $C$. Furthermore, $cur_s_table.write_enable == 1$ at $C$, since $S$ stored the value 1 after the execution of its line 35. So, $S$ executes successfully the $SC$ on $s_table[p]$ of line 43, since there is no successful $SC$ on $s_table[p]$ by the time $S$ executed the $LL$ of line 39 on $s_table[p]$ (hypothesis). This is a contradiction. It follows that Lemma 1 holds in any case.

We next prove in **Lemma 2** that there is exactly one successful $SC$ instruction on $s_table[p]$ inside the execution interval of $S$. 

---

**Table 4: Commonly Used Notation ($\lambda - OPT$)**

<table>
<thead>
<tr>
<th>$a$</th>
<th>Any execution of $\lambda = OPT$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>A $SCAN$ operation inside $a$.</td>
</tr>
<tr>
<td>$C$</td>
<td>A configuration in $a$.</td>
</tr>
<tr>
<td>$seq_C$</td>
<td>The value of $seq$ at configuration $C$.</td>
</tr>
<tr>
<td>$SC_{s_table}^S$</td>
<td>The unique successful $SC$ on $s_table[p]$ in the execution interval of $S$.</td>
</tr>
<tr>
<td>$LL_{s_table}^S$</td>
<td>The matching $LL$ of $SC_{s_table}^S$.</td>
</tr>
<tr>
<td>$LL_{s_table}^{L_x}$, $LL_{s_table}^{L_y}$, $LL_{s_table}^{L_z}$</td>
<td>The three $LL$ operations on $seq$ executed by $S$.</td>
</tr>
<tr>
<td>$SC_x^{S}, SC_y^{S}, SC_z^{S}$</td>
<td>The three $SC$ operations on $seq$ executed by $S$.</td>
</tr>
<tr>
<td>$v^s_{seq}$</td>
<td>The value of $seq$ that $S$ reads with the execution of $LL_{s_table}^{L_x}$.</td>
</tr>
<tr>
<td>$SC_{v_{seq}}$</td>
<td>The $SC$ that stores the value $v$ to $seq$.</td>
</tr>
<tr>
<td>$LL_{v_{seq}}$</td>
<td>The matching $LL$ of $SC_{v_{seq}}$.</td>
</tr>
</tbody>
</table>
**Lemma 2:** Let $S$ be a SCAN operation that responds inside $a$, executed by process $p$. Inside the execution interval of $S$, there is exactly one successful $SC$ on $s\text{\_table}[p]$ denoted by $SC_{S,\text{table}}^{s}$ (line 43) that follows the execution of line 35 by $S$. $SC_{S,\text{table}}^{s}$ is executed by some process $q$, where $q$ may or may not be the same as $p$.

**Proof:** Lemma 1 states that there is at least one such $SC$ operation, so we need to prove that this operation is unique inside the execution interval of $S$. We will prove that after the first successful $SC$ on $s\text{\_table}[p]$ there is no other successful $SC$ on $s\text{\_table}[p]$ inside the execution interval of $S$. Let $SC_{1}$ be the first successful $SC$ of Lemma 1 and let $C_{1}$ be the configuration right after $SC_{1}$. We now distinguish the following two cases. There is no other successful $SC$ operation on $s\text{\_table}[p]$ inside the execution interval of $S$ that follows $SC_{1}$. In this case, the lemma holds trivially. In the second case, there is at least one successful $SC$ operation on $s\text{\_table}[p]$ inside the execution interval of $S$ that follows $SC_{1}$. Let $SC_{2}$ be the first such operation. We will now prove that this case is never true. $SC_{1}$ is executed on line 43 and by inspection of the pseudocode (lines 41 and 43) follows that this $SC$ sets the value of $scan\text{\_table}[p]$.write_enable to 0. Let $S_{2}$ be the SCAN operation that executes $SC_{2}$ we denote by $LL_{2}$ the matching $LL$ instruction of $SC_{2}$. In case $S_{2}$ executed $LL_{2}$ of line 39 after $SC_{1}$ then it reads $s\text{\_table}[p]$.write_enable = 0, thus the following if condition of line 40 is evaluated to false which is a contradiction. So $S_{2}$ has to execute $LL_{2}$ before $SC_{1}$ thus leading to an unsuccessful $SC$ operation on $s\text{\_table}[p]$. Since, between the $LL_{2}$ and the $SC_{2}$, an $SC$ operation ($SC_{1}$) is executed successfully on $s\text{\_table}[p]$. It follows that $S_{2}$ cannot execute a successful $SC$ on $s\text{\_table}[p]$ which is a contradiction so, the second case cannot stand true. Thus Lemma 2 holds in any given case. ■

We will now study the $LL$ and $SC$ operations that are executed on variable $seq$ by a SCAN.

By inspection of the pseudocode of SCAN operation (line 46), it follows that any SCAN operation that responds inside $a$ executes the $SC$ on $seq$ of line 46 exactly 3 times.

**Observation 2:** Let $S$ be a SCAN operation that responds inside $a$, then inside the execution interval of $S$ there would be 3 executions of line 46 denoted by $SC_{S,1}^{s}, SC_{S,2}^{s}, SC_{S,3}^{s}$ and their corresponding $LL$ operations executed at line 37 denoted by $LL_{S,1}^{s}, LL_{S,2}^{s}, LL_{S,3}^{s}$.

We further have to assign linearization points and prove that the assignment of linearization points is inside the execution interval so we continue with the proof of the following lemma.

**Lemma 3:** Let $S$ be a SCAN operation that responds inside $a$ executed by a process $p$, then $SC_{S,\text{table}}^{s}$ precedes $SC_{S}^{1}$.

**Proof:** Lemma 2 states that $SC_{S,\text{table}}^{s}$ follows the execution of line 35 by $S$. Assume by contradiction that $SC_{S,\text{table}}^{s}$ follow the execution of $SC_{S}^{1}$, we will prove that this cannot be true. Lemma 2 implies that inside the execution interval of $S$, $SC_{S,\text{table}}^{s}$ is the only successful $SC$ operation on $scan\text{\_table}[p]$. Since only $S$ can execute a write operation on $s\text{\_table}[p]$, it is trivial that the $LL$ of line 39 executed by $S$ on $s\text{\_table}[p]$ reads the value that $S$ stored in it with the write of line 35. Let $< 1, x >$ be the value that $S$ stored on $s\text{\_table}[p]$ after the execution of line 35, where $x$ is the value of $seq$ that $S$ reads at line 35. We now prove that on the first loop of lines 36-47 and the $p-th$ loop of lines 38-45, $S$ evaluates the if condition of line 40 to true. Let $C$ be the configuration right before the step that evaluates the if condition of line 40. We denote by $seq_{C}$ the value of $seq$ at $C$. Observation 1 implies that at $C$ it holds that $x \leq seq_{C} < seq_{C} + 2$, since $x$ is the value of $seq$ in a previous configuration than $C$. 

30
Furthermore, \texttt{cur\_s\_table.write\_enable} \(==\) 1 at \(C\), since \(S\) stored the value 1 after the execution of its line 35. So, \(S\) executes successfully the \texttt{SC} on \texttt{s\_table}[p] of line 43, since there is no successful \texttt{SC} on \texttt{s\_table}[p] between the execution of line 35 by \(S\) and the \texttt{SC} \(r_1\) (line 46) by our hypothesis. This is a contradiction since in that case, there would be a successful \texttt{SC} operation on \texttt{s\_table}[p] between the execution of line 35 by \(S\) and \texttt{SC} \(r_3\) (line 46). Thus, lemma 3 stands in any given case. \(\blacksquare\)

![Figure 3: SC\text{\_table} \(r_3\) precedes SC\text{\_table} \(r_1\).

