# ΣΧΕΔΙΑΣΗ ΚΑΙ ΑΝΑΛΥΣΗ ΑΤΟΜΙΚΩΝ ΑΝΤΙΚΕΙΜΕΝΩΝ

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Νικόλαο Καλλιμάνη

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΗΝ ΘΕΩΡΙΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Οκτώβριος 2007

# ΑΦΙΕΡΩΣΗ

Στην οικογένειά μου

# ΕΥΧΑΡΙΣΤΙΕΣ

Πρωτίστως θα ήθελα να ευχαριστήσω την επιβλέπουσα καθηγήτριά μου κα Παναγιώτα Φατούρου. Την ευχαριστώ ιδιαίτερα για την υπομονή που επέδειξε, για τις πολύτιμες συμβουλές που μου έδωσε, καθώς και για τον πολύτιμό χρόνο της που σπατάλησε παρέχοντάς μου αμέριστη στήριξη και βοήθεια στην προσπάθειά μου να ολοκληρώσω τη διατριβή μου. Επίσης θα ήθελα να ευχαριστήσω την οικογένειά μου για την ηθική και υλική υποστήριξη που μου παρείχε. Τέλος θα ήθελα να ευχαριστήσω όλους όσους με στήριξαν καθ' οιοδήποτε τρόπο καθ' όλη τη διάρκεια των σπουδών μου.

# ΠΕΡΙΕΧΟΜΕΝΑ

# ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ

# ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

# ΠΕΡΙΛΗΨΗ

Νικόλαος Καλλιμάνης του Δημητρίου και της Νικολέττας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Οκτώβριος, 2007. Σχεδίαση και ανάλυση ατομικών αντικειμένων. Επιβλέπουσα: Παναγιώτα Φατούρου.

Τα ατομικά στιγμιότυπα κοινής μνήμης είναι θεμελιώδη ατομικά αντικείμενα του κατανεμημένου υπολογισμού, τα οποία προσφέρουν συνεπείς όψεις της κοινής μνήμης ενός συστήματος. Ένα ατομικό στιγμιότυπο αποτελείται από $m$ συνιστώσες, όπου σε κάθε συνιστώσα αποθηκεύεται μία τιμή. Για την προσπέλαση και τροποποίηση των αποθηκευμένων δεδομένων, οι διεργασίες του συστήματος δύναται να εκτελούν δύο λειτουργίες, τη SCAN και την UPDATE. Η UPDATE ενημερώνει την τιμή μιας συνιστώσας, ενώ η SCAN επιστρέφει ένα 'συνεπές' διάνυσμα $m$ τιμών, μία τιμή για κάθε συνιστώσα. Τα ατομικά στιγμιότυπα έχουν πλήθος εφαρμογών του κατανεμημένου υπολογισμού και έχουν χρησιμοποιηθεί για την επίλυση πολύ σημαντικών προβλημάτων. Σε ένα Single-Scanner ατομικό στιγμιότυπο, σε κάθε χρονική στιγμή εκτελείται μία λειτουργία SCAN (ενώ δύναται πολλές UPDATE να εκτελούνται ταυτόχρονα).

Στην παρούσα εργασία παρουσιάζεται ένα σύνολο αποτελεσμάτων, στα οποία ανήκουν και οι πρώτες υλοποιήσεις Single-Scanner ατομικών στιγμιοτύπων από καταχωρητές ανάγνωσης εγγραφής με χρονική πολυπλοκότητα ανεξάρτητη του πλήθους των διεργασιών $n$. Επίσης παρουσιάζονται οι πρώτες χρονικά βέλτιστες τέτοιες υλοποιήσεις. Επιπρόσθετα οι υλοποιήσεις αυτές έχουν μικρή πολυπλοκότητα μνήμης, μικρό μέγεθος καταχωρητών και πολύ μικρές σταθερές στην πολυπλοκότητά τους, και έτσι έχουν όλα τα απαραίτητα χαρακτηριστικά για να εφαρμοστούν στην πράξη.

Οι αλγόριθμοι SweepLine και Linear που παρουσιάζονται είναι οι πρώτες υλοποιήσεις ατομικών στιγμιοτύπων που επιτυγχάνουν χρονική πολυπλοκότητα ανεξάρτητη του πλήθους των διεργασιών $n$. Ειδικότερα ο Linear επιτυγχάνει γραμμική πολυπλο-

κότητα στον αριθμό των συνιστωσών $m$ του στιγμιοτύπου (που μπορεί να είναι πολύ μικρότερος του $n$ ή ακόμα μπορεί να είναι και μία μικρή σταθερά), ενώ ο SweepLine επιτυγχάνει τετραγωνική πολυπλοκότητα του $m$. Επιπρόσθετα ο SweepLine χρησιμοποιεί μόνο $m + 1$ καταχωρητές ώστε να επιτύχει την καλή χρονική πολυπλοκότητά του. Στην παρούσα εργασία παρουσιάζονται δύο ακόμα αλγόριθμοι που ονομάζονται T-Opt και RT-Opt, οι οποίοι είναι οι πρώτες βέλτιστες υλοποιήσεις ατομικών στιγμιοτύπων, με χρονική πολυπλοκότητα για την UPDATE $O(1)$ και για τη SCAN $O(m)$. Ο T-Opt είναι ένας ιδιαίτερα αποτελεσματικός αλγόριθμος αν εφαρμοστεί σε συστήματα που υποστηρίζουν ανακύκλωση κοινής μνήμης (garbage collection), καθώς χρησιμοποιεί καταχωρητές μικρού μεγέθους και οι σταθερές των χρονικών πολυπλοκοτήτων του είναι ιδιαίτερα μικρές. Για τα συστήματα που δεν υποστηρίζουν ανακύκλωση κοινής μνήμης προτείνεται ο RT-Opt, ο οποίος επιτυγχάνει καλύτερη πολυπλοκότητα μνήμης δίχως να θυσιάζει την βέλτιστη χρονική πολυπλοκότητά του.

Για τη γενικότερη περίπτωση των ατομικών στιγμιοτύπων, όπου πολλές λειτουργίες SCAN δύναται να εκτελούνται ταυτόχρονα, παρουσιάζεται ο αλγόριθμος C-Snap. Ο C-Snap χρησιμοποιεί ισχυρότερους καταχωρητές από τους καταχωρητές ανάγνωσης-εγγραφής (CAS). Ο C-Snap είναι ο πρώτος αλγόριθμος που χρησιμοποιεί μόνο $O(m)$ κοινούς καταχωρητές, ενώ παράλληλα επιτυγχάνει πολυπλοκότητες $O(1)$ για την UPDATE και $O(m)$ για τη SCAN.

Είναι άξιο αναφοράς ότι οι αλγόριθμοι T-Opt, Linear και C-Snap δύναται να εφαρμοστούν σε συστήματα όπου τα αναγνωριστικά των διεργασιών (ids) δεν είναι διαθέσιμα (ανώνυμα συστήματα). Επίσης οι προαναφερθέντες αλγόριθμοι μπορούν να εφαρμοστούν με την ίδια αποτελεσματικότητα σε συστήματα όπου το πλήθος των διεργασιών είναι μη πεπερασμένο. Ακόμη κάποιοι από τους αλγορίθμους αποδεικνύουν ότι μερικά κάτω φράγματα που ισχύουν στη γενική περίπτωση δεν ισχύουν στην περίπτωση των Single-Scanner ατομικών στιγμιοτύπων.

# EXTENDED ABSTRACT IN ENGLISH

Kallimanis, Nikolaos, NK. MSc, Computer Science Department, University of Ioannina, Greece. September, 2007. Design and Implementation of Distributed, atomic objects. Thesis Supervisor: Panagiota Fatourou.

Snapshots are fundamental, atomic objects, which provide consistent views of blocks of shared memory. A snapshot object consists of $m$ components each storing a value from some set and supports two operations, UPDATE and SCAN, for accessing and modifying the stored data. An UPDATE operation writes a new value to some component, and a SCAN returns a consistent vector of $m$ values, one for each component. Snapshots have a lot of applications and have been used to simplify the solution of several fundamental problems of distributed computing. A Single-Scanner snapshot allows only one SCAN to be executed at each point in time (UPDATES can still be executed concurrently).

In this thesis, we present a collection of wait-free, linearizable, snapshot algorithms, including the first Single-Scanner snapshot implementations from read-write registers with time complexity independent of the number of processes $n$. We also present the first time-optimal such implementations. The aforementioned implementations have small space complexity, use small registers and have small constants in their time complexity; therefore they are of big practical interest.

The first two implementations we present, SweepLine and Linear are the first with time complexity that is independent of the number of processes $n$. More specifically, Linear achieves linear time complexity on the number $m$ of snapshot components (which in most applications is much smaller than $n$ while in others is just a small constant), while SweepLine achieves quadratic time complexity on $m$. Moreover, SweepLine uses only $m + 1$ shared registers in order to achieve its good time complexity. We also present the first two implementations, called T-Opt and RT-Opt, with

optimal time complexity. These implementations achieve time complexity $O(1)$ for UPDATE and $O(m)$ for SCAN. T-Opt is space-efficient in systems that support garbage collection. For systems that do not support garbage collection, we present RT-Opt, which significantly improves upon T-Opt in terms of space without sacrificing the optimal time complexity.

For the general case, where many SCANS can be executed concurrently, we present an implementation, called C-Snap which uses stronger than read-write registers (CAS registers). C-Snap is the first snapshot implementation to use only $O(m)$ registers, and achieve time complexity $O(1)$ for UPDATE and $O(m)$ for SCAN.

For T-Opt, Linear and C-Snap, processes do not need to have unique identifiers. Moreover, T-Opt and C-Snap work and are the same efficient even if the number of participating processes is infinite. Some of our implementations show that existing lower bounds on the complexity of snapshot implementations can be beaten if one restricts attention to the Single-Scanner case.

# ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

Ένα κατανεμημένο σύστημα αποτελείται από ένα σύνολο υπολογιστικών οντοτήτων (διεργασίες), οι οποίες έχουν τη δυνατότητα να επικοινωνούν μεταξύ τους. Ο παραπάνω ορισμός είναι αρκετά ευρύς ώστε να περιλαμβάνει συστήματα που απαρτίζονται από ένα σύνολο υπολογιστών και επικοινωνούν μέσω ενός δικτύου, συστήματα που αποτελούνται από κάποιον πολυεπεξεργαστή, καθώς και το διαδίκτυο.

Τα κατανεμημένα συστήματα χωρίζονται σε δύο βασικές κατηγορίες, ανάλογα με τον τρόπο με τον οποίο επικοινωνούν οι διεργασίες που τα απαρτίζουν. Στην πρώτη κατηγορία ανήκουν τα συστήματα των οποίων οι διεργασίες επικοινωνούν μέσω μίας κοινής μνήμης, όπου κάθε διεργασία έχει πρόσβαση σε αυτή (συστήματα κοινής μνήμης). Στη δεύτερη κατηγορία ανήκουν τα συστήματα, στα οποία οι διεργασίες που τα απαρτίζουν επικοινωνούν ανταλλάσσοντας μηνύματα (συστήματα ανταλλαγής μηνυμάτων).

Αρκετά μεγάλος όγκος έρευνας στα κατανεμημένα συστήματα έχει επιτελεσθεί σε συστήματα ανταλλαγής μηνυμάτων. Ωστόσο τα τελευταία χρόνια το ερευνητικό ενδιαφέρον έχει στραφεί σε συστήματα κοινής μνήμης. Είναι αξιοσημείωτο ότι οι νέες τεχνολογίες πολυπύρηνων επεξεργαστών έχουν δώσει μεγάλη ώθηση στα εν λόγω συστήματα. Σε λίγα χρόνια η πλειοψηφία των χρηστών θα διαθέτει ένα υπολογιστικό σύστημα που θα απαρτίζεται από έναν πολυπύρηνο επεξεργαστή. Ήδη έχουν κάνει την εμφάνισή τους εμπορικά, υπολογιστικά συστήματα χαμηλού κόστους που αποτελούνται από περισσότερους από δύο πυρήνες. Ένας άλλος παράγοντας που έχει δώσει ώθηση στα συστήματα κοινής μνήμης είναι ότι τα εν λόγω συστήματα θεωρούνται πιο εύκολο να προγραμματισθούν σε σχέση με τα συστήματα ανταλλαγής μηνυμάτων. Στην παρούσα εργασία εστιάζουμε μόνο σε συστήματα κοινής μνήμης.

Σε ένα κατανεμημένο σύστημα κοινής μνήμης οι διεργασίες του συστήματος επικοινωνούν προσπελάζοντας διάφορα, διαμοιραζόμενα ατομικά αντικείμενα κοινής μνήμης. Κάθε ατομικό αντικείμενο αποθηκεύει ένα σύνολο πληροφοριών, το οποίο είναι προσπελάσιμο από τις διεργασίες του συστήματος μέσω ατομικών λειτουργιών. Διαισθητικά ο όρος ατομική λειτουργία σημαίνει ότι σε κάθε χρονική στιγμή μόνο μία λειτουργία δύναται να είναι ενεργή (δηλαδή να εκτελείται από κάποια διεργασία)[1]. Ένα χαρακτηριστικό, αλλά και στοιχειώδες είδος ατομικών αντικειμένων είναι οι *καταχωρητές*. Ένας καταχωρητής αποθηκεύει μία τιμή από κάποιο σύνολο τιμών $T$, ενώ παρέχει κάποιες βασικές ατομικές λειτουργίες για την ανάγνωση και τροποποίηση των αποθηκευμένων δεδομένων του από τις διεργασίες. Το πιο συνηθισμένο ίσως είδος καταχωρητών είναι οι καταχωρητές *ανάγνωσης-εγγραφής* (read-write ή r/w). Οι καταχωρητές ανάγνωσης-εγγραφής υποστηρίζουν δύο λειτουργίες για την προσπέλαση των αποθηκευμένων δεδομένων: α) τη λειτουργία $read(R)$, η οποία επιστρέφει την τιμή που είναι αποθηκευμένη στον καταχωρητή $R$ δίχως να τροποποιήσει τα δεδομένα του, και β) τη λειτουργία $write(R, v)$ με την οποία εγγράφεται η τιμή $v$ στον καταχωρητή $R$ και επιστρέφεται μία επιβεβαίωση τερματισμού (acknowledgment). Υπάρχουν και άλλα είδη καταχωρητών που προσφέρουν πιο πολύπλοκες, ατομικές λειτουργίες προσπέλασης των αποθηκευμένων δεδομένων. Χαρακτηριστικό παράδειγμα τέτοιων καταχωρητών είναι οι καταχωρητές CAS (Compare-And-Swap). Ένας καταχωρητής $CAS$ $R'$ υποστηρίζει την λειτουργία $read(R')$ και τη λειτουργία $CAS(R', v_{old}, v_{new})$. Η λειτουργία $read(R')$ επιστρέφει την τιμή που είναι αποθηκευμένη στον $R'$ δίχως να τροποποιήσει τα δεδομένα του. Η λειτουργία $CAS(R', v_{old}, v_{new})$ δέχεται ως παράμετρο τις τιμές $v_{new}, v_{old}$, και αλλάζει την τιμή του καταχωρητή $R'$ σε $v_{new}$ μόνο αν η τρέχουσα τιμή που είναι αποθηκευμένη στον $R'$ είναι ίση με $v_{old}$. Αν η τιμή που είναι αποθηκευμένη στον $R'$ είναι διάφορη της $v_{old}$, η λειτουργία $CAS(R', v_{old}, v_{new})$ δε μεταβάλλει τα περιεχόμενα του καταχωρητή $R'$. Η λειτουργία $CAS(R', v_{old}, v_{new})$ επιστρέφει απλά μία επιβεβαίωση τερματι-

[1] Στο σύστημα είναι δυνατό να εκτελούνται ατομικές λειτουργίες σε άλλα αντικείμενα ή ατομικοί υπολογισμοί από άλλες διεργασίες ταυτόχρονα. Φορμαλιστικός ορισμός για τα πολυδιεργασιακά συστήματα θα δοθεί στο Κεφάλαιο 3.

σμού. Είναι αξιοσημείωτο ότι η λειτουργία CAS είναι πιο σύνθετη λειτουργία από τη λειτουργία write, αφού εμπεριέχει την ατομική εκτέλεση τόσο μιας λειτουργίας read όσο και μιας λειτουργίας write. Συνήθως το υλικό (hardware) του συστήματος εγγυάται την ατομικότητα των λειτουργιών των καταχωρητών.

Τα ατομικά αντικείμενα που προσφέρει το υλικό ενός κατανεμημένου συστήματος (π.χ. καταχωρητές) παρέχουν στοιχειώδεις λειτουργίες προσπέλασης των δεδομένων που είναι αποθηκευμένα σε αυτά. Η επίλυση διαφόρων προβλημάτων κατανεμημένου υπολογισμού χρησιμοποιώντας μόνο τα στοιχειώδη ατομικά αντικείμενα είναι αρκετά δύσκολη. Επομένως γεννάται η ανάγκη να αναπτυχθούν νέα, πιο σύνθετα ατομικά αντικείμενα, τα οποία θα χρησιμοποιούν ως θεμέλιους λίθους τα στοιχειώδη ατομικά αντικείμενα και θα παρέχουν πιο πολύπλοκες λειτουργίες επιτρέποντας την ευκολότερη επίλυση θεμελιωδών προβλημάτων του κατανεμημένου υπολογισμού. Έτσι μπορεί να δημιουργηθεί μία διαβάθμιση των ατομικών αντικειμένων, όπου η χαμηλότερη βαθμίδα θα αποτελείται από τα στοιχειώδη ατομικά αντικείμενα που προσφέρει το υλικό του συστήματος, ενώ οι υψηλότερες βαθμίδες θα αποτελούνται από αντικείμενα που θα παρέχουν πιο πολύπλοκες λειτουργίες και θα υλοποιούνται κάνοντας χρήση ατομικών αντικειμένων χαμηλότερων βαθμίδων. Μια υλοποίηση ενός σύνθετου ατομικού αντικειμένου πρέπει να χρησιμοποιεί τα στοιχειώδη ατομικά αντικείμενα για την αποθήκευση των δεδομένων και να παρέχει τους κατάλληλους αλγορίθμους για την υλοποίηση των λειτουργιών του αντικειμένου.

Ένα σύνθετο ατομικό αντικείμενο, το οποίο βρίσκει εφαρμογές σε ένα πλήθος προβλημάτων του κατανεμημένου υπολογισμού και θα μπορούσε να χρησιμοποιηθεί ως θεμέλιος λίθος για την ευκολότερη επίλυση προβλημάτων, είναι το *ατομικό στιγμιότυπο κοινής μνήμης* (atomic snapshot) [1], [4], [7]. Ένα ατομικό στιγμιότυπο αποτελείται από $m$ *συνιστώσες* (components), όπου σε κάθε συνιστώσα δύναται να αποθηκευθεί μία τιμή. Για την προσπέλαση και τροποποίηση των δεδομένων του, ένα ατομικό στιγμιότυπο υποστηρίζει δύο λειτουργίες, την $UPDATE(i, v)$, $1 \leq i \leq m$ και τη $SCAN()$. Η λειτουργία $UPDATE(i, v)$ ενημερώνει την $i$-οστή συνιστώσα του αντικειμένου με την τιμή $v$. Η λειτουργία SCAN επιστρέφει ένα *συνεπές διάνυσμα* $m$ τιμών, μία τιμή για κάθε συνιστώσα του ατομικού στιγμιότυπου. Διαισθητικά με τον όρο συνεπές διάνυσμα τιμών, εννοούμε ότι οι τιμές των συνιστωσών του επιστρεφό-

μενου διανύσματος συνυπήρξαν σε κάποια χρονική στιγμή στις συνιστώσες του ατομικού αντικειμένου (η έννοια της συνέπειας θα περιγραφεί αναλυτικά στο Κεφάλαιο 3). Αξίζει να σημειωθεί ότι η SCAN δε μεταβάλει κάποια από τις τιμές των $m$ συνιστωσών, ενώ η UPDATE επιστρέφει απλά μία επιβεβαίωση τερματισμού (acknowledgment). Οποιαδήποτε διεργασία του συστήματος δύναται να εκτελεί λειτουργίες SCAN ή UPDATE. Διαισθητικά, ένα ατομικό στιγμιότυπο μπορεί να μοντελοποιήσει την κοινή μνήμη του συστήματος (ή τις κοινές μεταβλητές που χρησιμοποιεί ένας κατανεμημένος αλγόριθμος) παρέχοντας όμως στις διεργασίες εκτός από τη δυνατότητα να εγγράφουν οποιαδήποτε θέση μνήμης και τη δυνατότητα να παίρνουν μία συνεπή όψη όλης της μνήμης μέσω της ισχυρής λειτουργίας SCAN.