We will now study the sequence of values that the variable \texttt{seq} takes through the following observation.

\textbf{Case 1:} \(LL\text{\_table}^2\) precedes \(LL\text{\_table}^3\)

![Figure 4: The two cases that are used to prove Lemma 4.

\textbf{Lemma 4:} Let \(S\) be a \texttt{SCAN} operation that responds inside \(a\). Let \(v_s\text{\_seq}^S\) be the value of shared variable \texttt{seq} that \(S\) reads with \(LL\text{\_table}^2\). The value of \texttt{seq} would be greater or equal to \(v_s\text{\_seq}^S + 2\) just before the end of the execution interval of \(S\).

\textbf{Proof:} Let \(C_3^S, C_4^S\) be the configurations right after the execution of \texttt{SC} \(r_2\), \texttt{SC} \(r_3\) respectively. If the execution of \texttt{SC} \(r_2\) is successful then at \(C_3^S\), pseudocode implies that the value of \texttt{seq} would be \(v_s\text{\_seq}^S + 1\) (lines 37 and 46). If the execution of \texttt{SC} \(r_2\) is not successful there is another successful \texttt{SC} operation between \(LL\text{\_table}^2, SC\text{\_table}^2\). Let \texttt{SC} \(r_3\) be the first successful \texttt{SC} on \texttt{seq} (line 46) between \(LL\text{\_table}^2, SC\text{\_table}^2\), and let also be \(LL\text{\_table}^2\) its corresponding \texttt{LL} operation. If \(LL\text{\_table}^2\) follows \(LL\text{\_table}^2\) (Case 1 of Figure 3) it returns the value \(v_s\text{\_seq}^S\) since between \(LL\text{\_table}^2\) and \(SC\text{\_table}^2\) there is no successful \texttt{SC} on \texttt{seq}. Thus, the value of \texttt{seq} didn’t change since the execution of \(LL\text{\_table}^2\). In this case, the successful \texttt{SC} \(r_3\) will store the value of \(v_s\text{\_seq}^S + 1\) to \texttt{seq}. Otherwise (Case 2 of Figure 3) if \(LL\text{\_table}^2\) precedes \(LL\text{\_table}^2\) since the execution of \texttt{SC} \(r_2\) is successful there is no successful \texttt{SC} on \texttt{seq} between \(LL\text{\_table}^2\) and \(SC\text{\_table}^2\), so the value of \texttt{seq} does not change between \(LL\text{\_table}^2\) and \(SC\text{\_table}^2\). Thus, the value that \(LL\text{\_table}^2\) returns is the same with the value that \(LL\text{\_table}^2\) returns. So, \(LL\text{\_table}^2\) returns the
Lemma 5: We now also prove the following lemma about the values of the shared variable seq.

Let seq be any successful SC on seq (line 46) that writes some value v on register seq in a. There is no other successful SC operation on seq that writes the value v in a.

Proof: We assume by contradiction that there are at least two successful SC operations of line 46 inside the execution interval of a that set the same value v to seq. Let SC_v^{seq} be the first successful SC operation of line 46 that sets the value of seq to v and let also be LL_v^{seq} its corresponding LL operation of line 37. Let also be SC_v^{seq} be the second successful SC operation of line 46 that sets the value of seq to v and let LL_v^{seq} be its corresponding LL operation of line 37. By inspection of the pseudocode (lines 37, 46) it follows that both LL_v^{seq} and LL_v^{seq} should return the value v − 1 in order for their corresponding SC to store the value v. Right after SC_v^{seq} the value of seq is v and observation 1 implies that this value would not decrease in any later configuration. Since SC_v^{seq} is successful it follows that LL_v^{seq} should follow the execution of SC_v^{seq} (Figure 5), in that case, it would return a value greater or equal to v, which is a contradiction. Thus lemma 5 stands true.

Figure 5: The value of seq is greater or equal than v_v^{seq} + 2 right before the end of S.

We will continue studying the values of variable seq through the following lemma.

Lemma 6: Let v be the value of seq right after any successful SC, denoted by SC_v^{seq}, on seq of line 46, such that v > 1. Then inside a there is exactly one successful SC execution of line 46 executed before SC_v^{seq} that sets the value of seq to v − 1.

Proof: By inspection of the pseudocode, lines 37,46 are the only lines that perform LL and SC operations on seq. The SC of line 46 can only increase the value of seq by one, if it is executed successfully. Let LL_v^{seq} be the matching LL operation of SC_v^{seq}. By inspection of the pseudocode lines 37,46 it follows that LL_v^{seq} returns the value v − 1, otherwise, the value that
Corollary 1: By using a straightforward induction on Lemma 6 we have the following corollary. For every integer value $v$,

\begin{align*}
\text{Lemma 7:} & \quad \text{value of } \text{seq} = v - 1. \text{ Since, } LL_v^\text{seq} \text{ precedes } SC_v^\text{seq} \text{ it follows that } C \text{ precedes } SC_v^\text{seq}. \text{ Since, } v > 1 \text{ it follows that } v - 1 \text{ is greater than 0, thus, } v - 1 \text{ is not the initial value of } \text{seq}. \text{ Recall that the initial value of } \text{seq} \text{ is 0}. \text{ Since, only a successful } SC \text{ of line 46 can change the value of } \text{seq}. \text{ It follows that inside } \alpha \text{ and before } LL_v^\text{seq} \text{ there is a successful } SC \text{ that stored the value } v - 1 \text{ to } \text{seq} (\text{Figure 6}). \text{ Furthermore, Lemma 4 implies that this } SC \text{ is the only } SC \text{ in } \alpha \text{ that stores the value } v - 1 \text{ in } \text{seq}. \text{ Thus, Lemma 6 holds.} 
\end{align*}

![Figure 7: The existence of a successful $SC_v^\text{seq}$ implies the existence of a successful $SC_{v-1}^\text{seq}$](image)

![Figure 8: The existence of a successful $SC_v^\text{seq}$ implies the existence of any successful $SC_i^\text{seq} (v > i > 0)$.](image)

By using a straightforward induction on Lemma 6 we have the following corollary.

**Corollary 1:** Let $v$ be the value of $\text{seq}$ right after a successful $SC$ of line 46 such that $v > 1$. For every integer value $i$ such that $0 < i \leq v$ there is exactly one successful $SC$ execution in $\alpha$ of line 46 that sets the value of $\text{seq}$ to $i$ (Figure 7).

We now proceed with the following lemma.

**Lemma 7:** Let $S$ be any $SCAN$ that responds in $\alpha$ executed by a process $p$. Let $v' + 2$ be the value of $s_{\text{table}}[p].\text{seq}$ right after the execution of $SC_s^\text{table}$ by $S'$ ($S'$ may or may not be the same as $S$). Then, inside the execution interval of $S$, there is exactly one successful $SC$ operation that sets the value of $\text{seq}$ to $v' + 2$. Thus, $SC_{i+2}^\text{seq}$ is inside the interval of $S$.

**Proof:** We first prove that the value of $v'$ is greater or equal to 0, thus $v' + 2 > 1$. By inspection of the pseudocode lines 42 and 43, it follows that, $\text{seq}$ has a value equal to $v'$ just before the execution of line 42. The variable $\text{seq}$ is initialized with value 0 at $C_0$ and Observation 1 implies that the values of $\text{seq}$ do not decrease in any execution $\alpha$. Thus any read operation on $\text{seq}$ in $\alpha$ should return a number greater or equal to 0. It follows that $v' \geq 0$. Let $v$ be the value that is stored in $\text{seq}$ right after $SC_s^\text{table}$. $SC_s^\text{table}$ is executed in line 43 of $S'$. At line 42 of the pseudocode, $S'$ reads the value of $\text{seq}$, we denote this read operation with $r_s^\text{seq}$. At line 43 $S'$ executes $SC_s^\text{table}$. Thus, $S'$ stores the value that $r_s^\text{seq}$ reads increased by two, on $s_{\text{table}}[p].\text{seq}$. Therefore, $S'$ stores $v' + 2$ at $s_{\text{table}}[p].\text{seq}$. Since $S'$ stores the value $v' + 2$ at $s_{\text{table}}[p].\text{seq}$, it follows that at line 42 $S'$ reads the value of $\text{seq}$ and that value is $v'$. Since $r_s^\text{seq}$ precedes the $SC_s^\text{table}$, Observation 1 implies that the value of $\text{seq}$ right after $SC_s^\text{table}$ is greater or equal than the value of $\text{seq}$ right after $r_s^\text{seq}$, thus, $v' \leq v$. Lemma 4 implies that after $SC_s^\text{table}$ the value of $\text{seq}$ is increased at least two times inside the execution interval of $S$. Right after $SC_s^\text{table}$ the value of $\text{seq}$ is $v$, so there is a configuration.
We will now continue with the assignment of linearization points to any given configuration that exists and is unique in line 46 by some process that sets the value of the seq, which is at most \( v + 2 \). So, Corollary 1 implies that before \( C \) there was a successful SC of line 46 that updated the value of seq with the value \( v' + 2 \). We now prove that this SC, denoted by \( SC_{v' + 2}^{seq} \), is inside the execution interval of \( S \). C is inside the execution interval of \( S \) and \( SC_{v' + 2}^{seq} \) precedes \( C \) so \( SC_{v' + 2}^{seq} \) precedes the last configuration of the execution interval of \( S \). Furthermore right after \( r_{S}^{seq} \) the value of seq is \( v' \). The abovementioned read operation is between the \( LL_{S}^{S, table} \) and \( SC_{S}^{S, table} \), since \( LL_{S}^{S, table} \) reads the value that \( S \) stored in \( s, table \) after the execution of line 35 and \( SC_{S}^{S, table} \) is executed by \( S' \) at line 43 while \( r_{S}^{seq} \) is executed by \( S' \) at line 42. So the execution of line 35 by \( S \) precedes the \( LL_{S}^{S, table} \) and \( LL_{S}^{S, table} \) precedes \( r_{S}^{seq} \). Thus, \( r_{S}^{seq} \) is following the execution of line 35. It follows that inside the execution interval of \( S \) there is a configuration where the value of seq is \( v' \). The \( SC_{v' + 2}^{seq} \) follows that configuration but precedes the last configuration of the execution interval of \( S \). So \( SC_{v' + 2}^{seq} \) is inside the execution interval of \( S \). ■

\[ \begin{align*}
  w_{S}^{S, table} & \Rightarrow LL_{S}^{S, table} \Rightarrow SC_{v'}^{seq} \Rightarrow r_{S}^{seq} \Rightarrow SC_{v}^{seq} \Rightarrow SC_{S}^{S, table} \Rightarrow SC_{v + 1}^{seq} \Rightarrow SC_{v' + 2}^{seq} \Rightarrow C \\
  \text{In this interval many successful SC operations on seq may be executed.} \\
  \text{In this interval many successful SC operations on seq may be executed.} \\
  seq_{c} \leq v + 2
\end{align*} \]

**Figure 9:** There is exactly one successful SC on seq that sets the value of seq to \( v' + 2 \).

We will now continue with the assignment of linearization points to any given SCAN operation \( S \) that responds in \( a \). Let \( C_{S} \) be the configuration, right after \( SC_{S}^{S, table} \). Let \( x \) be the value of \( s, table[p] \). seq at \( C_{S} \). Denote by \( C_{x}^{seq} \) the configuration right after a successful execution of line 46 by some process that sets the value of seq to \( x \), consider that Lemma 7 implies that such a configuration exists and is unique in \( a \). We assign the linearization point of \( S \) at \( C_{x}^{seq} \).

If multiple SCAN operations are linearized in the same configuration we linearize them in ascending order based on process id.

We will now prove that the linearization point of any SCAN operation is inside its execution interval.

**Lemma 8:** Let \( S \) be a SCAN operation that responds in \( a \), the linearization point of \( S \) resides inside its execution interval.

**Proof:** Let \( v' + 2 \) be the value of \( s, table[p] \). seq right after the execution of \( SC_{S}^{S, table} \) by \( S' \) (\( S' \) may or may not be the same as \( S \)). Lemma 7 implies that inside the execution interval of \( S \), there is exactly one successful SC operation that sets the value of seq to \( v' + 2 \). Thus, \( SC_{v' + 2}^{seq} \) is inside the interval of \( S \). \( S \) is linearized right after \( SC_{v' + 2}^{seq} \) which resides inside its execution interval. It follows that Lemma 8 stands true in any case.

In order to assign linearization points to any UPDATE that responds in \( a \). We need to first state some observations.
Observation 3: Any \textit{UPDATE} operation can execute at maximum one successful \textit{SC} of line 26.

Observation 4: Any register of the \textit{values} array can only be modified by the execution of lines 26 or 81.

Observation 5: Let \(j\) be any component of the snapshot object, then the value of \(\text{values}[j]\) is modified only by lines 26 and 81.

We will now study the successful \textit{SC} operations on any register of the \textit{values} array. Given any integer value \(j\), such that \(m > j \geq 0\), we denote with \(SC_{i}^{val[j]}\) the \(i\)th successful \textit{SC} on \(\text{values}[j]\) register. Thus, we denote by \(SC^{val[j]} = SC_{0}^{val[j]}, SC_{1}^{val[j]}, \ldots\) the sequence of successful \textit{SC} on \(\text{values}[j]\) register. There is a chance in some executions and for some value \(j\), the sequence of successful \textit{SC} on \(\text{values}[j]\) to be empty, so \(SC^{val[j]} = \emptyset\).

Lemma 9: Given an execution \(a\) and an integer value \(j\) such that \(m > j \geq 0\) and let \(SC_{0}^{val[j]}, SC_{1}^{val[j]}, \ldots\) be a non-empty sequence of \textit{SC} on \(\text{values}[j]\). Then for any \(SC_{t}^{val[j]}\) (\(t\) is an integer value) that is term of the abovementioned sequence, it follows that if \(t \mod 2 = 0\) then the \(SC_{t}^{val[j]}\) is a successful \textit{SC} of line 26 and is of type announce, otherwise \(SC_{t}^{val[j]}\) is a successful \textit{SC} of line 81 and is of type apply.

Proof: We will prove this statement using mathematical induction on the amount \(t\) of successful \textit{SC} instructions in \(a\).