Χρησιμοποιώντας αντικείμενα ατομικών στιγμιοτύπων είναι δυνατή η παρακολούθηση της καθολικής κατάστασης (state) ενός συστήματος, η οποία περιγράφει το σύστημα σε κάποια χρονική στιγμή. Έτσι διευκολύνεται η επίλυση διάφορων, σημαντικών προβλημάτων του κατανεμημένου υπολογισμού, όπως η ανίχνευση τερματισμού σε κατανεμημένους αλγορίθμους (termination detection) [31], η ανίχνευση αδιεξόδων (deadlock detection) [31], η ανακύκλωση κοινής μνήμης (garbage collection) [31], [26], η επίτευξη εξισορρόπησης φορτίου (load balancing) [31]. Επίσης τα ατομικά στιγμιότυπα έχουν χρησιμοποιηθεί για την επίλυση και την επαλήθευση προβλημάτων, όπως η δημιουργία παράλληλων, εφεδρικών αντιγράφων ασφαλείας (backups) [3], η δημιουργία σημείων ελέγχου (checkpointing) [31], η αποσφαλμάτωση παράλληλων εφαρμογών (debugging) [13], η ανάθεση χρονοσφραγίδων (timestamps) [21], η επίτευξη ομοφωνίας με χρήση τυχαιότητας (randomized consensus) [6], και η επίτευξη κάποιων πιο ειδικών μορφών ομοφωνίας (approximate agreement) [11]. Ακόμη τα ατομικά στιγμιότυπα έχουν αποτελέσει τον θεμέλιο λίθο για την κατασκευή διαφόρων κατανεμημένων δομών δεδομένων [7] και έχουν χρησιμοποιηθεί για την εύκολη επίλυση του προβλήματος του αμοιβαίου αποκλεισμού [13].

Η αποτελεσματική υλοποίηση ενός αντικειμένου ατομικών στιγμιοτύπων είναι αρκετά δύσκολη υπόθεση. Αρχικά αναπτύχθηκαν αντικείμενα ατομικών στιγμιοτύπων, των οποίων κάθε συνιστώσα ενημερώνεται από μία μόνο διεργασία (ενώ οποιαδήποτε διεργασία δύναται να επιτελεί λειτουργίες SCAN). Αυτού του είδους τα ατομικά στιγμιότυπα ονομάζονται ατομικά στιγμιότυπα *απλής εγγραφής* (Single-Writer ή SW)

[4], [7], [11], [9], [12], [14], (32). Τα τελευταία χρόνια το ερευνητικό ενδιαφέρον έχει εστιαστεί στη γενικότερη κατηγορία ατομικών στιγμιοτύπων, στην οποία οποιαδήποτε διεργασία δύναται να επιτελεί UPDATE σε οποιαδήποτε συνιστώσα του αντικειμένου. Αυτού του είδους τα ατομικά στιγμιότυπα ονομάζονται ατομικά στιγμιότυπα *πολλαπλής εγγραφής* (Multi-Writer ή MW) [26], [5], [8], [15], [25], [29]. Στην παρούσα εργασία θα μελετηθούν ατομικά στιγμιότυπα πολλαπλής εγγραφής.

Τα ατομικά στιγμιότυπα θα μπορούσαν να υλοποιηθούν χρησιμοποιώντας κλειδώματα (locks). Μία διεργασία που επιθυμεί να επιτελέσει μία SCAN, «κλειδώνει» την περιοχή κοινής μνήμης, όπου επιθυμεί να πάρει ένα ατομικό στιγμιότυπο, αποτρέποντας τις υπόλοιπες διεργασίες να εκτελούν ταυτόχρονα UPDATE, έως ότου ολοκληρωθεί η SCAN. Η συγκεκριμένη τεχνική έχει εφαρμοστεί ευρέως σε συστήματα βάσεων δεδομένων. Όμως η τεχνική αυτή έχει αρκετά μειονεκτήματα καθώς μία διεργασία είναι δυνατό να καταρρεύσει ενώ εκτελεί μία SCAN. Σε μία τέτοια περίπτωση όλο το σύστημα θα οδηγηθεί σε συνολική κατάρρευση αφού το κλείδωμα θα παραμείνει ενεργό για απεριόριστο χρονικό διάστημα αποτρέποντας τις υπόλοιπες διεργασίες να προσπελάζουν το αντικείμενο. Επίσης η μη προσεκτική χρήση κλειδωμάτων είναι δυνατό να οδηγήσει το σύστημα σε αδιέξοδο, καθώς και να μειώσει την απόδοση του συστήματος.

Δεδομένου ότι οι διεργασίες σε ένα κατανεμημένο σύστημα είναι δυνατό να καταρρέουν, δηλαδή είναι δυνατό να σταματούν την εκτέλεσή τους σε οποιαδήποτε χρονική στιγμή, είναι σημαντικό το σύστημα να έχει *ανοχή σε σφάλματα* (fault tolerance). Μία ιδιότητα που εγγυάται υψηλού βαθμού ανθεκτικότητα σε σφάλματα είναι η ιδιότητα *ελεύθερη-αναμονής* (wait-freedom) [7], [11]. Μία υλοποίηση ενός ατομικού αντικειμένου πληροί την ιδιότητα ελεύθερη-αναμονής, αν κάθε μια μη αποτυχημένη διεργασία ολοκληρώνει την εκτέλεση οποιασδήποτε λειτουργίας του αντικειμένου μέσα σε ένα πεπερασμένο αριθμό υπολογιστικών βημάτων, ανεξάρτητα από την ταχύτητα, αλλά και την κατάσταση στην οποία βρίσκονται οι υπόλοιπες διεργασίες του συστήματος.

Παρότι τα ατομικά στιγμιότυπα έχουν πάρα πολλές εφαρμογές, η πραγματική τους αξία εξαρτάται από το αν μπορούν να υλοποιηθούν με αποτελεσματικό τρόπο. Η αποτελεσματικότητα μιας υλοποίησης ατομικών στιγμιοτύπων εξαρτάται κατά κύριο

λόγο από τη χρονική πολυπλοκότητα των λειτουργιών SCAN και UPDATE, ενώ είναι σημαντικός ο αριθμός και το μέγεθος των καταχωρητών που χρησιμοποιεί η υλοποίηση. Η χρονική πολυπλοκότητα της SCAN (ή UPDATE) είναι ο μέγιστος αριθμός υπολογιστικών βημάτων που επιτελείται από οποιαδήποτε διεργασία για την εκτέλεση μιας SCAN (ή UPDATE) σε οποιαδήποτε εκτέλεση.

Μια ιδεώδης υλοποίηση ατομικών στιγμιοτύπων θα είχε το σύνολο των έξι χαρακτηριστικών που περιγράφονται στη συνέχεια.

1) Η χρονική πολυπλοκότητα της SCAN θα ήταν μικρή (αν είναι δυνατόν ανεξάρτητη από το πλήθος των διεργασιών $n$ του συστήματος, που είναι συνήθως αρκετά μεγαλύτερο από το πλήθος των συνιστωσών $m$).
2) Η επιτέλεση μιας UPDATE δε θα κόστιζε σε χρόνο πολύ περισσότερο από την εκτέλεση μιας εντολής write. Έτσι είναι επιθυμητό, η χρονική πολυπλοκότητα της UPDATE να είναι σταθερή ($O(1)$).
3) Θα χρησιμοποιούσε καταχωρητές ανάγνωσης-εγγραφής, αφού πολλά συστήματα δεν υποστηρίζουν πιο ισχυρούς καταχωρητές.
4) Θα χρησιμοποιούσε όσο το δυνατό μικρότερο αριθμό καταχωρητών.
5) Οι απαιτούμενοι καταχωρητές θα είχαν μικρό μέγεθος ισοδύναμο με το μέγεθος των καταχωρητών που προσφέρει το υλικό του συστήματος ($\leq$128 bit).
6) Θα ήταν ανθεκτική σε σφάλματα, οπότε και θα πληρούσε την ιδιότητα ελεύθερη-αναμονής.

Δυστυχώς, η σχεδίαση ατομικών στιγμιοτύπων που πληρούν ακόμη και ένα υποσύνολο των παραπάνω ιδιοτήτων φαίνεται πως είναι ένα εξαιρετικά δύσκολο πρόβλημα. Η αποτελεσματικότερη ελεύθερη-αναμονής υλοποίηση ατομικών στιγμιοτύπων πολλαπλής εγγραφής που χρησιμοποιεί καταχωρητές ανάγνωσης-εγγραφής έχει παρουσιαστεί στο [9] και επιτυγχάνει χρονική πολυπλοκότητα $O(n)$ για τη SCAN και την UPDATE, όπου $n$ είναι ο αριθμός των διεργασιών του συστήματος. Γενικά όλες οι υλοποιήσεις που είχαν παρουσιασθεί ως τώρα και χρησιμοποιούν καταχωρητές ανάγνωσης-εγγραφής, επιτυγχάνουν πολυπλοκότητα που είναι μία συνάρτηση του $n$.

Στο [17] έχει αποδειχθεί ότι σε κάθε υλοποίηση που χρησιμοποιεί έναν καθορισμένο (fixed) αριθμό καταχωρητών, η πολυπλοκότητα της SCAN είναι μία αύξουσα συνάρ-

τηση του αριθμού των διεργασιών $n$ στο σύστημα. Στην παρούσα εργασία μελετήθηκε αν το άνω φράγμα του [17] ισχύει, αν επικεντρωθούμε σε μια κατηγορία ατομικών στιγμιοτύπων που ονομάζεται Single-Scanner (SS). Σε ένα Single-Scanner ατομικό στιγμιότυπο σε κάθε χρονική στιγμή εκτελείται μία λειτουργία SCAN (όμως πολλές UPDATE δύναται να εκτελούνται ταυτόχρονα). Τα Single-Scanner ατομικά στιγμιότυπα έχουν πολλές εφαρμογές (π.χ. δημιουργία εφεδρικών αντιγράφων ασφαλείας, ανακύκλωση κοινής μνήμης, απόδοση χρονοσφραγίδων κ.α.), οι οποίες προσδίδουν αρκετό ενδιαφέρον στην μελέτη τους.

Σε όλες τις υλοποιήσεις Single-Scanner ατομικών στιγμιοτύπων που έχουν παρουσιασθεί, η SCAN έχει χρονική πολυπλοκότητα που είναι μία συνάρτηση του $n$ [25], [29], (32). Ειδικότερα η καλύτερη υλοποίηση Single-Scanner ατομικών στιγμιοτύπων έχει παρουσιασθεί στο [25] και επιτυγχάνει χρονική πολυπλοκότητα για τη SCAN $O(n)$.

Στην παρούσα εργασία παρουσιάζουμε τις πρώτες υλοποιήσεις Single-Scanner ατομικών στιγμιοτύπων των οποίων η χρονική πολυπλοκότητα είναι συνάρτηση μόνο του πλήθους των συνιστωσών $m$ και όχι του πλήθους των διεργασιών $n$. Έτσι αποδεικνύεται ότι το κάτω φράγμα που παρουσιάζεται στο [17] δεν ισχύει στα Single-Scanner ατομικά στιγμιότυπα.

Η πρώτη εκ των υλοποιήσεων που επιτυγχάνει χρονική πολυπλοκότητα ανεξάρτητη του $n$, είναι ο αλγόριθμος Linear. Ο Linear επιτυγχάνει χρονική πολυπλοκότητα $O(m)$ για τη SCAN και την UPDATE. Ο Linear είναι ο πρώτος αλγόριθμος που παρουσιάστηκε και επιτυγχάνει πολυπλοκότητα για τη SCAN και την UPDATE ανεξάρτητη του πλήθους των διεργασιών του συστήματος $n$.

Αν και ο αλγόριθμος Linear απαντά στο ερώτημα της ύπαρξης ενός αλγορίθμου που θα επιτυγχάνει πολυπλοκότητα για τη SCAN ανεξάρτητη του αριθμού των διεργασιών του συστήματος, δημιούργησε το ερώτημα αν είναι χρονικά βέλτιστος. Στο [28] παρουσιάζεται ένα κάτω φράγμα για ατομικά στιγμιότυπα, με το οποίο αποδεικνύεται ότι η πολυπλοκότητα της λειτουργίας SCAN ενός ατομικού στιγμιοτύπου που χρησιμοποιεί καταχωρητές ανάγνωσης εγγραφής είναι $\Omega(m)$. Έτσι συμπεραίνουμε ότι η πολυπλοκότητα της SCAN του Linear είναι βέλτιστη.

Ωστόσο όπως αναφέρθηκε, η επιτέλεση μιας UPDATE δε πρέπει να κοστίζει σε χρόνο πολύ περισσότερο από την εκτέλεση μιας εντολής write (χαρακτηριστικό 2 του ιδεώδους αλγορίθμου). Έτσι είναι επιθυμητό, η χρονική πολυπλοκότητα της UPDATE να είναι σταθερή ($O(1)$). Οπότε η σχεδίαση ενός αλγορίθμου με μικρότερη χρονική πολυπλοκότητα για την UPDATE είναι πολύ σημαντική. Αυτό επιτυγχάνεται από τον δεύτερο αλγόριθμο που παρουσιάζεται, ο οποίος ονομάζεται T-Opt. Ο T-Opt είναι ο πρώτος αλγόριθμος που επιτυγχάνει χρονική πολυπλοκότητα για την UPDATE $O(1)$ και για τη SCAN $O(m)$, οι οποίες και είναι βέλτιστες. Ο T-Opt βελτιώνει τον Linear ως προς την πολυπλοκότητα της UPDATE, και έτσι πληροί τα χαρακτηριστικά (1) και (2) του ιδεώδους αλγορίθμου που περιγράφηκαν νωρίτερα. Επιπρόσθετα ο T-Opt βελτιώνει τον Linear ως προς το μέγεθος των καταχωρητών που απαιτεί. Ο Linear χρησιμοποιεί καταχωρητές μεγέθους $O(\max\{m \log|T|, \log k\})$ bit (με $T$ συμβολίζουμε το σύνολο τιμών που αποθηκεύονται σε κάθε συνιστώσα ενός ατομικού στιγμιοτύπου), ενώ ο T-Opt χρησιμοποιεί καταχωρητές μεγέθους $O(\max\{\log|T|, \log k\})$ bit. Αξίζει να σημειωθεί ότι ο T-Opt είναι ιδιαίτερα αποτελεσματικός και πρακτικά εφαρμόσιμος αν χρησιμοποιηθεί σε συστήματα που διαθέτουν *ανακυκλωτή κοινής μνήμης* (garbage collector), καθώς σε αυτή την περίπτωση χρησιμοποιείται περιορισμένος αριθμός καταχωρητών. Έτσι αν ο T-Opt εφαρμοστεί σε συστήματα που διαθέτουν ανακυκλωτή κοινής μνήμης, πληροί όλα τα χαρακτηριστικά ενός ιδεώδους αλγορίθμου.

Σε συστήματα που δεν παρέχουν τη δυνατότητα ανακύκλωσης κοινής μνήμης, το πλήθος των καταχωρητών που χρησιμοποιεί ο T-Opt είναι ανάλογο του πλήθους των SCAN $k$ που εκτελούνται στο σύστημα, το οποίο μπορεί να είναι μη πεπερασμένο. Αυτό καθιστά την χρήση του T-Opt σε συστήματα που δεν υποστηρίζουν ανακύκλωση κοινής μνήμης να μην είναι ιδιαίτερα πρακτική. Έτσι ο επόμενος στόχος της παρούσας εργασίας ήταν να αναπτυχθεί ένας αλγόριθμος που θα επιτυγχάνει τις πολυπλοκότητες του T-Opt, αλλά και θα χρησιμοποιεί περιορισμένο αριθμό καταχωρητών σε συστήματα που δεν υποστηρίζουν ανακύκλωση κοινής μνήμης. Αυτός ο στόχος ικανοποιείται με την παρουσίαση των αλγορίθμων RT και RT-Opt. Με τον RT αλγόριθμο γίνεται μία πρώτη προσπάθεια ώστε να περιορισθεί ο αριθμός των καταχωρητών που χρησιμοποιεί ο T-Opt σε συστήματα όπου δεν είναι διαθέσιμος κάποιος ανακυκλωτής καταχωρητών. Ο αλγόριθμος RT επιτυγχάνει να χρησιμοποιεί μόνο $O(mn)$

καταχωρητές (δηλαδή πεπερασμένο πλήθος καταχωρητών) και να έχει πολυπλοκότητα για τη SCAN $O(n)$ και για την UPDATE $O(1)$. Ο RT χρησιμοποιεί καταχωρητές ιδιαίτερα μικρού μεγέθους. Ειδικότερα όλοι οι καταχωρητές αποθηκεύουν μόνο την τιμή της συνιστώσας του στιγμιότυπου, εκτός από έναν που αποτελείται από $O(\log n)$ bit.

Ο RT διαθέτει όλα τα χαρακτηριστικά ενός ιδεώδους αλγορίθμου εκτός από το πρώτο (που είναι όμως αρκετά σημαντικό). Και το χαρακτηριστικό αυτό επιτυγχάνεται από τον αλγόριθμο RT-Opt. Ο αλγόριθμος RT-Opt επιτυγχάνει να συνδυάσει τη βέλτιστη χρονική πολυπλοκότητα του T-Opt με την καλή χωρική πολυπλοκότητα του RT. Για να επιτευχθεί αυτό, εφαρμόζεται μια πολυπλοκότερη και αποτελεσματικότερη τεχνική ανακύκλωσης από αυτή του RT. Ο RT-Opt χρησιμοποιεί μόνο $O(mn)$ καταχωρητές και επιτυγχάνει χρονική πολυπλοκότητα για τη SCAN $O(m)$ και για την UPDATE $O(1)$. Το μέγεθος των καταχωρητών του RT-Opt είναι ίδιο με το μέγεθος των καταχωρητών του RT. Με τον αλγόριθμο RT-Opt επιτυγχάνεται η σχεδίαση ενός αλγορίθμου που πληροί όλα τα χαρακτηριστικά του ιδεώδους αλγορίθμου που προαναφέρθηκαν. Επιπρόσθετα οι σταθερές των χρονικών πολυπλοκοτήτων της UPDATE και της SCAN είναι αρκετά μικρές, κάτι που καθιστά τον RT-Opt έναν ιδιαίτερα αποτελεσματικό και πρακτικό αλγόριθμο. Επίσης αξίζει να σημειωθεί ότι η τεχνική ανακύκλωσης κοινών καταχωρητών που χρησιμοποιεί ο εν λόγω αλγόριθμος, φαίνεται πως ίσως μπορεί να εφαρμοστεί και σε άλλα προβλήματα του κατανεμημένου υπολογισμού, βελτιώνοντας την απόδοσή τους.