\textbf{Induction Base:} \(SC_{0}^{val[j]}\) is a successful \textit{SC} of line 26 and \(SC_{1}^{val[j]}\) is a successful \textit{SC} of type apply. (Note that this is true as long as \(SC_{1}^{val[j]}\) is a term of \(SC^{val[j]}\). On the other hand, \(SC_{0}^{val[j]}\) is a term of \(SC^{val[j]}\) since the \(SC^{val[j]}\) is a non-empty sequence)

\textbf{Proof of Induction Base:} By inspection of the pseudocode line 11 it follows that at the initial configuration of \(a\), the value of \textit{value}[j].\textit{proposed}_value\ is set to be \textit{NULL}. Observation 4 implies that the value of \textit{value}[j] can only be modified by \textit{SC} operations of line 26 and 81. By inspection of the pseudocode (lines 64, 77, 81), it follows that the \textit{SC} of line 81 can only be executed in case that the if condition of line 77 is evaluated to true. If an \textit{SC} of line 81 is the first \textit{SC} of the \(SC^{val[j]}\) then its matching \textit{LL} should read the preset values of \(\text{values}[j]\). In that case the if condition of line 77 would not be evaluated to true. Thus, \(SC_{0}^{val[j]}\) cannot be an \textit{SC} of type apply. On the other hand by inspection of lines 15, 22-26 it follows that \(SC_{0}^{val[j]}\) is an \textit{SC} of type announce. We will now prove that if \(SC_{1}^{val[j]}\) is a term of the \(SC^{val[j]}\) then it should be an \textit{SC} of type apply. Since \(SC_{0}^{val[j]}\) is an \textit{SC} of type announce and by inspection of the pseudocode (lines 22-26), it follows that \(SC_{0}^{val[j]}\) sets the value of \textit{values}[j].\textit{proposed}_value\ to be a non \textit{NULL} value. Observation 5 implies that \(\text{values}[j]\) can only be modified by successful \textit{SC} operations of line 26 or 81. It follows that between \(SC_{0}^{val[j]}, SC_{1}^{val[j]}\) the value of \textit{value}[j] remains the same. Let \(op\) be the operation that executes \(SC_{1}^{val[j]}\). Since \(SC_{1}^{val[j]}\) is a successful \textit{SC} on \(\text{values}[j]\) it follows that its matching \textit{LL} should follow the execution of \(SC_{0}^{val[j]}\). Thus, the matching \textit{LL} of \(SC_{1}^{val[j]}\) should read the value that \(SC_{0}^{val[j]}\) stored in \textit{values}[j]. It follows, (by inspection of the pseudocode, lines 22-26) that the if condition of line 25 is evaluated to false by \(op\) (if ever executed). Thus, \(op\) never executes line 26 and \(SC_{1}^{val[j]}\) cannot be an \textit{SC} of type announce. On the other hand, by inspection of the pseudocode (lines 64, 77, 81) it follows that the if condition of line 77 is
evaluated to true by \( op \) and \( op \) successfully executes line 81. Thus, \( SC_{t+1}^{\text{val}}[j] \) is an \( SC \) of type apply.

**Induction Hypothesis:** If \( SC_t^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \) then \( SC_t^{\text{val}}[j] \) is an \( SC \) of type announce if \( mod 2 = 0 \), else \( SC_t^{\text{val}}[j] \) is an \( SC \) of type apply.

**Induction Step:** If \( SC_t^{\text{val}}[j] \) is a term of \( SC^{\text{val}}[j] \) and \( SC_{t+1}^{\text{val}}[j] \) is also a term of \( SC^{\text{val}}[j] \) then \( SC_{t+1}^{\text{val}}[j] \) is an \( SC \) of type announce if \((t + 1) mod 2 = 0 \), else \( SC_t^{\text{val}}[j] \) is an \( SC \) of type apply.

**Proof of Induction Step:** We proceed with case analysis.

- **Case 1:** Let \( t \ mod \ 2 = 0 \), then \( (t + 1) \ mod \ 2 = 1 \). Since \( t \ mod \ 2 = 0 \), by our induction hypothesis, it follows that \( SC_t^{\text{val}}[j] \) is an \( SC \) of type apply. By definition, between \( SC_t^{\text{val}}[j] \) and \( SC_{t+1}^{\text{val}}[j] \) there is no successful \( SC \) on \( values[j] \). By inspection of the pseudocode (lines 22-26), it follows that right after the execution of \( SC_t^{\text{val}}[j] \), the value of \( values[j] \), \( \text{proposed\_value} \) is not equal to \( \text{NULL} \). Since \( SC_{t+1}^{\text{val}}[j] \) is a successful \( SC \), it follows that its matching \( LL \) operation follows \( SC_t^{\text{val}}[j] \), as shown in Figure 1. The matching \( LL \) of \( SC_{t+1}^{\text{val}}[j] \) reads the value that \( SC_t^{\text{val}}[j] \) stored in \( values[j] \). By inspection of the pseudocode (lines 22-26), it follows that the matching \( LL \) of \( SC_{t+1}^{\text{val}}[j] \) cannot be an \( LL \) of line 22, since in that case the if condition of line 25 would be evaluated to false, preventing the execution of \( SC \) of type announce. Thus, the matching \( LL \) of \( SC_{t+1}^{\text{val}}[j] \) is an \( LL \) of line 64. By inspection of the pseudocode (lines 64 and 77-81), it follows that the if condition of line 77 is evaluated to true and thus the \( SC \) of line 81 is executed successfully. It follows that \( SC_{t+1}^{\text{val}}[j] \) is an \( SC \) of type apply.

- **Case 2:** Let \( t \ mod \ 2 = 1 \), then \((t + 1) \ mod \ 2 = 0 \). Since \( t \ mod \ 2 = 1 \), by our induction hypothesis, it follows that \( SC_t^{\text{val}}[j] \) is an \( SC \) of type apply. Between \( SC_t^{\text{val}}[j] \) and \( SC_{t+1}^{\text{val}}[j] \) there is no successful \( SC \) on \( values[j] \), so the value of \( values[j] \) does not change in this execution interval. By inspection of the pseudocode (lines 78-81), it follows that right after the execution of \( SC_t^{\text{val}}[j] \) the value of \( values[j] \), \( \text{proposed\_value} \) is \( \text{NULL} \). Using the same argument as in case 1 it follows that \( SC_{t+1}^{\text{val}}[j] \) is an \( SC \) of type announce. ■

---

**Figure 10:** Two \( SC \) operations of \( SC^{\text{val}}[j] \) with their corresponding \( LL \) operations.

In order to assign linearization points to \( UPDATE \) operations, we study the \( UPDATE \) operations that successfully execute an \( SC \) of line 26 by proposing the following definition and prove lemma 4.

Let \( U \) be any \( UPDATE \) operation in \( a \). We denote by \( C^{st}_U \) the configuration right after the invocation of \( U \). The value of \( seq \) at \( C^{st}_U \) is denoted by \( seq^{st}_U \).
Lemma 10: Let $U$ be an UPDATE operation that wants to store the value $t$ to the $i$-th component such that: (a) $U$ successfully executes an SC of line 26, denoted by $SC_{\text{ann} U}^{\text{val} [i]}$ (b) inside the execution interval of $U$ and after $SC_{\text{ann} U}^{\text{val} [i]}$ at least one SC of line 81 we denote the first such SC by $SC_{\text{app} U}^{\text{val} [i]}$. $SC_{\text{app} U}^{\text{val} [i]}$ is executed by an operation $op$ that may, or may not be the same as $U$. At the configuration right after the execution of $SC_{\text{app} U}^{\text{val} [i]}$, denoted with $C_{\text{app} U}^{\text{val} [i]}$, the value of $\text{values} [i]. value$ is $t$ and the value of $\text{values} [i]. seq$ is greater or equal to $seq_{st}$. 

Proof: By inspection of the pseudocode (lines 22-26), it follows that at $C_{\text{ann} U}^{\text{val} [i]}$ the value of $\text{values} [i]. proposed. value$ will be $t$. Observation 5 implies that the struct $\text{values} [i]$ can only be modified by the execution of lines 26 and 81. Lemma 9 implies that an SC of line 26 cannot be successfully executed between $SC_{\text{ann} U}^{\text{val} [i]}$ and $SC_{\text{app} U}^{\text{val} [i]}$. Furthermore, $SC_{\text{app} U}^{\text{val} [i]}$ is the first SC of line 81 that follows $SC_{\text{ann} U}^{\text{val} [i]}$. Thus, between $SC_{\text{ann} U}^{\text{val} [i]}$ and $SC_{\text{app} U}^{\text{val} [i]}$ there is no successful SC on $\text{values} [i]$. Let $LL_{\text{app} U}^{\text{val} [i]}$ be the matching LL operation of $SC_{\text{app} U}^{\text{val} [i]}$ executed by $op$. Given that $SC_{\text{app} U}^{\text{val} [i]}$ is executed successfully $LL_{\text{app} U}^{\text{val} [i]}$ should follows the execution of $SC_{\text{ann} U}^{\text{val} [i]}$. It follows that $LL_{\text{app} U}^{\text{val} [i]}$ reads the value that $SC_{\text{ann} U}^{\text{val} [i]}$ stores in $\text{values} [i]$ (as shown in Figure 2). Right after $LL_{\text{app} U}^{\text{val} [i]}$, $op$ reads the value of $seq$ (line 65), we denote this value with $seq_{U}$. The value $seq_{U}$ is greater or equal than $seq_{st}$, this is true since inside $a$ the value of $seq$ never decreases (Observations 1 and 2). By inspection of the pseudocode (lines 64, 65 and 77-81) it follows that $op$ evaluates the if condition of line 77 to true and with the execution of $SC_{\text{app} U}^{\text{val} [i]}$ sets the value of $\text{values} [i]. value$ to $t$ and the value of $\text{values} [i]. seq$ to the value of $seq$ that $op$ reads with the execution of line 65. So lemma 10 stands true. 

Figure 11: At $C_{t}^{\text{val} [i]}$ the value of $\text{values} [i]. value$ would be $t$.

Lemma 10 introduces the notation $seq_{U}$ that refers to the value that $SC_{\text{app} U}^{\text{val} [i]}$ stores to $\text{values} [i]. seq$. The processes that executed the $SC_{\text{app} U}^{\text{val} [i]}$ read the value of $seq$ with the execution of line 65 and then stores this value to $\text{values} [i]. seq$ without modifying it. Thus, right before the abovementioned read of line 65, the value of $seq$ is equal to $seq_{U}$. Furthermore, Observation 1 implies that only SCAN operations can modify the value of $seq$. Thus the following observation stands true.

Observation 6: Let a be any execution of $\lambda - OPT$. Let also $U$ be an UPDATE operation that wants to store the value $t$ to the $i$-th component such that: (a) $U$ successfully executes an SC of line 26, denoted by $SC_{\text{ann} U}^{\text{val} [i]}$ (b) inside the execution interval of $U$ and after $SC_{\text{ann} U}^{\text{val} [i]}$ at least one SC of line 81 is successfully executed. Then, inside a there is a SCAN operation that stores the value $seq_{U}$ to $seq$ before the response of the execution of $U$. 

37
We now assign a linearization point to any \textit{UPDATE} operation \( U \) that successfully executes an \textit{SC} of line 26 (\( SC_{\text{ann},U}^{\text{val}[i]} \)) and inside its execution interval but after \( SC_{\text{ann} U}^{\text{val}[i]} \), there is a successful execution of an \textit{SC} of line 81. We distinguish the following two cases:

- **Case 1:** Right after the execution of \( SC_{\text{app},U}^{\text{val}[i]} \), the value of \( \text{seq} \) is equal to \( \text{seq}_U \). \( U \) is linearized right after the execution of \( SC_{\text{app},U}^{\text{val}[i]} \).
- **Case 2:** Right after the execution of \( SC_{\text{app},U}^{\text{val}[i]} \), the value of \( \text{seq} \) is greater than \( \text{seq}_U \). \( U \) is linearized at \( \text{seq}_U + 1 \). Recall that \( SC_{\text{seq} \text{seq}_U+1}^{\text{seq}} \) is the \textit{SC} operation of line 46 that sets the value of \( \text{seq} \) to \( \text{seq}_U + 1 \). Since in case 2 the value of \( \text{seq} \) right after the execution of \( SC_{\text{app},U}^{\text{val}[i]} \) is greater or equal than \( \text{seq}_U \), it should be greater or equal to \( \text{seq}_U + 1 \) (Observation 2 implies that the value of \( \text{seq} \) is increased only by one each time). Thus, before the execution of \( SC_{\text{app},U}^{\text{val}[i]} \) a \textit{SCAN} operation set the value of \( \text{seq} \) to \( \text{seq}_U + 1 \).

Finally, we assign linearization points to any \textit{UPDATE} operation \( U \) that respond in a but do not execute a successful \textit{SC} of line 26.

- **Case 3:** Inside the execution interval of \( U \) an \textit{UPDATE} operation, which is assigned a linearization point using case 1 or case 2, is linearized. Let \( U' \) be the first such operation. We assign the linearization point of \( U \) right before the linearization point of \( U' \) (We show in lemma 8 that there is at least one \textit{UPDATE} operation that is linearized inside the execution interval of \( U \). We prove in lemma 9 that the linearization point of \( U \) resides inside the execution interval of \( U \)).

If an \textit{UPDATE} operation is linearized at the same configuration as another \textit{SCAN} operation, then the \textit{UPDATE} operation is linearized before the \textit{SCAN} operation.

We now show that the linearization point that is assigned in \textit{UPDATE} operations that belong to case 1 or case 2 resides inside their execution interval.

**Lemma 11:** Let also \( U \) be an \textit{UPDATE} operation that wants to store the value \( t \) to the \( i - \text{th} \) component such that: (a) \( U \) successfully executes an \textit{SC} of line 26, denoted by \( SC_{\text{ann},U}^{\text{val}[i]} \) (b) inside the execution interval of \( U \) and after \( SC_{\text{ann} U}^{\text{val}[i]} \) at least one \textit{SC} of line 81 is successfully executed by some operation \( op \) (\( op \) may, or may not be the same as \( U \)). The linearization point of \( U \) resides inside its execution interval.

**Proof:** If \( U \) is linearized right after the execution of \( SC_{\text{app},U}^{\text{val}[i]} \) then its linearization point belongs to the execution interval of \( U \). If \( U \) is linearized at \( SC_{\text{seq} \text{seq}_U+1}^{\text{seq}} \) then, we will show that \( SC_{\text{seq} \text{seq}_U+1}^{\text{seq}} \) is inside the execution interval of \( U \). The value \( \text{seq}_U \) is the value of \( \text{seq} \) that \( op \) reads after the execution of line 65, we denote this operation with \( r_{\text{seq} \text{seq}_U} \). \( r_{\text{seq} \text{seq}_U} \) follows the execution of \( LL_{\text{app},U}^{\text{val}[i]} \) and precedes the execution of \( SC_{\text{app},U}^{\text{val}[i]} \). Both \( LL_{\text{app},U}^{\text{val}[i]} \) and \( SC_{\text{app},U}^{\text{val}[i]} \) reside inside the execution interval of \( U \). Thus, the execution of \( r_{\text{seq} \text{seq}_U} \) is inside the execution interval of \( U \).

Although, right after \( SC_{\text{app},U}^{\text{val}[i]} \), the value of \( \text{seq} \) is greater or equal than \( \text{seq}_U \) (\( U \) is linearized following the case 2). Since \( \text{seq}_U < \text{seq}_U + 1 \) and \( \text{seq}_U + 1 \) is smaller or equal than the value of \( \text{seq} \) right after \( SC_{\text{app},U}^{\text{val}[i]} \), it follows, that somewhere between the \( r_{\text{seq} \text{seq}_U} \) and \( SC_{\text{app},U}^{\text{val}[i]} \) there was a \textit{SCAN} operation that set the number of \( \text{seq} \) to \( \text{seq}_U + 1 \). Thus, \( SC_{\text{seq} \text{seq}_U+1}^{\text{seq}} \) is inside the execution interval of \( U \). It follows that lemma 11 stands true in any given case.
The proof of lemma 11 implies that any UPDATE operation $U$ that is linearized based on the abovementioned assignment of linearization points has its linearization point inside its execution interval and somewhere in between $LL_{app,u}^{val[i]}$ and the configuration right after $SC_{app,u}^{val[i]}$.