Όλοι οι αλγόριθμοι που παρουσιάστηκαν παραπάνω (εκτός του RT) επιτυγχάνουν χρονική πολυπλοκότητα ανεξάρτητη του $n$. Όμως η χωρική τους πολυπλοκότητα στην καλύτερη περίπτωση είναι μια συνάρτηση του $n$. Το εύλογο ερώτημα που τίθεται είναι αν μπορεί να αναπτυχθεί μια Single-Scanner υλοποίηση που θα επιτυγχάνει χωρική πολυπλοκότητα ανεξάρτητη του αριθμού των διεργασιών του συστήματος, ακόμα και αν το μέγεθος των καταχωρητών που χρησιμοποιείται είναι αρκετά μεγάλο. Με την παρουσίαση του αλγορίθμου SweepLine, η απάντηση στο προαναφερθέν ερώτημα είναι καταφατική. Ο SweepLine χρησιμοποιεί $m + 1$ καταχωρητές για να επιτύχει χρονική πολυπλοκότητα $O(m^2)$ για τη SCAN και την UPDATE. Στο [18] έχει αποδειχθεί ότι η SCAN σε οποιαδήποτε υλοποίηση Single-Scanner ατομικών

στιγμιοτύπων που χρησιμοποιεί $m$ καταχωρητές ανάγνωσης-εγγραφής, έχει χρονική πολυπλοκότητα $\Omega(m^2)$. Έτσι ο SweepLine επιτυγχάνει την παραπάνω χρονική πολυπλοκότητα χρησιμοποιώντας μόνο έναν παραπάνω καταχωρητή.

Αν και η Single-Scanner περίπτωση των ατομικών στιγμιοτύπων είναι ιδιαίτερα χρήσιμη, πολλές φορές είναι απαραίτητη η χρήση ατομικών στιγμιοτύπων που υποστηρίζουν την ταυτόχρονη εκτέλεση SCAN (Multi-Scanner ή MS). Όμως όπως προαναφέρθηκε δε είναι δυνατό να υπάρξει μία αποτελεσματική υλοποίηση ατομικών στιγμιοτύπων που να χρησιμοποιεί καταχωρητές ανάγνωσης-εγγραφής, όπως και αποδεικνύεται στο [17]. Η κατάσταση μπορεί να βελτιωθεί αν χρησιμοποιηθούν καταχωρητές που υποστηρίζουν πιο πολύπλοκες λειτουργίες (π.χ. CAS) [26], (32), [25]. Ο πιο αποτελεσματικός αλγόριθμος τέτοιου είδους έχει παρουσιαστεί στο [25], ο οποίος επιτυγχάνει πολυπλοκότητα $O(m)$ για τη SCAN και $O(1)$ για την UPDATE χρησιμοποιώντας $O(mn)$ καταχωρητές CAS. Με την ανάπτυξη του αλγορίθμου C-Snap, επιτυγχάνονται οι ίδιες χρονικές πολυπλοκότητες με αυτές του αλγορίθμου που παρουσιάζεται στο [25], ενώ παράλληλα μειώνεται σημαντικά η χωρική πολυπλοκότητα σε $O(m)$. Ο C-Snap βασίζεται ουσιαστικά στον T-Opt και με τη χρήση εντολών CAS επιτυγχάνει να κάνει ανακύκλωση καταχωρητών, ενώ παράλληλα επιτρέπει σε πολλές διεργασίες να εκτελούν SCAN. Ο C-Snap είναι ο πρώτος αλγόριθμος που παρουσιάστηκε που έχει τις προαναφερθείσες χρονικές πολυπλοκότητες και η πολυπλοκότητα μνήμης είναι ανεξάρτητη του αριθμού των διεργασιών $n$ του συστήματος. Η προαναφερθείσα ιδιότητα δίνει τη δυνατότητα να χρησιμοποιηθεί ο C-Snap ακόμα και σε συστήματα όπου το πλήθος των διεργασιών που εκτελούνται είναι άπειρο. Το μόνο μειονέκτημα του C-Snap είναι ότι το μέγεθος ενός εκ των καταχωρητών που χρησιμοποιεί είναι αρκετά μεγάλο.

Σε πολλά συστήματα, οι διεργασίες δεν έχουν αναγνωριστικά (ανώνυμα συστήματα). Αυτό είναι ένα σύνηθες φαινόμενο σε συστήματα, όπου προέχει η ασφάλεια και η ανωνυμία των χρηστών (π.χ. συστήματα p2p). Το ερώτημα που τίθεται είναι αν μπορεί να υλοποιηθεί ένα ατομικό στιγμιότυπο χωρίς τη χρήση αναγνωριστικών. Την απάντηση σε αυτό το ερώτημα τη δίνουν οι αλγόριθμοι Linear, T-opt και C-Snap. Οι αλγόριθμοι Linear, T-opt και C-Snap δεν κάνουν χρήση αναγνωριστικών (ids) των διεργασιών του συστήματος. Επιπρόσθετα οι εν λόγω αλγόριθμοι δύναται να λει-

τουργήσουν ακόμα και σε συστήματα όπου το πλήθος των διεργασιών είναι άπειρο. Στον Πίνακα 1 παρουσιάζονται τα χαρακτηριστικά των αλγορίθμων που αναπτύχθηκαν. Οι αλγόριθμοι Linear και SweepLine έχουν παρουσιασθεί στο [20], ενώ οι αλγόριθμοι T-Opt, RT, RT-Opt και C-Snap έχουν παρουσιασθεί στα [18], [19].

Η διάρθρωση της παρούσας εργασίας είναι η ακόλουθη. Στο Κεφάλαιο 2 παρουσιάζονται συνοπτικά οι σχετικές εργασίες που έχουν γίνει. Στο Κεφάλαιο 3 παρουσιάζεται το φορμαλιστικό μοντέλο του συστήματος που θα χρησιμοποιηθεί. Στα κεφάλαια 4-9 παρουσιάζονται οι αλγόριθμοι Linear, T-Opt, RT, RT-Opt, C-Snap, SweepLine με τη σειρά που αναφέρθηκαν.

Πίνακας 1 Αλγόριθμοι που αναπτύχθηκαν στην παρούσα εργασία[2].

| Αλγόριθμος | UPDATE | SCAN | Αριθμός καταχωρητών | Μέγεθος καταχωρητών (bit) |
|---|---|---|---|---|
| Linear, SS | $O(m)$ | $O(m)$ | $O(m+k)$ | $O(\max\{m\log\|T\|, \log k\})$ |
| T-Opt, SS | $O(1)$ | $O(m)$ | $O(mk)$ | $O(\max\{\log k, \log\|T\|\})$ |
| RT, SS | $O(1)$ | $O(n)$ | $O(mn)$ | $O(\max\{\log n, \log\|T\|\})$ |
| RT-Opt, SS | $O(1)$ | $O(m)$ | $O(mn)$ | $O(\max\{\log n, \log\|T\|\})$ |
| SweepLine, SS | $O(m^2)$ | $O(m^2)$ | $m+1$ | $O(m\log\|T\| + \log k + \log n)$ |
| C-Snap | $O(1)$ | $O(m)$ | $2m+1$ | $O(\log k + m\log\|T\|)$ |

[2] Με $k$ συμβολίζουμε τον μέγιστο αριθμό SCAN που εκτελούνται σε μία εκτέλεση και $T$ το σύνολο τιμών που αποθηκεύονται σε κάθε συνιστώσα ενός ατομικού στιγμιότυπου.

# ΚΕΦΑΛΑΙΟ 2. ΣΧΕΤΙΚΕΣ ΕΡΓΑΣΙΕΣ

Τα τελευταία χρόνια πλήθος υλοποιήσεων MS/MW ατομικών στιγμιοτύπων από καταχωρητές ανάγνωσης-εγγραφής έχει δημοσιευθεί (Πίνακας 2). Κοινή συνισταμένη όλων των προαναφερθεισών υλοποιήσεων είναι ότι η πολυπλοκότητα των λειτουργιών τους είναι εξαρτημένη από το πλήθος των διεργασιών του συστήματος $n$. Ειδικότερα η καλύτερη υλοποίηση ατομικών στιγμιοτύπων έχει παρουσιαστεί στο [9] και επιτυγχάνει χρονική πολυπλοκότητα για τη SCAN και την UPDATE $O(n)$ χρησιμοποιώντας $O(n^2)$ κοινούς καταχωρητές.

Πίνακας 2 Υλοποιήσεις MW ατομικών στιγμιοτύπων από καταχωρητές ανάγνωσης-εγγραφής.

| Υλοποίηση | Αριθμός καταχωρητών | SCAN | UPDATE |
|---|---|---|---|
| Afek et al. [1] | O($n^2$) | $O(n^2 + mn)$ | $O(n^2 + mn)$ |
| Anderson [4] | $O(m^3 n^4)$ | $O(2^{mn})$ | $O(2^{mn})$ |
| Fatourou Fich & Ruppert [15] | $m$ | $O(mn)$ | $O(mn)$ |
| Συνδυασμός των [12], [5], [24] | $O(n^5)$ | $O(n)$ | $O(n \log n)$ |
| Συνδυασμός των [5], [24], [23] | $\infty$ | $O(n)$ | $O(n)$ |
| Attiya & Fouren [9] | $O(n^2)$ | $O(n)$ | $O(n)$ |

Όλες οι υλοποιήσεις Single-Scanner ατομικών στιγμιοτύπων που έχουν παρουσιασθεί στο παρελθόν έχουν χρονική πολυπλοκότητα για τη SCAN τουλάχιστον $\Theta(n)$ (Πίνακας 3). Βέβαια η πλειοψηφία αυτών των αλγορίθμων επιτυγχάνει σταθερή χρονική πολυπλοκότητα για την UPDATE. Οι αλγόριθμοι που παρουσιάστηκαν στο [18] είναι οι πρώτοι που έχουν χρονική πολυπλοκότητα ανεξάρτητη του αριθμού των διεργασιών $n$, ενώ οι αλγόριθμοι T-Opt και RT-Opt [20] είναι οι πρώτοι που επιτυγχάνουν βέλτιστη χρονική πολυπλοκότητα για τη SCAN και την UPDATE. Αξίζει να σημειω-

θεί ότι ο Linear [18] είναι ο πρώτος αλγόριθμος που επιτυγχάνει χρονική και χωρική πολυπλοκότητα ανεξάρτητη του πλήθους των διεργασιών $n$.

Πίνακας 3 Υλοποιήσεις Single-Scanner ατομικών στιγμιοτύπων από καταχωρητές ανάγνωσης-εγγραφής.

| Υλοποίηση | Αριθμός καταχωρητών | Μέγεθος καταχωρητών (bit) | SCAN | UPDATE |
|---|---|---|---|---|
| SweepLine [18] | $m+1$ | $O(m\log\|T\| + \log k + \log n)$ | $O(m^2)$ | $O(m^2)$ |
| Linear [18] | $m+k$ | $O(\max\{m\log\|T\|, \log k\})$ | $O(m)$ | $O(m)$ |
| T-Opt [20] | $km$ | $O(\max\{\log k, \log\|T\|\})$ | $O(m)$ | $O(1)$ |
| RT [20] | $O(mn)$ | $O(\max\{\log n, \log\|T\|\})$ | $O(n)$ | $O(1)$ |
| RT-Opt [20] | $O(mn)$ | $O(\max\{\log n, \log\|T\|\})$ | $O(m)$ | $O(1)$ |
| Jayanti [25] | $O(n)$ | $O(\log \|T\|)$ | $O(n)$ | $O(1)$ |
| Kirousis et al. [29] | $O(mn)$ | $O(\max\{mn\log n, \log\|T\|\})$ | $O(mn)$ | $O(1)$ |
| Riany et al. (32) | $n+1$ | $O(\max\{\log n, \log\|T\|\})$ | $O(n)$ | $O(1)$ |

Η καλύτερη υλοποίηση ατομικών στιγμιοτύπων που χρησιμοποιεί καταχωρητές ισχυρότερους από τους καταχωρητές ανάγνωσης-εγγραφής έχει παρουσιαστεί στο [25]. Αυτή η υλοποίηση επιτυγχάνει χρονική πολυπλοκότητα για τη SCAN O$(m)$ και για την UPDATE $O(1)$ χρησιμοποιώντας $O(mn)$ καταχωρητές CAS μικρού μεγέθους. Ο αλγόριθμος C-Snap επιτυγχάνει τις ίδιες χρονικές πολυπλοκότητες για τη SCAN και την UPDATE, χρησιμοποιώντας μόνο $O(m)$ καταχωρητές CAS, αλλά ένας από αυτούς έχει μεγάλο μέγεθος. Στον παρακάτω πίνακα παρουσιάζεται μία σύνοψη των χαρακτηριστικών των αλγορίθμων που χρησιμοποιούν καταχωρητές ισχυρότερους από τους καταχωρητές ανάγνωσης-εγγραφής.

Πίνακας 4 Υλοποιήσεις ατομικών στιγμιοτύπων από ισχυρότερους καταχωρητές.

| Υλοποίηση | Τύπος καταχωρητών | Αριθμός καταχωρητών | SCAN | UPDATE |
|---|---|---|---|---|
| C-Snap [20] | CAS & r/w | $O(m)$ | $O(m)$ | $O(1)$ |
| Jayanti [25] | CAS & r/w | $O(mn^2)$ | $O(m)$ | $O(1)$ |
| Jayanti [26] | CAS & r/w | $O(mn^2)$ | $O(m)$ | $O(m)$ |
| Riany et al. (32) | CAS & Fetch&Inc & r/w | $O(n^2)$ | $O(n)$ | $O(1)$ |

# ΚΕΦΑΛΑΙΟ 3. ΜΟΝΤΕΛΟ



## 3.1. Μοντελοποίηση συστήματος

Υποθέτουμε ότι στο σύστημα εκτελούνται $n$ διεργασίες $p_1, \ldots, p_n$, οι οποίες μοντελοποιούνται ως μηχανές καταστάσεων (ενδεχόμενα μη πεπερασμένες), όπου κάθε διεργασία $p_i$ έχει ένα σύνολο καταστάσεων $Q_i$. Οι διεργασίες του συστήματος επικοινωνούν μεταξύ τους προσπελάζοντας κοινά ατομικά αντικείμενα. Φορμαλιστικά, κάθε *ατομικό αντικείμενο* δύναται να αποθηκεύσει ένα σύνολο πληροφοριών και έχει έναν τύπο, ο οποίος προσδιορίζει το είδος των τιμών που μπορούν να αποθηκευθούν στο ατομικό αντικείμενο και τις ατομικές λειτουργίες που το αντικείμενο υποστηρίζει.

Το απλούστερο ατομικό αντικείμενο είναι ο καταχωρητής. Ένας *καταχωρητής* αποθηκεύει στοιχεία από ένα σύνολο $T$[1] και υποστηρίζει διάφορες λειτουργίες για την προσπέλαση και τροποποίηση των αποθηκευμένων δεδομένων. Αν το σύνολο $T$ περιέχει μη πεπερασμένο αριθμό στοιχείων, τότε το μέγεθος του καταχωρητή είναι μη πεπερασμένο. Θεωρούμε ότι το υλικό του συστήματος εγγυάται την ατομική εκτέλεση όλων των λειτουργιών των καταχωρητών. Για αυτό το λόγο, από τούδε και εις το εξής οι λειτουργίες των καταχωρητών θα αποκαλούνται *εντολές*. Υπάρχουν διάφορα είδη καταχωρητών. Στην παρούσα εργασία θα επικεντρωθούμε σε συστήματα που

[1] Το $T$ μπορεί να είναι ένα σύνολο τιμών ή ένα σύνολο διανυσμάτων, κ.τ.λ.

παρέχουν καταχωρητές ανάγνωσης-εγγραφής ή/και καταχωρητές *CAS* (ή Compare& Swap).

Ένας καταχωρητής *ανάγνωσης-εγγραφής* $R$ αποθηκεύει κάποια τιμή από κάποιο σύνολο τιμών $T$, ενώ υποστηρίζει την εντολή $read(R)$ και την εντολή $write(R, v)$ για την προσπέλαση και τροποποίηση των δεδομένων του. Η εντολή $read(R)$ επιστρέφει την τιμή που είναι αποθηκευμένη στον $R$ δίχως να τροποποιήσει τα δεδομένα του καταχωρητή. Η εντολή $write(R, v)$ δέχεται ως παράμετρο μία τιμή $v \in T$ και αλλάζει την τιμή του $R$ σε $v$ επιστρέφοντας μία επιβεβαίωση τερματισμού.

Ένας καταχωρητής *CAS* $R'$ αποθηκεύει τιμές από κάποιο σύνολο $T$ και υποστηρίζει τις εντολές $read(R')$ και $CAS(R', v_{old}, v_{new})$ για την προσπέλαση και τροποποίηση των δεδομένων του. Η εντολή $CAS(R', v_{old}, v_{new})$ δέχεται ως παράμετρο τις τιμές $v_{new}, v_{old} \in T$ και αλλάζει την τιμή του καταχωρητή $R'$ σε $v_{new}$ μόνο αν η τρέχουσα τιμή που είναι αποθηκευμένη στον $R'$ είναι ίση με $v_{old}$. Αν η τιμή που είναι αποθηκευμένη στον $R'$ είναι διαφορετική της $v_{old}$, τότε η εντολή $CAS(R', v_{old}, v_{new})$ δε μεταβάλλει τα περιεχόμενα του καταχωρητή $R'$. Αν η $CAS(R', v_{old}, v_{new})$ εγγράψει την τιμή $v_{new}$ στον $R'$, τότε λέμε πως η εν λόγω CAS είναι επιτυχημένη, σε αντίθετη περίπτωση λέμε ότι η CAS είναι αποτυχημένη. Σε κάθε περίπτωση η εντολή $CAS(R', v_{old}, v_{new})$ επιστρέφει μία επιβεβαίωση τερματισμού.

Σε έναν καταχωρητή *πολλαπλής εγγραφής*, οποιαδήποτε διεργασία δύναται να τροποποιεί τα περιεχόμενά του. Αντίθετα σε έναν καταχωρητή *απλής εγγραφής* μόνο μία διεργασία δύναται να τροποποιεί τα περιεχόμενά του.

Ένα *αντικείμενο ατομικών στιγμιοτύπων* αποτελείται από $m$ συνιστώσες. Σε κάθε *συνιστώσα* του αντικειμένου αποθηκεύεται μια τιμή από ένα σύνολο τιμών $T$. Κάθε συνιστώσα ενός ατομικού στιγμιότυπου έχει μία αρχική τιμή, η οποία καθορίζεται από την υλοποίηση (συνήθως η αρχική αυτή τιμή είναι $null$). Τα ατομικά στιγμιότυπα υποστηρίζουν δύο λειτουργίες, τη $SCAN()$ και την $UPDATE(i, v)$ με $i \in \{1, \dots, m\}$ και $v \in T$, οι οποίες είναι δυνατό να εκτελούνται από πολλές διεργασίες ταυτόχρονα. Η λειτουργία $UPDATE(i, v)$ ενημερώνει την $i$-οστή συνιστώσα του αντικειμένου, αποθηκεύοντας σε αυτή την τιμή $v$ και επιστρέφει μία επιβεβαίωση τερματισμού. Η

SCAN επιστρέφει ένα ‘συνεπές’ διάνυσμα $m$ τιμών (μία τιμή για κάθε συνιστώσα)[2]. Υποθέτουμε ότι τις αρχικές τιμές των συνιστωσών του ατομικού στιγμιότυπου τις έχουν εγγράψει κάποιες «φανταστικές» λειτουργίες UPDATE.

Μία υλοποίηση ενός αντικειμένου ατομικών στιγμιοτύπων χρησιμοποιεί καταχωρητές για να αποθηκεύει τα δεδομένα των συνιστωσών του αντικειμένου και παρέχει αλγορίθμους για την υλοποίηση των λειτουργιών SCAN και UPDATE. Κάθε αλγόριθμος που υλοποιεί τη SCAN ή την UPDATE, εκτελεί εντολές στους καταχωρητές του συστήματος (ή γενικότερα στα στοιχειώδη ατομικά αντικείμενα) που χρησιμοποιούνται από την υλοποίηση. Στην παρούσα εργασία μελετώνται υλοποιήσεις ατομικών στιγμιοτύπων από καταχωρητές ανάγνωσης-εγγραφής και υλοποιήσεις που χρησιμοποιούν επιπρόσθετα καταχωρητές CAS.

Τα αντικείμενα ατομικών στιγμιότυπων διακρίνονται σε ατομικά στιγμιότυπα πολλαπλής εγγραφής (Multi-Writer ή MW) και σε ατομικά στιγμιότυπα απλής εγγραφής (Single-Writer ή SW). Στα ατομικά στιγμιότυπα *πολλαπλής εγγραφής*, οποιαδήποτε διεργασία δύναται να εκτελέσει μια UPDATE σε οποιαδήποτε συνιστώσα του αντικειμένου. Έτσι κάθε συνιστώσα είναι δυνατό να ενημερώνεται ταυτόχρονα από περισσότερες από μία διεργασίες. Στα ατομικά στιγμιότυπα *απλής εγγραφής* κάθε συνιστώσα του αντικειμένου αντιστοιχίζεται με μία διεργασία, και κάθε συνιστώσα ενημερώνεται μόνο από τη διεργασία που της έχει αντιστοιχιστεί. Και στις δύο περιπτώσεις είναι δυνατό οποιαδήποτε διεργασία να εκτελεί SCAN ταυτόχρονα με την εκτέλεση άλλων λειτουργιών SCAN ή UPDATE από άλλες διεργασίες.

Μία *καθολική κατάσταση* $C$ είναι ένα διάνυσμα, του οποίου τα στοιχεία είναι οι καταστάσεις (states) των διεργασιών και οι τιμές των κοινών καταχωρητών του συστήματος. Μία καθολική κατάσταση περιγράφει το σύστημα σε κάποια χρονική στιγμή. Στην αρχική καθολική κατάσταση, κάθε διεργασία βρίσκεται σε αρχική κατάσταση και κάθε καταχωρητής περιέχει μία αρχική τιμή. Ένα *βήμα υπολογισμού* (step) μιας διεργασίας αποτελείται από μία ακριβώς προσπέλαση σε κάποιον από τους διαμοιρα-

[2] Η έννοια του συνεπούς διανύσματος ορίζεται φορμαλιστικά στην επόμενη ενότητα.

ζόμενους καταχωρητές, ενώ επίσης δύναται να εμπεριέχει και την εκτέλεση μίας σειράς τοπικών λειτουργιών πριν την προσπέλαση αυτή. Μία *εκτέλεση* $a$ είναι μία ακολουθία βημάτων υπολογισμού των διάφορων διεργασιών, η οποία εκτελείται από την αρχική κατάσταση του συστήματος. Μία εκτέλεση $a$ ενός αλγορίθμου ξεκινώντας από μία καθολική κατάσταση $C$ είναι *έγκυρη*, αν τα βήματα υπολογισμού που επιτελούνται από κάθε διεργασία ακολουθούν τον αλγόριθμο της διεργασίας και για κάθε αντικείμενο, η αποκρίση κάθε λειτουργίας που επιτελείται στο αντικείμενο είναι συμβατή με τον ορισμό της λειτουργίας. Ένα *διάστημα εκτέλεσης* της $a$ είναι ένα τμήμα της $a$ που αποτελείται από έναν αριθμό συνεχόμενων βημάτων. Έστω $s_1$ και $s_2$ δύο βήματα της εκτέλεσης $a$, τέτοια ώστε το $s_1$ να επιτελέσθηκε πριν το $s_2$. Συμβολίζουμε με $a(s_1, s_2)$ το διάστημα εκτέλεσης της $a$ που ξεκινά με το $s_1$ και τελειώνει με το $s_2$. Μία εκτέλεση ονομάζεται *σειριακή*, αν σε κάθε χρονική στιγμή της εκτέλεσης εκτελείται μόνο μία λειτουργία.

Μία λειτουργία SCAN ή UPDATE ξεκινά την εκτέλεσή της όταν εκτελείται το πρώτο βήμα υπολογισμού του αλγορίθμου της. Η απόκριση της λειτουργίας σημειοδοτείται από την εκτέλεση του τελευταίου βήματος υπολογισμού του αλγορίθμου της. Ένα ατομικό στιγμιότυπο είναι *Single-Scanner* (ή SS), αν μόνο μία SCAN μπορεί να εκτελείται κάθε χρονική στιγμή. Το *διάστημα εκτέλεσης* μιας SCAN (ή UPDATE) είναι το διάστημα εκτέλεσης που ξεκινά με την εκτέλεση του πρώτου βήματος υπολογισμού του αλγορίθμου της SCAN (ή UPDATE) και τελειώνει με την εκτέλεση του τελευταίου βήματος υπολογισμού του αλγορίθμου της. Έστω $op$ μία SCAN (ή UPDATE) που επιτελέσθηκε στην $a$. Αν το τελευταίο βήμα υπολογισμού της $op$ προηγείται του πρώτου βήματος μίας άλλης SCAN (ή UPDATE) $op'$, τότε λέμε ότι η $op$ *προηγείται* της $op'$ (ή ότι η $op'$ έπεται της $op$). Σε κάθε άλλη περίπτωση το διάστημα εκτέλεσης της $op$ επικαλύπτει το διάστημα εκτέλεσης της $op'$ (ή το διάστημα εκτέλεσης της $op'$ επικαλύπτει το διάστημα εκτέλεσης της $op$).

Η *χρονική πολυπλοκότητα* μίας λειτουργίας SCAN (ή UPDATE) μίας υλοποίησης ατομικών στιγμιοτύπων είναι ο μέγιστος αριθμός βημάτων υπολογισμού που επιτελούνται από μια διεργασία για την εκτέλεση της SCAN (ή UPDATE) σε οποιαδήποτε εκτέλεση της υλοποίησης. Από τα παραπάνω είναι προφανές ότι η χρονική πολυπλοκότητα μίας λειτουργίας SCAN ή UPDATE εξαρτάται μόνο από πλήθος των προσπε-

λάσεων στην κοινή μνήμη και όχι από το πλήθος των τοπικών υπολογισμών που εκτελούνται. Η *χρονική πολυπλοκότητα μίας υλοποίησης* είναι η μέγιστη χρονική πολυπλοκότητα των πολυπλοκοτήτων των λειτουργιών SCAN και UPDATE της υλοποίησης. Η *χωρική πολυπλοκότητα* (ή πολυπλοκότητα μνήμης) είναι ο μέγιστος αριθμός καταχωρητών που η υλοποίηση χρησιμοποιεί σε οποιαδήποτε εκτέλεση. Υποθέτουμε ότι οι διεργασίες δύναται να αποτυγχάνουν, σταματώντας την εκτέλεσή τους σε οποιαδήποτε χρονική στιγμή. Στην παρούσα εργασία μελετώνται υλοποιήσεις ατομικών στιγμιοτύπων που πληρούν την ιδιότητα *ελεύθερη-αναμονής*, όπου κάθε διεργασία ολοκληρώνει την εκτέλεσή της μέσα σε έναν πεπερασμένο αριθμό υπολογιστικών βημάτων ανεξάρτητα από την κατάσταση και την ταχύτητα εκτέλεσης των υπολοίπων διεργασιών.

### 3.2. Σειριοποιησιμότητα (linearizability)

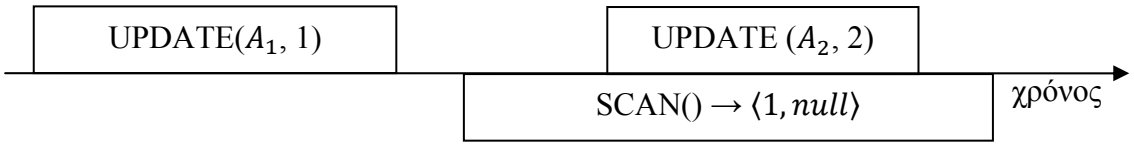
Σε αυτή την ενότητα θα παρουσιαστεί η έννοια της σειριοποιησιμότητας [22]. Έστω μία υλοποίηση $Y$ ενός ατομικού στιγμιοτύπου και έστω $a$ μία οποιαδήποτε εκτέλεση της $Y$. Η $a$ είναι σειριοποιήσιμη αν ισχύουν τα ακόλουθα.

1) Για κάθε λειτουργία SCAN ή UPDATE *op* που έχει περατωθεί στην $a$, δύναται να επιλέξουμε ένα *σημείο σειριοποίησης* **op* σε ένα σημείο μεταξύ της κλήσης και της απόκρισης της *op* τέτοιο ώστε να ισχύει η (3).
2) Δύναται να επιλέξουμε ένα υποσύνολο Φ των λειτουργιών των οποίων η εκτέλεση δεν έχει ολοκληρωθεί στην $a$, έτσι ώστε για κάθε λειτουργία *op* στο Φ:
   a) να μπορούμε να εισάγουμε μία απόκριση, και
   b) να μπορούμε να επιλέξουμε ένα σημείο σειριοποίησης **op*, το οποίο να βρίσκεται σε κάποιο σημείο της εκτέλεσης που έπεται της κλήσης της *op* τέτοιο ώστε να ισχύει η (3).
3) Αν οι λειτουργίες που έχουν επιτελεσθεί στην $a$ και οι λειτουργίες του Φ εκτελεστούν σειριακά (η μία μετά την άλλη) με τη σειρά που καθορίζουν τα σημεία σειριοποίησής τους, πρέπει να έχουν ακριβώς τις ίδιες αποκρίσεις με αυτές στην παράλληλη εκτέλεση.

Τοποθετώντας σημεία σειριοποίησης σε κάθε λειτουργία, οι λειτουργίες συμπεριφέρονται σαν να εκτελέστηκαν στιγμιαία στο χρονικό σημείο που ορίζει το σημείο σειριοποίησης. Η σειριοποιησιμότητα ενός ατομικού αντικειμένου εγγυάται ότι ακόμα και αν εκτελούνται παράλληλα περισσότερες από μία λειτουργίες στο αντικείμενο, τότε αυτές θα «φαίνεται» σαν να εκτελούνται σειριακά (ή *ατομικά*).

Μία υλοποίηση είναι σειριοποιήσιμη αν οποιαδήποτε εκτέλεση της υλοποίησης είναι σειριοποιήσιμη. Η ιδιότητα της σειριοποιησιμότητας είναι γνωστή και ως ιδιότητα της *ατομικότητας*. Μία SCAN $S$ της $a$ επιστρέφει ένα *συνεπές διάνυσμα* τιμών, αν για κάθε συνιστώσα $A_i, i \in \{1, \dots, m\}$, η $S$ επιστρέφει για την $A_i$ την τιμή που εγγράφηκε από την τελευταία UPDATE στην $A_i$ που έχει σειριοποιηθεί πριν την $S$.

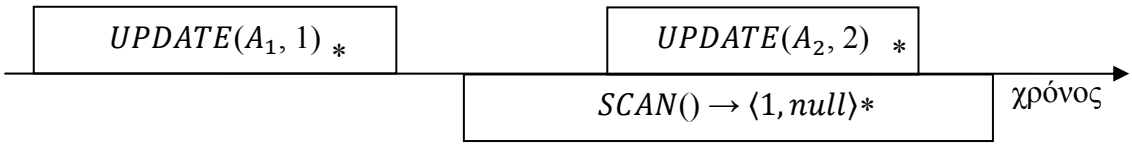
Για να γίνει κατανοητή η έννοια της σειριοποιησιμότητας θα παρατεθούν διάφορα παραδείγματα σειριοποιήσιμων εκτελέσεων. Έστω ότι έχουμε ένα αντικείμενο ατομικών στιγμιοτύπων, το οποίο αποτελείται από δύο συνιστώσες $(\mathrm{A}_1, \mathrm{A}_2)$. Υποθέτουμε ότι οι αρχικές τιμές των συνιστωσών του αντικειμένου είναι ίσες με $null$. Έστω ότι έχουμε την ακόλουθη εκτέλεση.




Σχήμα 3.1 Παράδειγμα εκτέλεσης λειτουργιών ατομικού στιγμιότυπου.

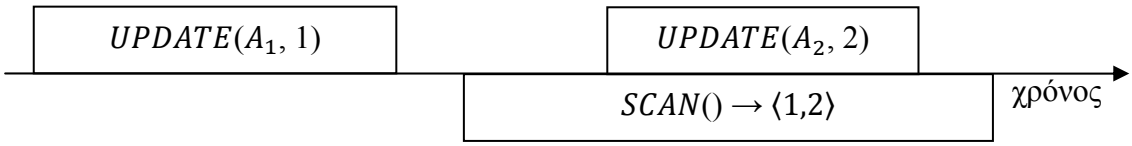
Θα εξετάσουμε αν η εκτέλεση του Σχήματος 3.1 είναι σειριοποιήσιμη. Πρέπει να τοποθετηθούν σημεία σειριοποίησης στις τρεις λειτουργίες με τέτοιο τρόπο ώστε στη σειριακή εκτέλεση που ορίζεται από τα σημεία σειριοποίησης, κάθε λειτουργία να έχει την ίδια απόκριση με αυτή που έχει στην παράλληλη εκτέλεση του Σχήματος 3.1. Από το σχήμα παρατηρούμε ότι η SCAN επιστρέφει την τιμή της $UPDATE(A_1, 1)$, ενώ δεν επιστρέφει την τιμή της $UPDATE(A_2, 2)$, αφού το διάνυσμα τιμών που επιστρέφει είναι το $\langle 1, null \rangle$. Άρα συμπεραίνουμε ότι πρώτα πρέπει να σειριοποιηθεί η $UPDATE(\mathrm{A}_1, 1)$, έπειτα η $SCAN() \rightarrow \langle 1, null \rangle$ και τέλος η $UPDATE(\mathrm{A}_2, 2)$, αφού μόνο σε αυτή την περίπτωση οι αποκρίσεις των λειτουργιών θα είναι ίδιες στην παράλληλη εκτέλεση και στην σειριακή εκτέλεση που ορίζεται από τα σημεία σειριοποίησης. Στο Σχήμα 3.2 φαίνεται ο τρόπος με τον οποίο ανατίθενται τα σημεία σειρι-

οποίησης στις λειτουργίες της εκτέλεσης του Σχήματος 3.1. Παρατηρούμε ότι τα σημεία σειριοποίησης των λειτουργιών έχουν επιλεχθεί έτσι ώστε να βρίσκονται εντός του διαστήματος εκτέλεσης των λειτουργιών τους, όπως και απαιτείται.



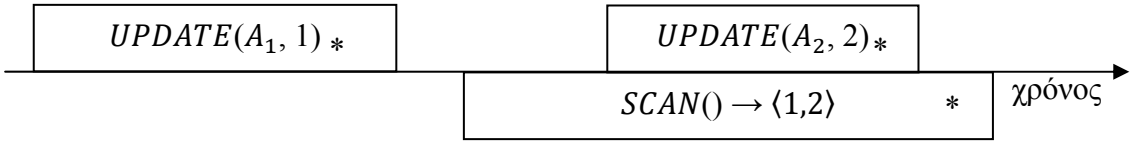

Σχήμα 3.2 Παράδειγμα σειριοποίησης λειτουργιών.

Ας δούμε τώρα ένα ακόμη παράδειγμα σειριοποιήσιμης εκτέλεσης. Έστω η εκτέλεση του Σχήματος 3.3.



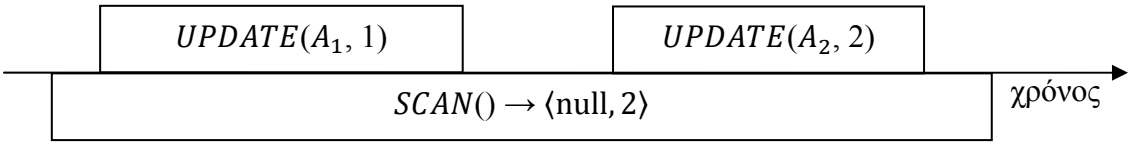

Σχήμα 3.3 Παράδειγμα εκτέλεσης λειτουργιών ατομικού στιγμιοτύπου.

Από το Σχήμα 3.3 παρατηρούμε ότι η SCAN επιστρέφει την τιμή της $UPDATE(A_1, 1)$ και την τιμή της $UPDATE(A_2, 2)$, αφού το διάνυσμα τιμών που επιστρέφει είναι το $\langle 1,2 \rangle$. Έτσι συμπεραίνουμε ότι πρώτα πρέπει να σειριοποιηθεί η $UPDATE(\mathrm{A}_1, 1)$ λειτουργία, έπειτα η λειτουργία $UPDATE(\mathrm{A}_2, 2)$ και τέλος η SCAN (βλ. Σχήμα 3.4), αφού μόνο σε αυτή την περίπτωση οι αποκρίσεις των λειτουργιών θα είναι ίδιες στην παράλληλη εκτέλεση και στην σειριακή εκτέλεση που ορίζεται από τα σημεία σειριοποίησης. Στο Σχήμα 3.4 φαίνεται ο τρόπος με τον οποίο τοποθετούνται τα σημεία σειριοποίησης στην εκτέλεση του Σχήματος 3.3. Παρατηρούμε ότι και σε αυτή την περίπτωση τα σημεία σειριοποίησης των λειτουργιών, βρίσκονται εντός του διαστήματος εκτέλεσης των λειτουργιών τους.



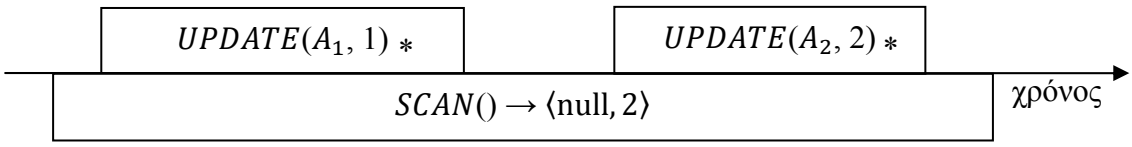

Σχήμα 3.4 Παράδειγμα σειριοποίησης λειτουργιών.

Σε αυτό το σημείο θα μελετήσουμε ένα παράδειγμα μη σειριοποιήσιμης εκτέλεσης. Έστω η εκτέλεση του Σχήματος 3.5.




Σχήμα 3.5 Παράδειγμα εκτέλεσης λειτουργιών στιγμιοτύπου.

Οι λειτουργίες $UPDATE(\mathrm{A}_1, 1)$, $UPDATE(\mathrm{A}_2, 2)$ θα σειριοποιηθούν σε κάποιο σημείο μεταξύ της έναρξης και του τέλους της εκτέλεσής τους (βλ. Σχήμα 3.6). Έτσι απομένει να σειριοποιηθεί η SCAN. Από το Σχήμα 3.6 παρατηρούμε ότι η SCAN δε δύναται να σειριοποιηθεί πριν την $UPDATE(\mathrm{A}_1, 1)$ διότι σε αυτή την περίπτωση σύμφωνα με τη σειριακή εκτέλεση που ορίζουν τα σημεία σειριοποίησης, η SCAN έπρεπε να έχει επιστρέψει το διάνυσμα τιμών $\langle null, null \rangle$. Επίσης παρατηρούμε ότι η SCAN δε δύναται να σειριοποιηθεί ανάμεσα στην $UPDATE(\mathrm{A}_1, 1)$ και στην $UPDATE(\mathrm{A}_2, 2)$, καθώς σε αυτή την περίπτωση σύμφωνα με τη σειριακή εκτέλεση που ορίζουν τα σημεία σειριοποίησης, η SCAN έπρεπε να έχει επιστρέψει το διάνυσμα τιμών $\langle 1, null \rangle$. Τέλος η SCAN δε δύναται να σειριοποιηθεί ούτε έπειτα από την $UPDATE(\mathrm{A}_2, 2)$, αφού σύμφωνα με τη σειριακή εκτέλεση που ορίζουν τα σημεία σειριοποίησης η SCAN σε αυτή την περίπτωση θα επέστρεφε $\langle 1,2 \rangle$. Έτσι όπου και να βάλουμε το σημείο σειριοποίησης της SCAN, η SCAN δε θα επιστρέφει στη σειριακή εκτέλεση που ορίζεται από τα σημεία σειριοποίησης, το ίδιο διάνυσμα τιμών με εκείνο που επιστρέφει στην παράλληλη εκτέλεση. Άρα η υλοποίηση που μας οδηγεί σε μία τέτοια εκτέλεση είναι εσφαλμένη.