Observation 7: Let also $U$ be an UPDATE operation that wants to store the value $t$ to the $i-th$ component such that: (a) $U$ successfully executes an SC of line 26, denoted by $SC_{ann,u}^{val[i]}$ (b) inside the execution interval of $U$ and after $SC_{ann,u}^{val[i]}$ at least one SC of line 81 is successfully executed by some operation $op$ ($op$ may, or may not be the same as $U$). $U$ has its linearization point somewhere between $LL_{app,u}^{val[i]}$ and the configuration right after $SC_{app,u}^{val[i]}$.

We now continue through proving the following lemma.

Lemma 12: Let $U$ be an UPDATE operation that wants to store the value $t$ to the $i-th$ component such that: (a) $U$ successfully executes an SC of line 26, denoted by $SC_{ann,u}^{val[i]}$ (b) outside the execution interval of $U$, after $SC_{ann,u}^{val[i]}$ at least one SC of line 81 is successfully executed by some operation $op$ ($op$ may, or may not be the same as $U$). Then, inside the execution interval of $U$ and after $SC_{ann,u}^{val[i]}$ at least one SC of line 81 is successfully executed by some operation $op'$ ($op'$ may, or may not be the same as $U$).

Proof: Let’s assume by contradiction that there is an UPDATE operation $U$ such that statements (a) and (b) stand true, but there is no operation $op'$ that successfully executes an SC of line 81 inside the execution interval of $U$ and after $SC_{ann,u}^{val[i]}$. Right after $SC_{ann,u}^{val[i]}$ the value of $values[i].proposed_value$ is equal to $t$. Inside the execution interval of $U$ and after $SC_{ann,u}^{val[i]}$ there is no successful execution of an SC of line 81. Although there is an SC of line 81 successfully executed after $SC_{ann,u}^{val[i]}$, we denote the first such SC operation as $SC_1$. Between $SC_{ann,u}^{val[i]}$ and $SC_1$ there is no successful SC on $values[i]$. By inspection of the pseudocode (lines 26, 27) it follows that $U$ invokes an ApplyUpdate function after $SC_{ann,u}^{val[i]}$. By inspection of the pseudocode (lines 64 and 77-82), it follows that the if condition of line 77 will be evaluated to true by $U$ ($LL$ of line 64 reads the value that $SC_{ann,u}^{val[i]}$ stored in $values[i]$). Thus, an SC of line 81 is executed by $U$ inside its execution interval. The abovementioned SC operation is successful, which is a contradiction. Thus, lemma 12 stands true in any case. 

We now study UPDATE operations that are linearized using case 3.

Lemma 13: Let $U$ be an UPDATE operation that wants to update the $i-th$ component of the snapshot object with the value $t$ such that: (a) $U$ responds in a but, (b) $U$ does not execute a successful SC of line 26. Then, inside the execution interval of $U$, there is some operation $U'$ that executes $LL_{app,u}^{val[i]}$, and some operation $op$ that executes $SC_{app,u}^{val[i]}$.

Proof: Let’s assume by contradiction that there is no UPDATE operation $U'$ that executes $LL_{app,u}^{val[i]}$, and an operation $op$ that executes $SC_{app,u}^{val[i]}$ inside the execution interval of $U$. Let’s study the first iteration of the loop of lines 21-32. Since $U$ does not execute successfully an SC of line 26 it follows that either the if condition of line 25 is evaluated to false or the execution was evaluated to true but the execution of line 26 was unsuccessful.
Let’s study the case where the execution of line 25 is evaluated to false. By inspection of the pseudocode (lines 22 and 25), it follows that right after the execution of line 22 by \( U \), the value of \( \text{values}[j] \), \( \text{proposed}_{\text{value}} \) is not NULL. Since only SC operations of type announce write a value that is not NULL in \( \text{values}[j] \), \( \text{proposed}_{\text{value}} \) it follows that, the last successful SC on \( \text{values}[j] \) was an SC of type announce. Let’s denote this SC operation as \( \text{SC}^{U}_{\text{ann},1} \), note that this SC operation may or may not be inside the execution interval of \( U \). \( U \) executes line 27 and afterwards lines 64 and 77-82 are executed by \( U \). Following the same reasoning, it follows that after the execution of the LL of line 22 and before the execution of line 82 an SC of type apply is successfully executed. We denote this SC by \( \text{SC}^{U}_{\text{app},1} \).

The case where the execution of line 25 is evaluated to true is similar.

Following the same reasoning for the second loop of lines 21-32, it follows that, after \( \text{SC}^{U}_{\text{app},1} \) and inside the execution interval of \( U \) there is an SC of type announce denoted by \( \text{SC}^{U}_{\text{ann},2} \). Furthermore, there is a successful SC operation of type apply, denoted by \( \text{SC}^{U}_{\text{app},2} \), that is inside the execution interval of \( U \) and follows the execution of \( \text{SC}^{U}_{\text{ann},2} \). Since \( \text{SC}^{U}_{\text{ann},2} \) is an SC of type announce executed inside the execution interval of \( U \) it follows that \( \text{SC}^{U}_{\text{ann},2} \) is executed by some process \( U' \). So \( U' \) executed \( \text{SC}^{\text{val}[i]}_{\text{ann},U'} \) inside the execution interval of \( U \).

Lemma 12 implies that, since the execution of \( \text{SC}^{\text{val}[i]}_{\text{ann},U} \) is followed by a successful SC execution of type apply (\( \text{SC}^{U}_{\text{app},2} \)) it follows that \( U' \) is an UPDATE operation that is described in Lemma 10. The first successful SC of type apply that follows \( \text{SC}^{\text{val}[i]}_{\text{ann},U} \) is the \( \text{SC}^{\text{val}[i]}_{\text{app},U} \). Thus, \( \text{SC}^{U}_{\text{app},2} \) may or may not be the same as \( \text{SC}^{\text{val}[i]}_{\text{app},U} \) but if \( \text{SC}^{\text{val}[i]}_{\text{app},U} \) is not the same with \( \text{SC}^{U}_{\text{app},2} \) then \( \text{SC}^{\text{val}[i]}_{\text{app},U} \) precedes \( \text{SC}^{U}_{\text{app},2} \). The corresponding LL operation of \( \text{SC}^{\text{val}[i]}_{\text{app},U} \) precedes \( \text{SC}^{\text{val}[i]}_{\text{app},U} \), and follows \( \text{SC}^{\text{val}[i]}_{\text{ann},U} \). Thus, \( \text{LL}^{\text{val}[i]}_{\text{app},U} \), and \( \text{SC}^{\text{val}[i]}_{\text{app},U} \), are inside the execution interval of \( U \), which is a contradiction. Lemma 13 stands true in any case. ■

We now prove that inside the execution interval of an UPDATE operation that is linearized using the case 3 another UPDATE operation is linearized.

Lemma 14: Let \( U \) be an UPDATE operation that wants to update the \( i - h \) component of the snapshot object with the value \( t \) such that: (a) \( U \) responds in \( a \) but, (b) \( U \) does not execute a successful SC of line 26. Then, inside the execution interval of \( U \) another UPDATE operation, denoted \( U' \), is linearized.