Σχήμα 3.6 Προσπάθεια σειριοποίησης της εκτέλεσης.

### 3.3. Ένας εσφαλμένος αλγόριθμος

Η υλοποίηση ενός ατομικού στιγμιοτύπου δεν είναι μία εύκολη διαδικασία. Για να γίνει διαισθητικά φανερό αυτό, θα παρουσιαστεί ένας προφανής αλγόριθμος για την υλοποίηση ενός αντικειμένου ατομικών στιγμιοτύπων, ο οποίος όπως θα δειχθεί στη

συνέχεια δεν είναι ορθός. Συγκεκριμένα θα αποδειχθεί ότι ο εν λόγω αλγόριθμος δεν πληροί την ιδιότητα της σειριοποιησιμότητας.

Ας θεωρήσουμε την απλή περίπτωση που το αντικείμενο ατομικών στιγμιότυπων αποτελείται από 2 συνιστώσες. Στο εν λόγω αντικείμενο οποιαδήποτε διεργασία δύναται να ενημερώνει οποιαδήποτε συνιστώσα (ατομικό στιγμιότυπο πολλαπλής εγγραφής), και φυσικά οποιαδήποτε διεργασία έχει τη δυνατότητα να επιτελεί SCAN στο σύστημα.

Η συγκεκριμένη υλοποίηση χρησιμοποιεί 2 καταχωρητές, τον $R_1$ και τον $R_2$. Με κάθε συνιστώσα έχει συσχετισθεί ένας καταχωρητής, στον οποίο αποθηκεύεται η τιμή της συνιστώσας. Έστωσαν $\mathrm{A}_1$, $\mathrm{A}_2$ οι δύο συνιστώσες του ατομικού στιγμιοτύπου. Στη συνιστώσα $\mathrm{A}_1$ αντιστοιχίζεται ο καταχωρητής ανάγνωσης-εγγραφής $R_1$, ενώ στη συνιστώσα $\mathrm{A}_2$ αντιστοιχίζεται ο καταχωρητής ανάγνωσης-εγγραφής $R_2$. Η υλοποίηση της UPDATE είναι εξαιρετικά απλή. Η διεργασία που εκτελεί μία UPDATE σε μια συνιστώσα, απλώς εγγράφει τη νέα τιμή στον καταχωρητή που έχει αντιστοιχισθεί στη συνιστώσα. Η SCAN υλοποιείται ως εξής, η διεργασία που εκτελεί μία SCAN διαβάζει με τη σειρά όλους τους καταχωρητές και επιστρέφει τις τιμές που διάβασε ως αποτέλεσμα.

Ο παραπάνω αλγόριθμος είναι ο πλέον προφανής. Όμως η συγκεκριμένη υλοποίηση δεν είναι σειριοποιήσιμη. Θα κατασκευαστεί ένα παράδειγμα, στο οποίο θα δειχθεί ότι δεν πληρούται η ιδιότητα της σειριοποιησιμότητας.

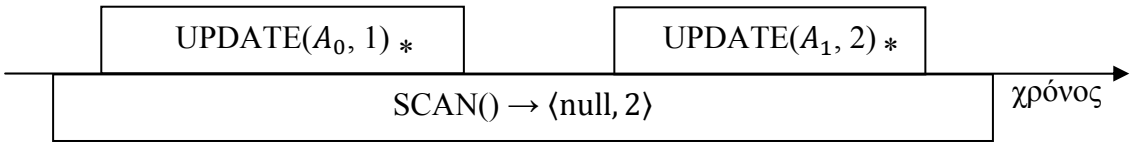
Υποθέτουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι η εν λόγω προφανής υλοποίηση είναι ορθή. Έστω ότι στο σύστημα εκτελούνται δύο διεργασίες, η $P$ και η $S$. Η διεργασία $P$ ενημερώνει πρώτα τη συνιστώσα $\mathrm{A}_1$ και έπειτα ενημερώνει τη συνιστώσα $\mathrm{A}_2$ (βλ. Σχήμα 3.7). Η διεργασία $S$ επιτελεί μία SCAN. Υποθέτουμε ότι οι καταχωρητές του συστήματος έχουν αρχικές τιμές ίσες με $null$ και ως εκ τούτου αρχικά και οι τιμές των συνιστωσών είναι ίσες με $null$. Η διεργασία $S$ ξεκινά να εκτελεί μία SCAN, εκτελώντας αρχικά μία read στον $R_1$, όπου διαβάζει την τιμή $null$ και σταματά. Έπειτα εκτελείται η διεργασία $P$ και ενημερώνει τη συνιστώσα $A_1$, αποθηκεύοντας την τιμή 1 στον καταχωρητή $R_1$. Αμέσως μετά η διεργασία $P$ συνεχίζει την εκτέλεσή της ενημερώνοντας τη συνιστώσα $\mathrm{A}_2$, αποθηκεύοντας την τιμή 2 στον κατα-

χωρητή $R_2$. Ύστερα η SCAN συνεχίζει την εκτέλεσή της, διαβάζει την τιμή 2 από τον καταχωρητή $R_2$ και έτσι επιστρέφει το διάνυσμα τιμών $\langle \text{null}, 2 \rangle$. Ο παραπάνω τρόπος εκτέλεσης των δύο UPDATE και της SCAN, φαίνεται στο Σχήμα 3.7.

| Λειτουργίες Διεργασιών | | Τιμές Καταχωρητών | |
|---|---|---|---|
| $S$ | $P$ | $R_1$ | $R_2$ |
| | | null | null |
| $read(R_1) \rightarrow \text{null}$ | | | |
| | $write(R_1, 1)$ | | |
| | | 1 | null |
| | $write(R_2, 2)$ | | |
| | | 1 | 2 |
| $read(R_2) \rightarrow 2$ | | | |

Σχήμα 3.7 Μία εκτέλεση του προφανούς αλγορίθμου.

Έχουμε υποθέσει πως ο αλγόριθμος είναι ορθός. Άρα θα πρέπει κάθε εκτέλεση του αλγορίθμου να είναι σειριοποιήσιμη. Επομένως πρέπει να αποδοθούν σημεία σειριοποίησης και για την εκτέλεση του Σχήματος 3.7. Τα σημεία σειριοποίησης πρέπει να αποδοθούν με τέτοιο τρόπο έτσι ώστε η σειριακή εκτέλεση των τριών αυτών λειτουργιών όπως καθορίζουν τα σημεία σειριοποίησης να επιστρέφει τα ίδια αποτελέσματα με την παράλληλη εκτέλεση του Σχήματος 3.7. Στο Σχήμα 3.8 γίνεται προσπάθεια εισαγωγής σημείων σειριοποίησης για την προαναφερθείσα εκτέλεση του αλγορίθμου.




Σχήμα 3.8 Διάγραμμα εκτέλεσης του εσφαλμένου αλγορίθμου.

Από το Σχήμα 3.8 γίνεται φανερό πως η εκτέλεση του Σχήματος 3.7 είναι η μη σειριοποιήσιμη εκτέλεση που παρουσιάζεται στην ενότητα 3.2. Ειδικότερα το σημείο σειριοποίησης κάθε UPDATE βρίσκεται έπειτα από το σημείο εκκίνησής της και πριν από το σημείο τερματισμού της. Τα δυνατά σημεία στα οποία μπορούμε να τοποθε-

τήσουμε το σημείο σειριοποίησης για τη SCAN είναι είτε πριν το σημείο σειριοποίησης της UPDATE στη συνιστώσα $A_1$, είτε έπειτα από το σημείο σειριοποίησης της UPDATE στη συνιστώσα $A_2$, είτε ανάμεσα στα σημεία σειριοποίησης των δύο UPDATE. Όπου και να βάλουμε το σημείο σειριοποίησης της SCAN, η SCAN δε θα επιστρέφει στη σειριακή εκτέλεση που ορίζεται από τα σημεία σειριοποίησης το ίδιο διάνυσμα τιμών με εκείνο που επιστρέφει στην παράλληλη εκτέλεση. Συγκεκριμένα, όπου και να βάλουμε τα σημεία σειριοποίησης σε καμία περίπτωση η SCAN δε θα επιστρέψει ως αποτέλεσμα το διάνυσμα τιμών $\langle \text{null}, 2 \rangle$. Άρα ο προφανής αλγόριθμος είναι εσφαλμένος.

### 3.4. Συμβάσεις ψευδοκώδικα

Ένας αλγόριθμος κοινής μνήμης περιγράφεται (για κάθε διεργασία του συστήματος) με μια μορφή ψευδοκώδικα, η οποία είναι σχεδόν ίδια με αυτή που χρησιμοποιείται και στους σειριακούς αλγορίθμους. Ο ψευδοκώδικας περιλαμβάνει τις προσπελάσεις στις τοπικές μεταβλητές (που αποτελούν μέρος της κατάστασης του επεξεργαστή), αλλά και τις προσπελάσεις στους κοινούς καταχωρητές. Οι κοινές μεταβλητές (καταχωρητές) δηλώνονται στον ψευδοκώδικα με τον ίδιο τρόπο όπως και οι τοπικές μεταβλητές, μόνο που στην αρχή της δήλωσης προστίθεται η λέξη $shared$ (π.χ. η δήλωση $shared\ int\ seq$ σημαίνει πως ο καταχωρητής $seq$ είναι κοινός, ενώ αντίθετα η δήλωση $int\ seq$ σημαίνει πως ο καταχωρητής $seq$ είναι τοπικός). Επίσης ο ψευδοκώδικας είναι δυνατό να περιλαμβάνει και προσπελάσεις σε μεταβλητές που στην αρχή της δήλωσή τους έχει προστεθεί η λέξη $static$. Αυτές οι μεταβλητές είναι τοπικές, μπορούν να χρησιμοποιηθούν μόνο στον ψευδοκώδικα των συναρτήσεων και έχουν την ιδιότητα να κρατούν την τιμή που είχε εκχωρήσει σε αυτές η προηγούμενη κλήση της συνάρτησης, στην οποία χρησιμοποιούνται. Ακόμη υποθέτουμε ότι στις συνιστώσες ενός στιγμιοτύπου αποθηκεύονται δεδομένα του τύπου $data$.

Για απλότητα, αντί να χρησιμοποιούνται οι εντολές $read$ και $write$ στον ψευδοκώδικα γίνονται οι ακόλουθες συμβάσεις: (α) μία αναφορά σε έναν κοινό καταχωρητή στο αριστερό μέρος μίας εκχώρησης σημαίνει ότι επιτελείται μία εντολή $write$ στον καταχωρητή, (β) μία αναφορά σε έναν κοινό καταχωρητή στο δεξιό μέρος μίας εκχώρη-

σης σημαίνει ότι επιτελείται μία εντολή $read$ στον καταχωρητή. Είναι δυνατό μία απλή εντολή του ψευδοκώδικα να αντιστοιχίζεται με πολλές εντολές στο φορμαλιστικό μοντέλο. Σε αυτή την περίπτωση, οι εντολές $read$ σε κοινές μεταβλητές στο δεξί μέρος μίας εκχώρησης επιτελούνται από τα αριστερά προς τα δεξιά, οι επιστρεφόμενες τιμές αποθηκεύονται σε τοπικές μεταβλητές, ο οριζόμενος υπολογισμός επιτελείται στις τοπικές μεταβλητές, και τέλος το αποτέλεσμα εγγράφεται στις μεταβλητές του αριστερού μέρους της εκχώρησης. Για παράδειγμα αν $X, Y$ και $Z$ είναι κοινές μεταβλητές, τότε η εντολή του ψευδοκώδικα $X = Y + Z$ σημαίνει ότι αρχικά διαβάζεται η τιμή της $Y$ και αποθηκεύεται σε μια τοπική μεταβλητή, έπειτα διαβάζεται η τιμή της $Z$ και αποθηκεύεται σε μια τοπική μεταβλητή, και τέλος το άθροισμα των τοπικών μεταβλητών εγγράφεται στη $X$.

Τα σχόλια στον ψευδοκώδικα ξεκινούν με //.

# ΚΕΦΑΛΑΙΟ 4. Ο ΑΛΓΟΡΙΘΜΟΣ LINEAR



Σε αυτό το κεφάλαιο παρουσιάζεται ένας Single-Scanner αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται Linear (Αλγόριθμος 4.1). Ο Linear έχει χρονική πολυπλοκότητα $O(m)$ για τη SCAN και την UPDATE. Από τους καταχωρητές που χρησιμοποιεί ο Linear, ο μεγαλύτερος σε μέγεθος αποτελείται από $O(\max\{m \log|T|, \log k\})$ bit, όπου $k$ είναι ο μέγιστος αριθμός SCAN που εκτελούνται μια εκτέλεση.

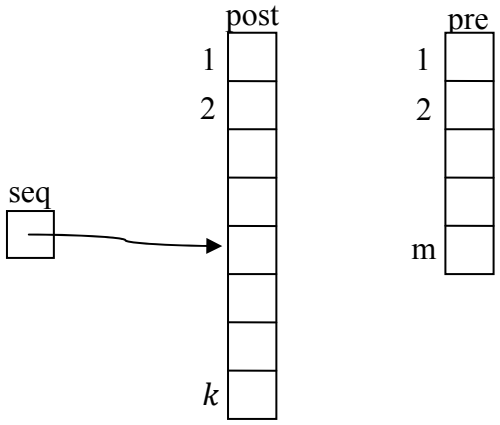
## 4.1. Περιγραφή του αλγορίθμου

Σε αυτή την ενότητα θα περιγραφεί ο αλγόριθμος Linear. Αρχικά θα παρουσιαστούν οι δομές δεδομένων που χρησιμοποιεί ο αλγόριθμος (βλ. Σχήμα 4.1). Ο καταχωρητής $seq$ έχει αρχική τιμή ίση με μηδέν, η οποία και αυξάνεται κάθε φορά που μια SCAN ξεκινά να εκτελείται. Επίσης χρησιμοποιούνται δύο πίνακες καταχωρητών, που ονομάζονται $post$ (περιέχει $k$ καταχωρητές) και $pre$ (περιέχει $m$ καταχωρητές). Για κάθε $j, 1 \leq j \leq m$, στον καταχωρητή $pre[j]$ αποθηκεύεται μία τιμή για τη συνιστώσα $A_j$. Ο καταχωρητής $pre[j]$ εγγράφεται μόνο από UPDATE που ενημερώνουν τη συνιστώσα $A_j$ και έχει αρχική τιμή ίση με $null$. Για κάθε $s \geq 1$ στον καταχωρητή $post[s], s \geq 1$ αποθηκεύεται ένα διάνυσμα $\vec{v}$ που αποτελείται από $m$ τιμές, μία για κάθε συνιστώσα του αντικειμένου. Αρχικά όλοι οι καταχωρητές του πίνακα $post$ περιέχουν το διάνυσμα $\langle null, \dots, null \rangle$.

Οι λειτουργίες SCAN και UPDATE του Linear προσπαθούν να υπολογίσουν ένα συνεπές διάνυσμα τιμών καλώντας τη συνάρτηση $get\_vector$. Συγκεκριμένα η λειτουργία SCAN αρχικά αυξάνει την τιμή του καταχωρητή $seq$, έπειτα καλεί τη συνάρ-

τηση $get\_vector$ με όρισμα την τρέχουσα τιμή του καταχωρητή $seq$ και επιστρέφει το διάνυσμα τιμών που επέστρεψε η συνάρτηση $get\_vector$. Μία UPDATE $U$ ξεκινά την εκτέλεσή της διαβάζοντας τον καταχωρητή $seq$. Η τιμή που είναι αποθηκευμένη εκεί χρησιμοποιείται για να δεικτοδοτήσει τη θέση του πίνακα $post$, στην οποία η $U$ θα αποθηκεύσει το διάνυσμα τιμών που υπολογίζει καλώντας τη συνάρτηση $get\_vector$.

Μία συνάρτηση $get\_vector$ $g$ παίρνει ως παράμετρο την τρέχουσα τιμή $s$ που έχει ο καταχωρητής $seq$ και επιτελεί τα ακόλουθα. Αρχικά η $g$ διαβάζει τους $m$ καταχωρητές του πίνακα $pre$ και έπειτα διαβάζει τον καταχωρητή $post[s]$. Αν η τιμή του καταχωρητή $post[s]$ είναι διαφορετική του $null$, τότε η $g$ επιστρέφει το διάνυσμα τιμών που ανέγνωσε στον καταχωρητή $post[s]$. Σε αντίθετη περίπτωση, η $g$ επιστρέφει τις τιμές που ανέγνωσε στους καταχωρητές του πίνακα $pre$.




Σχήμα 4.1 Δομές δεδομένων του Linear.

Μια SCAN προσπαθεί να επιστρέψει ως διάνυσμα τιμών, τις τιμές που εγγράφηκαν στον $pre$ από UPDATE που ξεκίνησαν την εκτέλεσή τους πριν από την έναρξη της SCAN. Όμως είναι δυνατό μία $UPDATE(i, v)$ που ξεκίνησε την εκτέλεσή της έπειτα από τη SCAN να πανωγράψει στον $pre[j]$ την τιμή μιας $UPDATE(i, v')$ που είχε ξεκινήσει την εκτέλεσή της πριν την έναρξη της SCAN, προτού η SCAN προλάβει να διαβάσει όλους τους καταχωρητές του $pre$. Για να αποφευχθεί το παραπάνω ενδεχόμενο, οι UPDATE που ξεκινούν τη λειτουργία τους έπειτα από την έναρξη της SCAN, αποθηκεύουν ένα διάνυσμα με τις τιμές του $pre$ σε έναν καταχωρητή του πίνακα $post$ προτού γράψουν τη νέα τιμή τους σε κάποιον καταχωρητή του $pre$. Ειδι-

κότερα, η SCAN κατά την έναρξή της αυξάνει την τιμή του καταχωρητή $seq$ κατά ένα (ο οποίος δείχνει σε κάποια θέση του πίνακα $post$). Έτσι την πρώτη φορά που τερματίζει μια UPDATE που ξεκινά την εκτέλεσή της έπειτα από την έναρξη της SCAN, αποθηκεύει ένα διάνυσμα που περιέχει παλαιότερες τιμές του $pre$, στον $post[seq]$ (γραμμή 3). Με αυτό τον τρόπο, αν η SCAN εντοπίσει πως έχει ξεκινήσει η εκτέλεση κάποιας UPDATE έπειτα από την έναρξή της (διαβάζει μια τιμή στον $post$ που είναι διάφορη του $null$) επιστρέφει τις τιμές που είναι αποθηκευμένες στον $post[seq]$, οι οποίες εγγράφηκαν στον $pre$ από UPDATE που είχαν ξεκινήσει πριν την έναρξη της SCAN.