Proof: Lemma 13 implies that inside the execution interval of \( U \) there is some operation \( U' \) that executes \( \text{LL}^{\text{val}[i]}_{\text{app},U} \), and some operation \( op \) that executes \( \text{SC}^{\text{val}[i]}_{\text{app},U} \). \( U' \) is linearized somewhere in between \( \text{LL}^{\text{val}[i]}_{\text{app},U} \) and the configuration right after \( \text{SC}^{\text{val}[i]}_{\text{app},U} \) (Observation 7). Thus Lemma 14 stands true. ■

We now prove that the linearization point of an UPDATE of case 3 is inside its execution interval.

Lemma 15: Let \( U \) be an UPDATE operation that wants to update the \( i - h \) component of the snapshot object with the value \( t \) such that: (a) \( U \) responds in \( a \) but, (b) \( U \) does not execute a successful SC of line 26. Let \( U' \) be the first UPDATE operation that is linearized inside the execution interval of \( U \). We assign the linearization point of \( U \) right before the linearization point of \( U' \). The linearization point of \( U \) resides inside the execution interval of \( U \).
**Proof:** Lemma 14 implies that there is at least one *UPDATE* operation that is linearized inside the execution interval of *U*. We denote the first such *UPDATE* operation with *U′*. Since *U* is linearized right before *U′*, it follows that the linearization point of *U* resides in its execution interval. ■

In order to prove that we assigned a linearization point on every operation that responds in a we need to prove the following lemma.

**Lemma 16:** Let *U* be any *UPDATE* operation that wants to change the value of the *i* – *th* component, of the snapshot object, and responds in *a*, then a linearization point is assigned to *U*.

**Proof:** *U* either executed a successful *SC* of type announce or it did not. Let’s first study the case where *U* didn’t execute a successful *SC* of type announce. Since *U* responds in *a*, *U* belongs to a case 3 *UPDATE* and is linearized as such. Let’s now study the case where *U* successfully executed an *SC* of type announce. If, after *SC*\(_{\text{ann},U}^{\text{val}}[i]*, there is a successful *SC* of type apply, then Lemma 12 implies that *U* is an *UPDATE* operation of case 1 or case 2. If after *SC*\(_{\text{ann},U}^{\text{val}}[i]*, there is no successful *SC* of type apply, we will prove that at least *U* executes an *SC* of type apply and that should be successful which would be a contradiction. *SC*\(_{\text{ann},U}^{\text{val}}[i]* is the last successful *SC* operation that changes the value of *values*[i]. By inspection of the pseudocode (lines 26, 27) it follows that *U* invokes an *ApplyUpdate* function after *SC*\(_{\text{ann},U}^{\text{val}}[i]*. By inspection of the pseudocode (lines 64 and 77-82), it follows that the if condition of line 77 will be evaluated to true by *U* (*LL* of line 64 reads the value that *SC*\(_{\text{ann},U}^{\text{val}}[i]* stored in *values*[i]). So, an *SC* of line 81 is executed by *U* inside its execution interval. The abovementioned *SC* operation is successful, which is a contradiction. Thus, Lemma 16 always stands true. ■

### 5.3. Step Complexity of λ-OPT

The step complexity of an operation of λ – OPT is measured by the number of operations that are executed in shared registers, inside its execution interval.

We start with the worst-case analysis of *ApplyUpdate*.

1. In lines 62-65 only an *LL* operation is performed at line 68 and a read of shared variable *seq* (line 69).
2. In lines 66-76 contain a loop that is executed exactly λ times. In any iteration of this loop the loop of lines 67-75 is executed exactly two times. In any iteration of the later loop, four shared register operations are executed at maximum. An *LL* at line 68, two read operations (line 69 and 70) and an *SC* operation at line 73. Thus, the loop of lines 67-75 executes at maximum eight shared register operations. Furthermore, the loop of lines 67-75 is a nested loop of that of lines 66-76, so it is executed exactly λ times. It follows that the loop of lines 66-76 executes at maximum 8 · λ shared register operations.
3. Lines 77-82 contain just a single *SC* operation (line 81).

It follows that *ApplyUpdate* executes at maximum 3 + 6 · λ shared memory accesses. Thus, *ApplyUpdate* has a step complexity of O(λ).

We now proceed with the worst-case analysis of the step complexity of any *UPDATE*. The loop of lines 21-32 can be executed two times at maximum and contains an *LL* (line 22), an
SC (line 26) and two invocations of ApplyUpdate (lines 27 and 31). We previously proved that any ApplyUpdate executes $O(\lambda)$ shared memory accesses. It follows that any UPDATE operation executes $O(\lambda)$ shared memory accesses.

We can finally proceed with the worst-case analysis of step complexity of any SCAN.

1. A write operation on the shared table $s\_table$ is executed in line 35.
2. Lines 36-47 contain a loop that is executed exactly three times. In each iteration of that loop, an LL is executed at line 37 and an SC at line 46. Furthermore, the loop of lines 38-45 is executed, and exactly $\lambda$ iterations of it are performed. In any iteration of loop of lines 38-45 at maximum three shared memory accesses are performed (an LL at line 39, a read of the shared seq variable at line 40 and an SC at line 43). It follows that the loop of lines 38-45 executes $O(\lambda)$ shared memory accesses. Since the loop of lines 36-47 is executed exactly three times it executes $O(\lambda)$ shared memory accesses.
3. Lines 49-58 contain a loop that is executed exactly $m$ times. In each iteration of that loop an ApplyUpdate is invoked (line 50) and two read operations are performed (lines 51, 52). Since ApplyUpdate executes $O(\lambda)$ shared memory accesses and at lines 49-58 is invoked exactly $m$ times it follows that lines 49-58 execute $O(\lambda m)$ shared memory accesses.

It follows that any SCAN operation executes $O(\lambda m)$ shared memory accesses.

5.4. Space Complexity of $\lambda$-OPT

The space complexity of $\lambda - OPT$ algorithm is measured through counting the number of shared registers that are needed for its implementation. The implementation of $\lambda - OPT$ deploys four different shared objects:

1. A shared integer variable called seq which is stored in an unbounded LL/SC register (a sequence number).
2. A shared table called values that is consisted of $m$ LL/SC registers of unbounded size (a sequence number and two integer values).
3. A shared table called pre\_values that is consisted of $\lambda m$ LL/SC registers of unbounded size (a sequence number and one integer value).
4. A shared table called s\_table that is consisted of $\lambda$ LL/SC write registers of unbounded size (a sequence number and one Boolean value).

Thus, our implementation deploys $1 + m + \lambda m + \lambda LL/SC write$ unbounded registers. It follows that the space complexity of our algorithm is $O(\lambda m)$.

The implementation of $\lambda - OPT$ presented in this work uses LL/SC write registers of unbounded size. Although, registers should be unbounded it can be proven that they need to have a size of $O(\log(s))$, where $s$ is the maximum number of SCANS in a given execution. Thus, in executions that the maximum number of SCAN operation is relative low $\lambda OPT$ may use bounded registers.

Theorem 2: $\lambda - OPT$ is a wait-free linearizable concurrent $\lambda$-scanner snapshot implementation that uses $O(\lambda m)$ registers, provides $O(\lambda)$ step complexity to any UPDATE operation and $O(\lambda m)$ to any SCAN operation.
5.5. A partial version of $\lambda$-OPT

We now present a modified version of $\lambda - \text{OPT}$ that implements a partial snapshot object. The data structures used in this modified version of $\lambda - \text{OPT}$ remain the same, as shown in Algorithm 11. Furthermore, the pseudocode of the UPDATE operation and the ApplyUpdate function remain the same as shown in Algorithms 12 and 14. A new function is introduced called Read (Algorithm 13). This function is invoked by SCAN operations in order to read the values of the snapshot object.