Αλγόριθμος 4.1 Ψευδοκώδικας για τον αλγόριθμο Linear.

```
 struct post_data{
    data view[1..m];
 };

 shared data pre[1..m]={null, ..., null};
 shared post_data post[1..k]={<null,...,null>, ..., <null,...,null>};
 shared int seq=0;

 void UPDATE(data v){
    data view[1..m];
    int curr_seq;

1.  curr_seq=seq;
2.  view=get_vector(curr_seq);
3.  post[curr_seq]=view;
4.  pre[j]=v;
 }

 data[] SCAN(){
    data view[1..m];
    int curr_seq;

5.  seq=seq+1;
6.  view=get_vector(seq);
    return view;
 }

 data[] get_vector(int curr_seq){
    data view[1..m];
    int j;

7.  for(j=1;j≤m;j++)
8.    view[j]=pre[j];
9.  if(post[curr_seq]≠null)
10.   return post[curr_seq];
11. return view;
 }
```

### 4.2. Χρονική και χωρική πολυπλοκότητα του αλγορίθμου

Σε αυτή την ενότητα θα μελετηθεί η χρονική πολυπλοκότητα καθώς και η πολυπλοκότητα μνήμης του αλγορίθμου Linear. Είναι προφανές ότι κάθε SCAN και UPDATE επιτελεί έναν σταθερό αριθμό προσβάσεων στην κοινή μνήμη επιπρόσθετα της κλήσης της συνάρτηση $get\_vector$. Από τον ψευδοκώδικα προκύπτει ότι η συνάρτηση $get\_vector$ επιτελεί $m + 1$ αναγνώσεις σε καταχωρητές της κοινής μνήμης. Έτσι η χρονική πολυπλοκότητα της SCAN και της UPDATE είναι $O(m)$.

Ο πίνακας $pre$ αποτελείται από $m$ καταχωρητές, όπου σε κάθε έναν από αυτούς αποθηκεύεται μία τιμή ενός συνόλου $T$. Άρα κάθε ένας από αυτούς έχει μέγεθος $O(\log|T|)$ bit. Ο πίνακας $post$ αποτελείται από τόσους καταχωρητές, όσες και οι SCAN που πρόκειται να επιτελεσθούν σε οποιαδήποτε εκτέλεση (ο αριθμός των οποίων δύναται να είναι μη πεπερασμένος). Κάθε καταχωρητής του πίνακα $post$ αποτελείται από ένα διάνυσμα $m$ τιμών ενός συνόλου $T$. Έτσι το μέγεθος κάθε καταχωρητή του πίνακα $post$ είναι $O(m \log|T|)$ bit. Τέλος ο καταχωρητής $seq$ έχει μέγεθος $O(\log k)$ bit. Επειδή το $k$ μπορεί να αυξάνει απεριόριστα, το μέγεθος του $seq$ δεν είναι απαραίτητα πεπερασμένο.

Ο Linear μπορεί να υλοποιηθεί αρκετά πιο αποτελεσματικά αν το σύστημα υποστηρίζει ανακύκλωση κοινής μνήμης, με τον ακόλουθο τρόπο. Κάθε φορά που μία SCAN $S$ ξεκινά την εκτέλεσή της, η διεργασία που επιτελεί την $S$ δεσμεύει δυναμικά ένα νέο τμήμα (block) κοινής μνήμης και θέτει σε έναν δείκτη (έστω $sptr$ ο εν λόγω δείκτης) τη διεύθυνση του εν λόγω τμήματος μνήμης. Όταν μία UPDATE στη συνιστώσα $A_i$ ξεκινά την εκτέλεσή της διαβάζει τον δείκτη $sptr$ (που πλέον παίζει τον ρόλο του καταχωρητή $seq$) και εγγράφει την τιμή που ανέγνωσε στον $pre[i]$, στην $i$-οστή καταχώρηση του τμήματος μνήμης στον οποίο δείχνει ο $sptr$. Η $S$ διαβάζει τις $m$ καταχωρήσεις του πίνακα $sptr$ και τις $m$ καταχωρήσεις του πίνακα $pre[i]$. Ψευδοκώδικας για αυτή την βελτιωμένη εκδοχή του Linear παρουσιάζεται στον Αλγόριθμο 4.1.

Στην βελτιωμένη έκδοση του Linear, ο καταχωρητής $seq$ έχει αντικατασταθεί από έναν δείκτη, ενώ μια διεργασία ανακύκλωσης μνήμης (garbage collector) δύναται να χρησιμοποιηθεί για την αποδέσμευση των τμημάτων μνήμης που δε χρησιμοποιού-

νται από τις διεργασίες (σημειώνεται ότι το πολύ $n$ δεσμευμένα τμήματα μνήμης θα χρησιμοποιούνται από τις $n$ διεργασίες του συστήματος.).

Αλγόριθμος 4.2 Βελτιωμένη εκδοχή του αλγορίθμου Linear.

```
 struct post_data{
    data view[1..m];
 };

 shared data pre[1..m]={null, ..., null};
 shared post_data sptr[]=new post_data(null);

 void UPDATE(data v){
    data view[1..m];
    data lptr[];

1.  lptr=sptr;
2.  v=get_vector(lptr);
3.  lptr.view=view;
4.  pre[j]=v;
 }

 data[] SCAN(){
    data view[1..m];

5.  sptr=new post_data(null);
6.  view=get_vector(sptr);
    return view;
 }

 data[] get_vector(post_data lptr[]){
    data view[1..m];
    int j;

7.  for(j=1;j≤m;j++)
8.    view[j]=pre[j];
9.  if(lptr.v≠null)
10.   return lptr;
11. return view;
 }
```

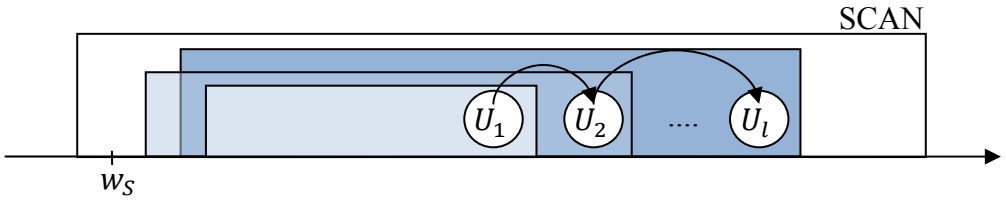
### 4.3. Απόδειξη της σειριοποιησιμότητας του αλγορίθμου

Σε αυτή την ενότητα θα αποδειχθεί η σειριοποιησιμότητα του Linear (Αλγόριθμος 4.1). Έστω $a$ μία οποιαδήποτε εκτέλεση του Linear, έστω $S$ μία οποιαδήποτε SCAN που εκτελείται στην $a$, και έστω $g$ η συνάρτηση $get\_vector$ που κλήθηκε από την $S$. Ο ακόλουθος συμβολισμός είναι χρήσιμος στους ορισμούς και στις αποδείξεις που

ακολουθούν. Ας υποθέσουμε ότι η $g$ επέστρεψε εκτελώντας τη γραμμή 10 του ψευδοκώδικα. Συμβολίζουμε με $k(g)$ τον μεγαλύτερο ακέραιο για τον οποίο ισχύουν τα ακόλουθα: υπάρχει μία ακολουθία από UPDATE $U_1, \ldots, U_{k(g)}$ τέτοιες ώστε:

1) η $U_{k(g)}$ να εγγράφει στον καταχωρητή $post$ το διάνυσμα τιμών που επιστράφηκε από την $g$.
2) για κάθε $l, 1 < l \leq k(g)$, η συνάρτηση $get\_vector$ $g_l$ που εκτελείται από την UPDATE $U_l$, τερματίζει στη γραμμή 10 του ψευδοκώδικα επιστρέφοντας το διάνυσμα τιμών που η λειτουργία $U_{l-1}$ έγγραψε στον $post$, ενώ η $g_1$ επιστρέφει εκτελώντας τη γραμμή 11 του ψευδοκώδικα.

Στο Σχήμα 4.2 φαίνεται καλύτερα ο ορισμός της ακολουθίας $U_1, \ldots, U_{k(g)}$. Ειδικότερα, η $U_l$ επιστρέφει το ίδιο διάνυσμα τιμών με την $U_{l-1}$, ενώ η $U_1$ είναι η πρώτη UPDATE που τερματίζει από τις $U_1, \ldots, U_{k(g)}$.




Σχήμα 4.2 Η $U_{l-1}$ επιστρέφει το ίδιο διάνυσμα τιμών με την $U_l$.

Έστω $SU(g) = U_1, \ldots, U_{k(g)}$ και έστω $SG(g) = g_1, \ldots, g_{k(g)}$. Στην περίπτωση όπου η $g$ δεν τερμάτισε στη γραμμή 10 του ψευδοκώδικα, θεωρούμε πως $SU(g) = SG(g) = \lambda$, όπου $\lambda$ είναι η κενή ακολουθία. Σημειώνεται ότι αν ισχύει $SG(g) \neq \lambda$, η συνάρτηση $g$ καθώς και οι $g_1, \ldots, g_{k(g)} \in SG(g)$ επιστρέφουν το ίδιο διάνυσμα τιμών. Με κάθε SCAN συσχετίζεται μία κλήση της $get\_vector$, οποία ονομάζεται $gv(S)$ και ορίζεται ως εξής.

Έστω ότι $gv(S) = \begin{cases} g_1, αν\ SG(g) \neq \lambda \\ g, διαφορετικά \end{cases}$

Το παρακάτω λήμμα περιγράφει τις ιδιότητες της συνάρτησης $gv(S)$, οι οποίες είναι άμεση συνεπαγωγή του ορισμού της.

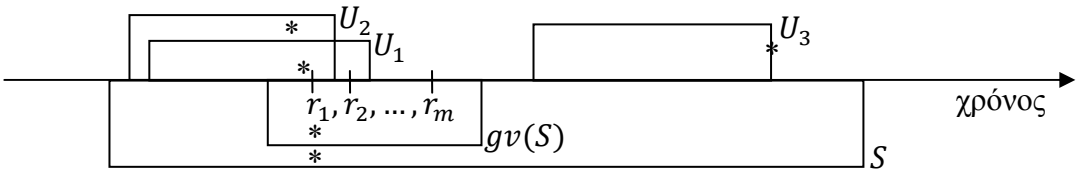
**Λήμμα 4.1** *Για κάθε SCAN S ισχύουν τα ακόλουθα:*

1. *η $gv(S)$ επιστρέφει εκτελώντας τη γραμμή 11 του ψευδοκώδικα, και*
2. *η S επιστρέφει το ίδιο διάνυσμα τιμών με αυτό που επιστρέφει η $gv(S)$.*

Για να αποδείξουμε την ορθότητα του αλγορίθμου αρχικά θα αποδώσουμε σημεία σειριοποίησης σε κάθε SCAN και UPDATE της $a$. Στη συνέχεια θα αποδείξουμε ότι το σημείο σειριοποίησης μίας οποιασδήποτε λειτουργίας της $a$ βρίσκεται εντός του διαστήματος εκτέλεσής της και εν τέλει θα αποδειχθεί ότι κάθε SCAN επιστρέφει ένα συνεπές διάνυσμα τιμών.

Σε αυτό το σημείο θα αναθέσουμε σημεία σειριοποίησης στις SCAN και UPDATE. Έστω $S$ μία οποιαδήποτε SCAN της $a$. Για λόγους ευκολίας θα εισάγουμε σημείο σειριοποίησης και στην $gv(S)$ (αν και δεν είναι απαραίτητο). Συμβολίζουμε με $r_1,\ldots,r_m$ τις εντολές read που επιτέλεσε η συνάρτηση $gv(S)$ στους καταχωρητές $pre[1],\ldots,pre[m]$, αντίστοιχα. Αναθέτουμε σημείο σειριοποίησης στην $gv(S)$ και στην $S$ στο σημείο όπου η $gv(S)$ επιτέλεσε την εντολή $r_1$ (βλ. Σχήμα 4.3). Για κάθε συνιστώσα $A_j, 1 \leq j \leq m$, όλες οι UPDATE στη συνιστώσα $A_j$ που ξεκίνησαν την εκτέλεσή τους πριν την εγγραφή του καταχωρητή $seq$ από την $S$ και τελείωσαν μεταξύ της $r_1$ και της $r_j$, σειριοποιούνται ακριβώς πριν το σημείο σειριοποίησης της $S$. Αν οι προαναφερθείσες λειτουργίες είναι περισσότερες από μία, τότε σειριοποιούνται με τη σειρά με την οποία επιτέλεσαν την write τους στον καταχωρητή $pre[j]$. Αφότου αναθέσουμε σημεία σειριοποίησης σε όλες τις SCAN, καθώς και σε μερικές UPDATE όπως περιγράφηκε παραπάνω, σειριοποιούμε τις υπόλοιπες UPDATE στο σημείο όπου επιτέλεσαν την write τους στον $pre$. Για παράδειγμα στο Σχήμα 4.3, όλες οι UPDATE $U_1, U_2, U_3$ είναι UPDATE στη συνιστώσα $A_j$. Οι $U_1, U_2$ που εκτελούν την εντολή write στον $pre[j]$ πριν την $r_j$, σειριοποιούνται ακριβώς πριν την $r_1$. Η $U_3$ επιτέλεσε την εντολή write στον $pre[j]$ έπειτα από την $r_j$, και έτσι σειριοποιείται στην write της στον $pre[j]$.

Σε αυτό το σημείο θα αποδείξουμε ότι τα σημεία σειριοποίησης κάθε SCAN ή UPDATE βρίσκονται μέσα στο διάστημα εκτέλεσής τους. Αρχικά αποδεικνύονται μερικά απλά τεχνικά λήμματα, τα οποία είναι απαραίτητα για την απόδειξη των προαναφερθέντων ισχυρισμών.




Σχήμα 4.3 Παράδειγμα σειριοποίησης λειτουργιών σε μία εκτέλεση του Linear.

**Λήμμα 4.2** *Έστω μία οποιαδήποτε SCAN S, έστω $g$ η $get\_vector$ που καλείται από την S και έστω ότι $SU(g) = U_1, \ldots, U_{k(g)} \neq \lambda$. Τότε για κάθε $1 < j \leq k(g)$, ισχύουν τα ακόλουθα:*

1. *η $U_{j-1}$ επιτελεί την write της στον $post$ πριν η $U_j$ εκτελέσει την write της στον $post$, και*
2. *η τιμή που διαβάζει η $U_{j-1}$ στον καταχωρητή $seq$ είναι ίση με αυτή που διάβασε η $U_j$ στον $seq$.*

**Απόδειξη.** Έστω $SG(g) = g_1, \ldots, g_{k(g)}$. Από τον τρόπο ορισμού της $SU(g)$ και της $SG(g)$, για κάθε $1 < j \leq k(g)$, η $g_j$ τερματίζει εκτελώντας τη γραμμή 10 του ψευδοκώδικα και επιστρέφει το διάνυσμα τιμών που έγγραψε η $U_{j-1}$ στον $post$. Έτσι η $U_{j-1}$ εκτελεί την write της στον $post$ πριν το τέλος της $g_j$ και άρα πριν η $U_j$ επιτελέσει την write της στον $post$, όπως και απαιτείται από τον ισχυρισμό 1.

Έστωσαν $seq_{j-1}$ και $seq_j$, οι τιμές που ανέγνωσαν στον καταχωρητή $seq$ οι λειτουργίες $U_{j-1}$, $U_j$. Από τον τρόπο ορισμού της $SU(g)$ και της $SG(g)$, για κάθε $1 < j \leq k(g)$, η $U_{j-1}$ εγγράφει στον $post$ το διάνυσμα τιμών που επιστράφηκε από την $g_j$. Όμως από τον ψευδοκώδικα του αλγορίθμου, η συνάρτηση $g_j$ διαβάζει μόνο τον καταχωρητή $post[seq_j]$ από τους καταχωρητές του πίνακα $post$, ενώ η $U_{j-1}$ εγγράφει μόνο τον καταχωρητή $post[seq_{j-1}]$. Έτσι θα πρέπει να ισχύει $seq_j = seq_{j-1}$, όπως και απαιτείται από τον ισχυρισμό 2. ∎

**Λήμμα 4.3** *Έστω μία οποιαδήποτε SCAN S, έστω $g$ η $get\_vector$ που καλείται από την S και έστω ότι $SU(g) = U_1, \ldots, U_{k(g)} \neq \lambda$. Για κάθε $1 \leq j \leq k(g)$, η τιμή που η $U_j$ ανέγνωσε στον καταχωρητή $seq$ εγγράφηκε από την S.*

**Απόδειξη.** Αρχικά θα αποδειχθεί ότι η $U_{k(g)}$ ανέγνωσε την τιμή που η $S$ έγγραψε στον καταχωρητή $seq$. Από τον τρόπο ορισμού της $SU(g)$, η $g$ επιστρέφει το διάνυσμα τιμών που η $U_{k(g)}$ έγγραψε στον $post$. Προφανώς η παράμετρος $curr\_seq$ με την οποία κλήθηκε η $g$ (έστω $curr\_seq_g$ αυτή) ισούται με την τιμή που έγγραψε η $S$ στον $seq$ (αφού η $S$ καλεί την $g$ με αυτή την παράμετρο). Από τον ψευδοκώδικα του αλγορίθμου, η $g$ διαβάζει μόνο τον καταχωρητή $post[curr\_seq_g]$ του πίνακα $post$. Έτσι συμπεραίνουμε πως η $U_{k(g)}$ πρέπει να έχει εγγράψει τον καταχωρητή $post[curr\_seq_g]$. Αφού ο καταχωρητής του $post$ στον οποίο η $U_{k(g)}$ εγγράφει καθορίζεται από την τιμή που η $U_{k(g)}$ ανέγνωσε στον $seq$, προκύπτει ότι η $U_{k(g)}$ ανέγνωσε στον καταχωρητή $seq$ την τιμή που έγγραψε η $S$. Από το Λήμμα 4.2 (ισχυρισμός 2) προκύπτει ότι το ίδιο ισχύει για όλες τις $U_j, 1 \leq j \leq k(g)$. ∎

**Λήμμα 4.4** *Έστω $S$ μία οποιαδήποτε SCAN και έστω $g$ η $get\_vector$ που κλήθηκε από την $S$. Για κάθε $get\_vector$ συνάρτηση $g' \in SG(g)$, το διάστημα εκτέλεσης της $g'$ ξεκινά έπειτα από την εγγραφή του καταχωρητή $seq$ από την $S$ και τελειώνει πριν το τέλος της $S$.*

**Απόδειξη.** Έστω $U'$ η UPDATE που κάλεσε την $g'$. Αφού $g' \in SG(g)$, θα ισχύει $U' \in SU(g)$. Από το Λήμμα 4.3 συνεπάγεται ότι η $U'$ ανέγνωσε στον καταχωρητή $seq$ την τιμή που έγγραψε η $S$. Έτσι η παράμετρος $curr\_seq$ με την οποία κλήθηκε η $g'$, έχει την τιμή που έγγραψε η $S$ στον καταχωρητή $seq$. Έτσι η $g'$ ξεκινά την εκτέλεσή της έπειτα από την έναρξη της $S$.