Algorithm 11. Data Structures of $\lambda$-OPT (partial version)

1. struct valuestruct {
2.  weightval  value;
3.  int        seq;
4.  weightval  proposed_value;
5. }

6. struct pre_valuestruct {
7.  weightval  value;
8.  int        seq;
9. }

10. struct Scanstruct{
11.  int        seq;
12.  boolean    write_enable;
13. }

14. shared int seq;

15. shared valuestruct values[0..m-1]=
   [<NULL,NULL,NULL>,...,<NULL,NULL,NULL>];

16. shared valuestruct pre_values[0..m-1]=
   [<NULL,NULL>,...,<NULL,NULL>];

17. shared scanstruct s_table[0..l-1]=
   [<NULL,0>,<NULL,0>,...,<NULL,0>];

18. private int view[0..m-1]=[NULL,NULL,...,NULL,NULL];

A SCAN operation through helping tries to get a sequence number and then executes a for loop in order to read the components of the snapshot object. For each integer $j$ that is contained in $A$ (the set that contains the components that a SCAN wants to read), the SCAN operation tries to help an UPDATE operation that wants to update the value of $c_j$ component by invoking the ApplyUpdate function. Afterwards, it reads the value of $c_j$ by invoking the Read function.

The only modification in this version of $\lambda - \text{OPT}$ is that the SCAN operations do not read every component of the snapshot object, they only read the components of set $A$. The correctness proof of the partial version of $\lambda - \text{OPT}$ is outside of the scope of this work.

Both partial $\lambda - \text{OPT}$ and non-partial $\lambda - \text{OPT}$ have the same step complexity of UPDATE operations, and the same space complexity. Although, $\lambda - \text{OPT}$ provides a step complexity to SCAN operations of $O(\lambda r)$ where, $r$ is the number of elements contained in $A$. 

43
Simpler said, \( r \) is the number of different components that the SCAN operation reads. Since, \( r \leq m \) the step complexity that partial \( \lambda - OPT \) provides to SCAN operations is lower than that the non-partial \( \lambda - OPT \). Recall that non-partial \( \lambda - OPT \) provides a step complexity of \( O(\lambda m) \) to any SCAN operation.

**Algorithm 12. UPDATE and SCAN implementations of \( \lambda \)-OPT (partial)**

```c
19. void UPDATE(int j, int value){
20.  struct valuesstruct up_value, cur_value;
21.  for (i=0; i<2; i++){
22.    cur_value=LL(values[j]);
23.    up_value=cur_value;
24.    up_value.proposed_value=value;
25.    if (cur_value.proposed_value==NULL){
26.      if (SC(values[j],up_value)){
27.        ApplyUpdate(j);
28.        break;
29.      }
30.    }
31.    ApplyUpdate(j);
32.  }

33. }

34. pointer SCAN(set A){
35.  s_table[p_id]={1,seq};
36.  for (i=0;i<3;i++){
37.    cur_seq=LL(seq);
38.    for (j=0;j<\lambda;j++){
39.      cur_s_table=LL(s_table[j]);
40.      if(cur_s_table.seq<seq+2 && cur_s_table.write_enable==1){
41.        cur_s_table.write_enable=0;
42.        cur_s_table.seq=seq+2;
43.        SC(s_table[j],cur_s_table);
44.      }
45.    }
46.    SC(seq,cur_seq+1);
47.  }
48. }
49. for each j in A){
50.  ApplyUpdate(j);
51.  Read(j);
52. }
53. }
```
Algorithm 13. **Read** implementation of 1-OPT (partial)

54. int Read(j){
55.    struct value_struct v1;
56.    struct pre_value_struct v2;
57.    v1=values[j];
58.    v2=pre_values[j];
59.    if (v1.seq<seq){
60.        view[j]=v1.value;
61.    }else{
62.        view[j]=v2.value;
63.    }
64.    return view[j];
65. }

Algorithm 14. **ApplyUpdate** implementation of 1-OPT (partial)

66. void ApplyUpdate(int j){
67.    struct value_struct cur_value;
68.    struct pre_value_struct cur_pre_value, proposed_pre_value
69.    cur_value=LL(values[j]);
70.    cur_seq=seq;
71.    for (i=0; i<λ; i++){
72.        for (t=0; t<2; t++){
73.            cur_pre_value=LL(pre_values[i].Val[j]);
74.            cur_value=values[j];
75.            if (cur_value.seq<s_table[j].seq){
76.                proposed_pre_value.seq=cur_value.seq;
77.                proposed_pre_value.value=cur_value.value;
78.                SC(pre_values[i].Val[j], proposed_pre_value);
79.            }
80.        }
81.    }
82.    if (cur_value.proposed_value!=NULL){
83.        cur_value.value=cur_value.proposed_value;
84.        cur_value.seq=cur_seq;
85.        cur_value.proposed_value=NULL;
86.        SC(values[j], cur_value);
87.    }
88. }

45
6. Discussion and Future Work

6.1. Discussion

In this work, we present two solutions to the snapshot problem. Our first implementation $1 - OPT$ solves the single-scanner version of snapshot problem. Although, in our algorithm, we only allow one process with a certain id to invoke $SCAN$ operations, this is a restriction that can be easily lifted. The system can support invocations of $SCAN$ operations by any process, although only one process can be active in any given configuration of the execution. In this case, our algorithm would be correct only in executions that no more than one $SCAN$ is active in any given configuration of the execution.

In this work an object called $\lambda - OPT\ snapshot$ is proposed. An implementation of such an object provides a solution to the single-scanner snapshot problem and the multi-scanner snapshot problem simultaneously. If $\lambda$ is equal to 1, then a single-canner snapshot object is simulated by our algorithm. On the other hand if $\lambda$ is equal to the maximum number of processes then a multi-scanner snapshot object is simulated by our algorithm. Our, $\lambda - OPT\ snapshot$, object provides a new idea to solve the snapshot problem. To the best of our knowledge, there is no publication that provides a solution to the snapshot problem that can support a preset amount of $SCAN$ operation that may run concurrently.

A $\lambda - OPT\ snapshot$ can have many applications. It can be applied in systems where only a preset amount of processes may want to execute $SCAN$ operations. Especially in systems that the amount of processes that may want to invoke a $SCAN$ operation is small enough, our algorithm has almost the same performance as a single-scanner snapshot object. An example of such a system may be a sensor network, where many sensors are communicating with a small amount of monitor devices. In this case, sensors essentially perform $UPDATE$ operations while monitor devices may invoke $SCAN$ operations.

6.2. Future Work

In this work there is no experimental analysis of the algorithms that we presented. In a future work we would like to present some experimental analysis of $1 - OPT$ and $\lambda - OPT$. We would like to compare our algorithms to other state-of-the-art algorithms that provide an implementation of a snapshot object. Furthermore, we would like to provide a version of those two algorithms that use registers of finite size, in order to be more appealing for real system applications.

Another idea that we would like to explore in future work is to use our algorithm to simulate a graph object. Certain graph objects may be simulated by snapshot objects. The main idea is to store the components of the graph object in a snapshot object. Afterwards, processes that may want to modify a value of the graph object should invoke an $UPDATE$ operation, while the processes that want to traverse the graph should do so by invoking a modified $SCAN$ operation.

Finally, we would also like to provide a proof to the partial versions of $1 - OPT$ and $\lambda - OPT$ implementations that presented in Section 4.5 and 5.5 respectively.
7. Bibliography