Απομένει να αποδείξουμε ότι η $g'$ τερματίζει την εκτέλεσή της πριν το τέλος του διαστήματος εκτέλεσης της $S$. Από τον ορισμό της $SU(g)$, η $g$ επιστρέφει το διάνυσμα τιμών που η $U_{k(g)}$ έγγραψε στον $post$. Έτσι η εγγραφή του $post$ από την $U_{k(g)}$ προηγείται του τέλους της $g$, και άρα του τέλους της $S$. Από το Λήμμα 4.2, ισχυρισμός 1 προκύπτει ότι η write της $U'$ στον $post$ προηγείται της write της $U_{k(g)}$ στον $post$. Έτσι προκύπτει ότι η $g'$ (που τερματίζει πριν την $U'$) τερματίζει πριν το τέλος της $S$. Άρα συμπεραίνουμε ότι το διάστημα εκτέλεσης της $g'$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$ και ξεκινά αφότου η $S$ εγγράψει τον καταχωρητή $seq$. ∎

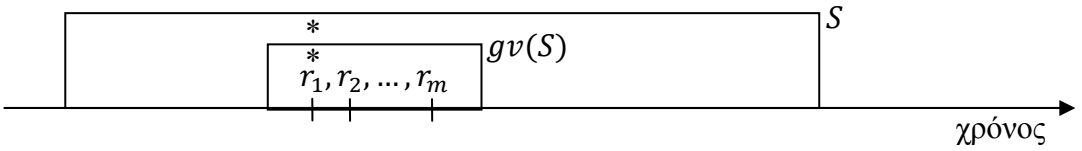
Έστω $S$ μία SCAN και έστω $g$ η $get\_vector$ που καλείται από την $S$. Από τον ψευδοκώδικα του Αλγόριθμος 4.1 (γραμμή 6), είναι προφανές ότι το διάστημα εκτέλεσης της $g$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$ (αφού η $S$ καλεί την $g$) και η $g$ ξεκινά τη λειτουργία της αφότου η $S$ εγγράψει τον καταχωρητή $seq$. Από τα προαναφερθέντα και από το Λήμμα 4.4 προκύπτει το ακόλουθο πόρισμα.

**Πόρισμα 4.5** Έστω $S$ μία οποιαδήποτε SCAN που γράφει την τιμή $seq_S$ στον καταχωρητή $seq$.Το διάστημα εκτέλεσης της $gv(S)$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$.

Σε αυτό το σημείο θα αποδειχθεί ότι το σημείο σειριοποίησης κάθε SCAN ή UPDATE βρίσκεται εντός του διαστήματος εκτέλεσής της.

**Λήμμα 4.6** *Έστω μία οποιαδήποτε εκτέλεση $a$ του αλγορίθμου Linear. Το σημείο σειριοποίησης κάθε SCAN ή UPDATE που εκτελείται στην $a$, βρίσκεται εντός των διαστήματος εκτέλεσής της.*

**Απόδειξη.** Έστω $S$ μία οποιαδήποτε SCAN που εκτελείται στην $a$ και έστω $g$ η $get\_vector$ που εκτελείται από την $S$. Υπενθυμίζεται ότι η $S$ σειριοποιείται στο ίδιο σημείο με αυτό της $gv(S)$. Από το Πόρισμα 4.5 προκύπτει ότι το διάστημα εκτέλεση της $g$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Στην περίπτωση όπου ισχύει $SG(g) = g_1, \ldots, g_{k(g)} \neq \lambda$ (οπότε $gv(S) = g_1$), από το Λήμμα 4.4 συνεπάγεται ότι το διάστημα εκτέλεσης της $g_1$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Συμπεραίνουμε ότι το διάστημα εκτέλεσης της $gv(S)$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Το σημείο σειριοποίησης της $gv(S)$ τοποθετείται στην εντολή $r_1$ της. Από τα παραπάνω προκύπτει ότι το σημείο σειριοποίησης της $S$ βρίσκεται εντός του διαστήματος εκτέλεσής της (βλ. Σχήμα 4.4), όπως και απαιτείται.




Σχήμα 4.4 Η S σειριοποιείται εντός του διαστήματος εκτέλεσής της στον Linear.

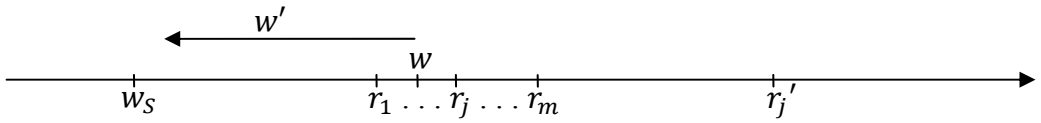
Έστω $U$ μία οποιαδήποτε UPDATE που επιτελείται στην $a$. Αν το σημείο σειριοποίησης της $U$ έχει τοποθετηθεί στην write της στον καταχωρητή $pre$, τότε προφανώς αυτό βρίσκεται εντός του διαστήματος εκτέλεσης της $U$. Ας υποθέσουμε ότι το σημείο σειριοποίησης της $U$ δεν τοποθετείται στη write της στον $pre$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S$, τέτοια ώστε το σημείο σειριοποίησης της $U$ να έχει τοποθετηθεί ακριβώς πριν το σημείο σειριοποίησης της $S$ (και άρα της $gv(S)$). Από το Λήμμα 4.1 (ισχυρισμός 1) προκύπτει ότι η $gv(S)$ έχει τερματίσει τη λειτουργία της εκτελώντας τη γραμμή 11 του ψευδοκώδικα. Έστωσαν $r_1,\ldots,r_m$ οι εντολές read στους καταχωρητές $pre[1],\ldots,pre[m]$ που επιτελέσθηκαν από την $gv(S)$. Από τον τρόπο ανάθεσης σημείων σειριοποίησης, η $U$ είναι μία UPDATE σε κάποια συνιστώσα $A_j, 1 < j \leq m$, τέτοια ώστε η $U$ να έχει ξεκινήσει τη λειτουργία της πριν την $S$ και να έχει ολοκληρωθεί μεταξύ των εντολών $r_1$ και $r_j$. Από το Πόρισμα 4.5 προκύπτει ότι το διάστημα εκτέλεσης της $gv(S)$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Έτσι η $U$ ξεκινά πριν την έναρξη της $gv(S)$ και τελειώνει έπειτα από την $r_1$. Έτσι το σημείο σειριοποίησης της $U$ (που έχει τοποθετηθεί πριν την $r_1$) βρίσκεται εντός του διαστήματος εκτέλεσής της, όπως και απαιτείται. ■

Θα αποδειχθεί στη συνέχεια ότι οι SCAN επιστρέφουν ένα συνεπές διάνυσμα τιμών. Αρχικά αποδεικνύεται ένα τεχνικό λήμμα.

**Λήμμα 4.7** *Έστω $S$ μία οποιοδήποτε SCAN και έστωσαν $r_1,\ldots,r_m$ οι εντολές read που επιτελέσθηκαν από τη συνάρτηση $gv(S)$ στους καταχωρητές $pre[1],\ldots,pre[m]$ αντίστοιχα. Έστω μια UPDATE $U$ στη συνιστώσα $A_j, 1 < j \leq m$, τέτοια ώστε να επιτελεί την write της στον καταχωρητή $pre[j]$ μεταξύ της $r_1$ και της $r_j$. Η $U$ ξεκινά την εκτέλεσή της πριν η $S$ εγγράψει τον καταχωρητή $seq$.*

**Απόδειξη.** Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $U$ ξεκινά την εκτέλεσή της έπειτα από την εγγραφή του καταχωρητή $seq$ από την $S$. Σε αυτή την περίπτωση η $U$ διαβάζει στον καταχωρητή $seq$ την τιμή που έγγραψε η $S$ (έστω $w_S$ η write της $S$ στον $seq$ και έστω $seq_S$ η τιμή που γράφει η $S$ στον $seq$). Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $U$ πρώτα εγγράφει τον καταχωρητή $post[seq_S]$ και έπειτα τον καταχωρητή $pre[j]$ (έστωσαν $w'$ και $w$ αντίστοιχα, οι εν λόγω δύο εντολές write της $U$, βλ. Σχήμα 4.5). Από το Πόρισμα 4.5 προκύ-

πτει ότι η συνάρτηση $gv(S)$ καλείται με παράμετρο $seq_S$. Από τον ψευδοκώδικα του αλγορίθμου, η εντολή read $r_j$ της $gv(S)$ στον καταχωρητή $pre[j]$ προηγείται της εντολής read $r'_j$ στον καταχωρητή $post[seq_S][j]$. Αφού η $w'$ προηγείται της $w$, η $w$ προηγείται της $r_j$ (από την υπόθεση του λήμματος) και η $r_j$ προηγείται της $r'_j$, προκύπτει ότι η $w'$ προηγείται της $r'_j$. Έτσι η $gv(S)$ διαβάζει μία τιμή διαφορετική του $null$ και επιστρέφει το διάνυσμα τιμών που εγγράφηκε στον $post[seq_S]$ εκτελώντας τη γραμμή 10 του ψευδοκώδικα, το οποίο και αντιτίθεται στο Λήμμα 4.1 (ισχυρισμός 1). ■




Σχήμα 4.5 Η $U$ έπεται της $w_S$ και η $w'$ προηγείται της $w$.

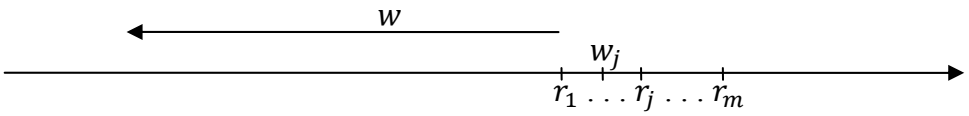
**Λήμμα 4.8** *Για κάθε SCAN το διάνυσμα τιμών που επιστρέφεται από την $gv(S)$ είναι συνεπές.*

**Απόδειξη.** Έστω $\vec{v} = \langle v_1, .., v_m \rangle$ το διάνυσμα τιμών που επιστρέφει η συνάρτηση $gv(S)$. Θα δείξουμε ότι για κάθε $j \in \{1, \ldots, m\}$, η $v_j$ είναι η τιμή της τελευταίας UPDATE στη συνιστώσα $A_j$ που έχει σειριοποιηθεί πριν την $gv(S)$. Για να καταλήξουμε σε άτοπο υποθέτουμε ότι υπάρχει ένας ακέραιος $j \in \{1, \ldots, m\}$ τέτοιος ώστε η παράμετρος της τελευταίας UPDATE $U$ στην $A_j$ που σειριοποιείται πριν την $gv(S)$ να είναι διαφορετική της $v_j$. Έστω $v$ η παράμετρος της $U$. Έστω $U_j$ η τελευταία UPDATE που έγγραψε την τιμή $v_j$ στον καταχωρητή $pre[i]$, πριν η $gv(S)$ αναγνώσει τον $pre[j]$ μέσω της εντολής $r_j$.

Από το Λήμμα 4.1 προκύπτει ότι η $gv(S)$ επιστρέφει εκτελώντας τη γραμμή 11 του ψευδοκώδικα. Ετσι η $gv(S)$ επιστρέφει τις τιμές που ανέγνωσε στους καταχωρητές $pre[1], \ldots, pre[m]$. Έστωσαν $r_1, \ldots, r_m$ οι εντολές read που επιτέλεσε η $gv(S)$ στους καταχωρητές $pre[1], \ldots, pre[m]$. Αφού η $r_j$ ανέγνωσε τιμή ίση με $v_j$, η εντολή write $w_j$ που επιτελέσθηκε από την $U_j$ είναι η τελευταία εντολή write στον καταχωρητή $pre[j]$ που προηγείται της $r_j$. Διακρίνουμε τις ακόλουθες περιπτώσεις.

1) Αρχικά υποθέτουμε ότι η $U$ επιτελεί την write της $w$ στον καταχωρητή $pre[j]$ πριν την εντολή $r_1$.

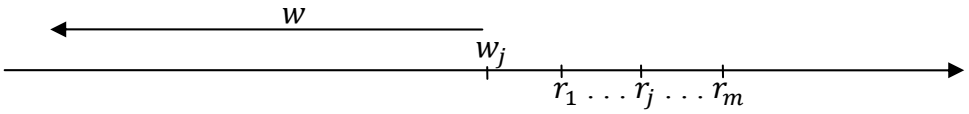
Υποθέτουμε ότι η εντολή $w_j$ έπεται της $r_1$ (βλ. Σχήμα 4.6). Υπενθυμίζουμε ότι η $w_j$ εκτελείται πριν την $r_j$. Από το Λήμμα 4.7 προκύπτει ότι η $U_j$ ξεκίνησε τη λειτουργία της πριν την εγγραφή του $seq$ από την $S$. Άρα από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $U_j$ σειριοποιείται ακριβώς πριν την $r_1$ και η $gv(S)$ σειριοποιείται στην $r_1$. Από το Λήμμα 4.6 η $U$ σειριοποιείται εντός του διαστήματος εκτέλεσής της, άρα η $U$ σειριοποιείται το αργότερο στην $w$ εντολή της. Έτσι η $U_j$ δε δύναται να έχει σειριοποιηθεί στην $w_j$ διότι σε αυτή την περίπτωση η $U_j$ πρέπει να έχει σειριοποιηθεί μεταξύ της $U$ και της $gv(S)$, το οποίο και αντιτίθεται στην υπόθεση.




Σχήμα 4.6 Η $w$ εκτελείται πριν την $r_1$ και η $w_j$ έπεται της $r_1$.

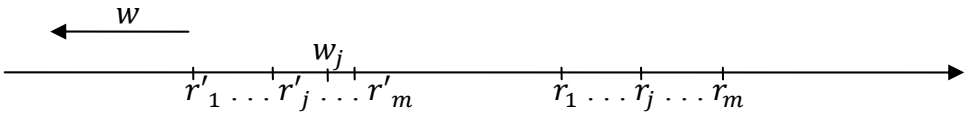
Υποθέτουμε ότι η $w_j$ προηγείται της $r_1$ (βλ. Σχήμα 4.7). Υπενθυμίζουμε ότι η $w$ προηγείται της $w_j$, έτσι η $w_j$ επιτελείται μεταξύ της $w$ και της $r_1$. Από το Λήμμα 4.6 προκύπτει ότι η $U$ σειριοποιείται το αργότερο στην $w$ εντολή της. Αφού έχουμε υποθέσει ότι η $U_j$ είναι η τελευταία λειτουργία που σειριοποιείται πριν την $S$ (η $S$ σειριοποιείται στην $r_1$), συμπεραίνουμε ότι η $U_j$ δεν σειριοποιείται στην εντολή $w_j$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι υπάρχει μία SCAN $S'$, τέτοια ώστε η $v_j$ να έχει επιστραφεί από την $S'$ και η $U_j$ να έχει σειριοποιηθεί ακριβώς πριν την $gv(S')$. Από το Λήμμα 4.1 προκύπτει ότι η $gv(S')$ επιστρέφει εκτελώντας τη γραμμή 11 του ψευδοκώδικα. Έτσι η $gv(S')$ επιστρέφει τις τιμές που ανέγνωσε στους καταχωρητές $pre[1],\ldots,pre[m]$. Έστωσαν $r'_1,\ldots,r'_m$ οι εντολές read που επιτέλεσε η $gv(S')$ στους καταχωρητές $pre[1],\ldots,pre[m]$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι η $U_j$ πρέπει να ξεκίνησε την εκτέλεσή της πριν η $S'$ επιτελέσει την write της στον $seq$ και να εκτέλεσε την τελευταία write της $w_j$ μεταξύ της $r'_1$ και της

$r'_j$. Έτσι το σημείο σειριοποίησης της $U_j$ τοποθετείται στην εντολή $r'_1$. Αφού η $w_j$ προηγείται της $r_1$, θα ισχύει $S' \neq S$.



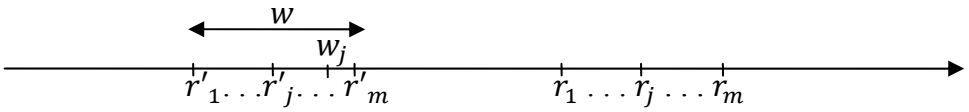

Σχήμα 4.7 Η $w_j$ προηγείται της $r_1$ και η $w$ προηγείται της $w_j$.

Έστω ότι η $w$ προηγείται της $r'_1$ (βλ. Σχήμα 4.8). Από το Λήμμα 4.6 προκύπτει ότι η $U$ σειριοποιείται το αργότερο στην $w$ και άρα η $U$ σειριοποιείται πριν την $r_1$ και πριν την $U_j$. Αυτό αντιτίθεται στην υπόθεσή μας ότι η $U$ σειριοποιείται ανάμεσα στο σημείο σειριοποίησης της $U_j$ και στην $r_1$.




Σχήμα 4.8 Η $w$ εκτελείται πριν την $r'_1$ .

Έτσι υποθέτουμε ότι η $w$ έπεται της $r'_1$ (βλ. Σχήμα 4.9). Αφού η $w$ προηγείται της $r_1$ και η $r'_1$ προηγείται της $r_1$, προκύπτει ότι η $S'$ προηγείται της $S$. Υπενθυμίζεται ότι η $r'_j$ έπεται της $w_j$, η οποία έπεται της $w$. Έτσι η $U$ εκτελεί την τελευταία λειτουργία της $w$ μεταξύ της $r'_1$ και της $r'_j$. Από το Λήμμα 4.6 προκύπτει ότι η $U$ ξεκινά την εκτέλεσή της πριν η $S'$ εγγράψει τον καταχωρητή $seq$. Έτσι από τον τρόπο ανάθεσης σημείων σειριοποίησης προκύπτει ότι η $U$ σειριοποιείται ακριβώς πριν την $r'_1$ (όπως και η $U_j$) Ωστόσο το σημείο σειριοποίησης της $U$ προηγείται του σημείου σειριοποίησης της $U_j$, αφού η $w$ προηγείται της $w_j$ (τα σημεία σειριοποίησης των $U, U_j$ εισάγονται με τη σειρά με την οποία οι εντολές write $w, w_j$ έχουν επιτελεσθεί). Έτσι η $U_j$ σειριοποιείται μεταξύ της $U$ και της $S$ (η οποία σειριοποιείται έπειτα από την $S'$), το οποίο είναι άτοπο καθώς αντιτίθεται στην υπόθεσή μας.




Σχήμα 4.9 Η $w$ εκτελείται ανάμεσα στην $r'_1$ και στην $w_j$.

2) Υποθέτουμε τώρα ότι η $w$ έπεται της $r_1$.

Αρχικά υποθέτουμε ότι η $w$ προηγείται της $r_j$. Το Λήμμα 4.6 συνεπάγεται ότι η $U$ ξεκινά την εκτέλεσή της πριν η $S$ εγγράψει τον καταχωρητή $seq$. Έτσι από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι η $U$ σειριοποιείται ακριβώς πριν την $r_1$. Υπενθυμίζεται ότι η $w_j$ είναι η τελευταία εντολή write στον καταχωρητή $pre[j]$ πριν την εντολή $r_j$, έτσι η $w_j$ θα έπεται της $w$. Αφού η $w$ έπεται της $r_1$, η $w_j$ επιτελείται μεταξύ της $r_1$ και της $r_j$. Από το Λήμμα 4.6 συνεπάγεται ότι η $U_j$ ξεκίνησε την εκτέλεσή της πριν η $S$ εγγράψει τον καταχωρητή $seq$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι και η $U_j$ σειριοποιείται ακριβώς πριν την $r_1$ (όπως και η $U$). Ωστόσο η $U$ σειριοποιείται πριν την $U_j$, αφού η $w$ προηγείται της $w_j$. Έτσι η $U_j$ σειριοποιείται μεταξύ της $U$ και της $S$, το οποίο και αντιτίθεται στην υπόθεσή μας.

Υποθέτουμε ότι η $w$ έπεται της $r_j$. Αφού η $S$ σειριοποιείται στην $r_1$ και έχουμε υποθέσει ότι η $U$ σειριοποιείται πριν την $S$, πρέπει η $U$ να μην σειριοποιείται στην $w$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι υπάρχει μία SCAN $S'$, τέτοια ώστε η $S'$ να επιστρέφει τιμή ίση με $v$ για τη συνιστώσα $A_j$ και η $U$ να σειριοποιείται ακριβώς πριν την $gv(S')$. Από το Λήμμα 4.1 προκύπτει ότι η $gv(S')$ επιστρέφει εκτελώντας τη γραμμή 11 του ψευδοκώδικα. Έτσι η $gv(S')$ επιστρέφει τις τιμές που ανέγνωσε στους καταχωρητές $pre[1], \ldots, pre[m]$. Έστωσαν $r'_1, \ldots, r'_m$ οι εντολές read που επιτέλεσε η $gv(S')$ στους καταχωρητές $pre[1], \ldots, pre[m]$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης προκύπτει ότι η $w$ επιτελείται μεταξύ της $r'_1$ και της $r'_j$. Αφού η $S$ επιστρέφει $v_j$ για τη συνιστώσα $A_j$, θα ισχύει $S \neq S'$. Αν η $S'$ έπεται της $S$, το σημείο σειριοποίησης της $U$ θα έπεται του σημείου σειριοποίησης της $U$, το οποίο και αντιτίθεται στην υπόθεσή μας. Έτσι υποθέτουμε ότι η $S'$ προηγείται της $S$.

Τότε η $r'_j$ προηγείται της $r_1$. Αφού η $w$ προηγείται της $r'_j$, προκύπτει ότι η $w$ προηγείται της $r_1$. Αυτό είναι άτοπο καθώς η $w$ έπεται της $r_1$

Άρα το λήμμα ισχύει σε κάθε περίπτωση. ■

Έστω $S$ μία οποιαδήποτε SCAN. Η $S$ σειριοποιείται στο ίδιο σημείο με αυτό της $gv(S)$ και επιστρέφει το ίδιο διάνυσμα τιμών. Το Λήμμα 4.7 συνεπάγεται ότι η $gv(S)$ επιστρέφει ένα συνεπές διάνυσμα τιμών. Έτσι συμπεραίνουμε ότι και η $S$ επιστρέφει ένα συνεπές διάνυσμα τιμών.

***Θεώρημα 4.9*** *Ο αλγόριθμος Linear είναι μία σειριοποιήσιμη, Single-Scanner υλοποίηση ατομικών στιγμιοτύπων που πληροί την ιδιότητα ελεύθερη-αναμονής και επιτυγχάνει χρονική πολυπλοκότητα $O(m)$ για τη SCAN και $O(m)$ για την UPDATE χρησιμοποιώντας $O(m+k)$ κοινούς καταχωρητές μεγέθους $O(max\{m\,log|T|, log\,k\})$ bit, όπου $k$ είναι ο μέγιστος αριθμός SCAN που μπορούν να επιτελεσθούν σε οποιαδήποτε εκτέλεση.*

# ΚΕΦΑΛΑΙΟ 5. Ο ΑΛΓΟΡΙΘΜΟΣ T-OPT



Σε αυτό το κεφάλαιο παρουσιάζεται ένας Single-Scanner αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται T-Opt (Time-Optimal). Ο T-Opt έχει βέλτιστη χρονική πολυπλοκότητα $O(m)$ για τη SCAN και $O(1)$ για την UPDATE. Ο T-Opt χρησιμοποιεί $O(mk)$ καταχωρητές, όπου $k$ είναι ο μέγιστος αριθμός SCAN που πρόκειται να επιτελεσθούν. Από τους καταχωρητές που χρησιμοποιεί T-Opt, ο μεγαλύτερος σε μέγεθος είναι ο καταχωρητής $seq$ που αποτελείται από $O(\max\{\log k, \log|T|\})$ bit.

## 5.1. Περιγραφή του αλγορίθμου

Ο αλγόριθμος T-Opt (Αλγόριθμος 5.1) χρησιμοποιεί ένα πίνακα $pre$, ο οποίος περιέχει $m$ καταχωρητές έναν για κάθε συνιστώσα του αντικειμένου. Κάθε UPDATE στη συνιστώσα $A_i$ αποθηκεύει την τιμή της στον καταχωρητή $pre[i]$ (γραμμή 6). Πριν συμβεί αυτό η UPDATE αποθηκεύει την τρέχουσα τιμή του $pre[i]$ σε κάποιον καταχωρητή του πίνακα $post$ (γραμμή 5), έτσι ώστε να «βοηθήσει» τη SCAN να επιστρέψει ένα συνεπές διάνυσμα τιμών. Ο $post$ είναι ένας διδιάστατος πίνακας, όπου κάθε γραμμή του περιέχει $m$ καταχωρητές. Ο αριθμός των γραμμών του $post$ εξαρτάται από τον μέγιστο αριθμό των SCAN $k$ που πρόκειται να εκτελεστούν σε οποιαδήποτε εκτέλεση (και έτσι ο αριθμός των γραμμών δύναται να είναι μη πεπερασμένος).

Κάθε φορά που ξεκινάει μία SCAN θέτει έναν νέο σειριακό αριθμό στον $seq$, αυξάνοντας την τιμή του $seq$ κατά 1 (γραμμή 7). Κάθε UPDATE στη συνιστώσα $A_i$ ξεκινά την εκτέλεσή της διαβάζοντας έναν σειριακό αριθμό που είναι αποθηκευμένος στον $seq$ (γραμμή 1). Ο σειριακός αριθμός που αναγνώσθηκε, χρησιμοποιείται από

την $U$ για τη δεικτοδότηση της κατάλληλης γραμμής του $post$, όπου σε κάποιον καταχωρητή αυτής της γραμμής η $U$ θα αποθηκεύσει την τιμή του $pre[i]$ προτού εγγράψει σε αυτόν την νέα τιμή της συνιστώσας $A_i$ (γραμμή 6). Για να υπολογίσει μία SCAN $S$ ένα συνεπές διάνυσμα τιμών, πρέπει να παραβλέψει τις τιμές που εγγράφηκαν από UPDATE που ξεκίνησαν την εκτέλεσή τους έπειτα από την $S$. Για να επιτευχθεί αυτό, η $S$ διαβάζει τους $m$ καταχωρητές του $pre$ (γραμμή 9) και τους $m$ καταχωρητές του $post[seq_S]$ (γραμμή 10), όπου $seq_S$ είναι η τιμή που εγγράφηκε από την $S$ στον $seq$. Οι UPDATE που ξεκίνησαν την εκτέλεσή τους κατά το διάστημα εκτέλεσης της $S$, εγγράφουν σε κάποιον καταχωρητή της γραμμής $seq_S$ του πίνακα $post$. Έτσι αν ισχύει $post[seq_S][i] \neq null$, $i \in \{1, \dots, m\}$, η $S$ πρέπει να επιστρέψει για τη συνιστώσα $A_i$ την παλιά τιμή του $pre[i]$ που είχε αποθηκευθεί στον $post[seq_S][i]$ (γραμμή 14), αφού η τρέχουσα τιμή του έχει εγγραφεί από κάποια UPDATE που ξεκίνησε την εκτέλεσή της μετά την $S$. Οι UPDATE που έχουν ξεκινήσει την εκτέλεσή τους πριν την $S$ έχουν γράψει σε μικρότερες γραμμές του $post$. Αν η $S$ αναγνώσει την τιμή που εγγράφηκε στον καταχωρητή $pre[i]$ από μία τέτοια UPDATE, θα την συμπεριλάβει στο διάνυσμα τιμών που θα επιστρέψει.

### 5.2. Χρονική και χωρική πολυπλοκότητα

Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι η χρονική πολυπλοκότητα της UPDATE είναι $O(1)$, ενώ η χρονική πολυπλοκότητα της SCAN είναι $O(m)$. Οι πολυπλοκότητες αυτές είναι βέλτιστες.

Όλοι οι καταχωρητές εκτός του καταχωρητή $seq$ αποθηκεύουν μία τιμή του συνόλου $T$. Άρα το μέγεθος όλων αυτών των καταχωρητών είναι $O(\log|T|)$ bit. Το μέγεθος του καταχωρητή $seq$ είναι $O(\log k)$, όπου $k$ είναι ο μέγιστος αριθμός SCAN που επιτελούνται σε κάθε εκτέλεση. Δηλαδή το μέγεθος του καταχωρητή $seq$ εξαρτάται από το πλήθος των SCAN που λαμβάνουν χώρα σε κάθε εκτέλεση. Το πλήθος των καταχωρητών που χρησιμοποιούνται από τον T-Opt είναι $O(mk)$, το οποίο μπορεί να είναι μη πεπερασμένο αφού δύναται να επιτελεσθεί μη πεπερασμένος αριθμός SCAN σε μία εκτέλεση. Ωστόσο είναι δυνατό να υλοποιήσουμε τον T-Opt πιο αποτελεσματικά σε συστήματα που υποστηρίζουν ανακύκλωση κοινής μνήμης, χρησιμοποιώντας

τις ίδιες τεχνικές με αυτές που χρησιμοποιήθηκαν στον Linear. Ψευδοκώδικας για αυτή την βελτιωμένη εκδοχή του αλγορίθμου T-Opt παρουσιάζεται στον Αλγόριθμος 5.2.

Αλγόριθμος 5.1 Ψευδοκώδικας για τον αλγόριθμο T-Opt.

```
 shared data post[1..k][1..m]={null, ..., null};
 shared data pre[1..m]={null, ..., null};
 shared int seq=1;

 void UPDATE(data v, int i){
    int curr_seq;
    data d1, d2;

1.   curr_seq=seq;
2.   d1=pre[i];
3.   d2=post[curr_seq][i];
4.   if(d2==null)
5.     post[curr_seq][i]=d1;
6.   pre[i]=v;
 }

 data[] SCAN(){
   data view[1..m], d1, d2;
   int j;

7.   seq=seq+1;
8.   for(i=1;i≤m;i++){
9.     d1=pre[i];
10.    d2=post[seq][i];
11.    if(d2==null)
12.      view[i]=d1;
13.    else
14.      view[i]=d2;
   }
   return view;
 }
```

### 5.3. Απόδειξη της σειριοποιησιμότητας του αλγορίθμου

Σε αυτή την ενότητα θα αποδειχθεί η σειριοποιησιμότητα του T-Opt (Αλγόριθμος 5.1). Για την απόδειξη της σειριοποιησιμότητας είναι χρήσιμος ο ακόλουθος συμβολισμός. Έστω $a$ μία οποιαδήποτε εκτέλεση του T-Opt και έστω $S$ μία οποιαδήποτε SCAN στην $a$. Συμβολίζουμε με $w_S$ την εντολή εγγραφής του $seq$ από την $S$ στη γραμμή 7 του ψευδοκώδικα, ενώ με $seq_S$ συμβολίζουμε την τιμή που ενέγραψε η $S$ στον $seq$. Για κάθε $i \in \{1,\ldots,m\}$, συμβολίζουμε με $r_i^S$ την εντολή read της $S$ στον $pre[i]$ (γραμμή 9), ενώ με $\tilde{r}_i^S$ συμβολίζουμε την εντολή read της $S$ στον

$post[seq_S][i]$ (γραμμή 10). Έστω ότι η $S$ επιστρέφει τιμή ίση με $v_i$ για τη συνιστώσα $A_i$,$1 \leq i \leq m$. Αν η $S$ έχει αναγνώσει τιμή ίση με $null$ στον $post[seq_S][i]$ (και την τιμή $v_i$ στον καταχωρητή $pre[i]$) συμβολίζουμε με $U_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία write στον $pre[i]$ πριν την εκτέλεση της $r_i^S$. Αν η $S$ έχει αναγνώσει τιμή ίση με $v_i$ στον $post[seq_S][i]$, χρησιμοποιείται ο ακόλουθος συμβολισμός. Συμβολίζουμε με $V_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον $post[seq_S][i]$ και η write της στον $post[seq_S][i]$ είναι η τελευταία write στον εν λόγω καταχωρητή που προηγείται της $\tilde{r}_i^S$. Από τον ψευδοκώδικα του αλγορίθμου μπορούμε εύκολα να συμπεράνουμε ότι η $V_i^S$ πρέπει να έχει αναγνώσει τιμή ίση με $v_i$ στον $pre[i]$. Συμβολίζουμε με $U_i^S$ την UPDATE στη συνιστώσα $A_i$ που εγγράφει την τιμή $v_i$ στον $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία εντολή write πριν η $V_i^S$ αναγνώσει τον εν λόγω καταχωρητή. Σε κάθε περίπτωση συμβολίζουμε με $w_i^S$ την write που επιτελεί η $U_i^S$ στον $pre[i]$.

Για να αποδείξουμε την ορθότητα του αλγορίθμου αρχικά θα αποδώσουμε σημεία σειριοποίησης σε κάθε SCAN και UPDATE της $a$. Στη συνέχεια θα αποδείξουμε ότι το σημείο σειριοποίησης μίας οποιασδήποτε λειτουργίας της $a$ βρίσκεται εντός του διαστήματος εκτέλεσής της και εν τέλει θα αποδειχθεί ότι κάθε SCAN επιστρέφει ένα συνεπές διάνυσμα τιμών. Σε αυτό το σημείο αποδίδουμε σημεία σειριοποίησης στις SCAN και UPDATE της $a$.

- Αναθέτουμε σε κάθε SCAN σημείο σειριοποίησης, στο σημείο όπου επιτέλεσε την $w_S$ εντολή της.
- Για κάθε $i \in \{1, \ldots, m\}$, αν η $w_i^S$ επιτελέσθηκε έπειτα από την $w_S$, τοποθετούμε το σημείο σειριοποίησης της $U_i^S$ ακριβώς πριν την $w_S$. Ακόμη για κάθε UPDATE στη συνιστώσα $A_i$ που επιτέλεσε την εντολή write της ανάμεσα στην $w_S$ και στην $w_i^S$, τοποθετούμε σημείο σειριοποίησης ακριβώς πριν την $w_S$. Τα σημεία σειριοποίησης των προαναφερθεισών λειτουργιών τοποθετούνται με τη σειρά με την οποία επιτελέσθηκαν οι εντολές write τους στον $pre[i]$.

- Αφότου αναθέσουμε σημεία σειριοποίησης σε όλες τις SCAN και σε κάποιες UPDATE (με τον τρόπο που περιγράφηκε παραπάνω), αναθέτουμε σημείο σειριοποίησης στις υπόλοιπες UPDATE στη εντολή write τους στον $pre[i]$.

Αλγόριθμος 5.2 Ψευδοκώδικας για την βελτιωμένη έκδοση του T-Opt.

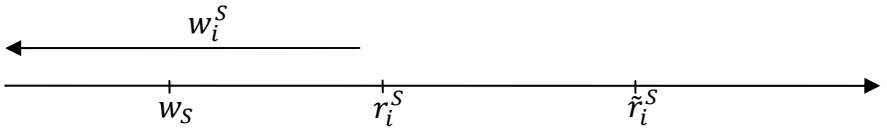
```
   shared data post[k][m]={null, ..., null};
   shared data pre[m]={null, ..., null};
   shared pointer sptr[]=new data[m];

   void UPDATE(data v, int i){
      data lptr[];
      data d1, d2;

1.    lptr=sptr;
2.    d1=pre[i];
3.    d2=post[curr_seq][i];
4.    if(d2==null)
5.      lptr[i]=d1;
6.    pre[i]=v;
   }

   data[] SCAN(){
      data view[m], d1, d2;
      int j;

7.    sptr=new data[m];
8.    for(j=1;j≤m;j++){
9.      d1=pre[i];
10.     d2=sptr[j];
11.     if(d2==null)
12.       view[j]=d1;
13.     else
14.       view[j]=d2;
15.   }
      return view;
}
```

Στη συνέχεια θα αποδειχθεί ότι το σημείο σειριοποίησης οποιασδήποτε λειτουργίας βρίσκεται εντός του διαστήματος εκτέλεσής της. Αρχικά αποδεικνύουμε μερικά τεχνικά λήμματα.

**Λήμμα 5.1** *Έστω S μία οποιαδήποτε SCAN. Για κάθε* $i \in \{1,..,m\}$*, η εντολή* $\tilde{r}_i^S$ *έπεται της εντολής* $w_i^S$.

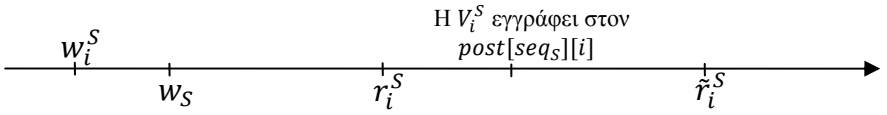
**Απόδειξη.** Αν η $w_i^S$, $i \in \{1, \dots, m\}$, προηγείται της $w_S$ το λήμμα ισχύει καθώς η $S$ επιτελεί πρώτα την $w_S$ και έπειτα την $\tilde{r}_i^S$. Έστω ότι η $w_i^S$ έπεται της $w_S$. Διακρίνουμε τις δύο ακόλουθες περιπτώσεις:

1) Έστω ότι η $S$ διάβασε τιμή ίση με $null$ στον $post[seq_S][i]$ και τιμή ίση με $v_i$ στον $pre[i]$. Από τον τρόπο ορισμού της $U_i^S$, η $w_i^S$ εγγράφει την τιμή που η $S$ ανέγνωσε (στον $pre[i]$) επιτελώντας την $r_i^S$. Έτσι η $w_i^S$ προηγείται της $r_i^S$ (βλ. Σχήμα 5.1). Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι η $\tilde{r}_i^S$ έπεται της $r_i^S$ (βλ. γραμμές 9-10). Έτσι η $\tilde{r}_i^S$ έπεται της $w_i^S$.




Σχήμα 5.1 Η $w_i^S$ προηγείται της $r_i^S$.

2) Έστω ότι η $S$ ανέγνωσε την τιμή $v_i$ στον $post[seq_S][i]$. Σε αυτή την περίπτωση η $V_i^S$ είναι καλά ορισμένη. Από τον τρόπο ορισμού της $V_i^S$ και της $U_i^S$ ισχύουν τα ακόλουθα: (1) η $V_i^S$ διαβάζει στον $pre[i]$ την τιμή που έγγραψε η $w_i^S$ και έτσι η $V_i^S$ διαβάζει τον $pre[i]$ έπειτα από την $w_i^S$, και (2) η $\tilde{r}_i^S$ διαβάζει στον $post[seq_S][i]$ την τιμή που εγγράφηκε από την $V_i^S$ και έτσι η $\tilde{r}_i^S$ έπεται της write που επιτέλεσε η $V_i^S$ στον $post[seq_S][i]$ (βλ. Σχήμα 5.2). Από τον ψευδοκώδικα του αλγορίθμου, η write της $V_i^S$ στον $post[seq_S][i]$ έπεται της read της $V_i^S$ στον ίδιο καταχωρητή, η οποία έπεται της $r_i^S$ στον $pre[i]$. Οπότε ισχύει ότι η $\tilde{r}_i^S$ έπεται της $w_i^S$.




Σχήμα 5.2 Η $V_i^S$ ξεκινά έπειτα από την $w_i^S$ και εγγράφει στον καταχωρητή $post[seq_S][i]$ πριν την $\tilde{r}_i^S$.

Σε κάθε περίπτωση η $\tilde{r}_i^S$ έπεται της $w_i^S$, όπως απαιτείται από το λήμμα. ∎

**Λήμμα 5.2** *Έστω ότι η $S$ διαβάζει την τιμή $v_i \neq null$ στον $post[seq_S][i]$, και έστω $r_{pre}$ η εντολή read στον $pre[i]$ που εκτελέστηκε από την $V_i^S$. Η $r_{pre}$ έπεται της $w_S$.*

**Απόδειξη.** Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $r_{pre}$ εκτελείται πριν την $w_S$. Η $V_i^S$ θα έχει αναγνώσει τον καταχωρητή $seq$ (μέσω της $r_{seq}$) πριν την επιτέλεση της $w_S$. Από τον ψευδοκώδικα του αλγορίθμου είναι ξεκάθαρο ότι η τιμή του καταχωρητή $seq$ αυξάνεται κάθε φορά που εκτελείται μία SCAN. Αφού σε κάθε χρονική στιγμή στο σύστημα εκτελείται μία μόνο SCAN, προκύπτει ότι η $r_{seq}$ που επιτελείται πριν από την $w_S$ διαβάζει μία τιμή $t < seq_S$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $V_i^S$ εγγράφει την τιμή $v_i$ στον $post[t][i]$. Αυτό είναι άτοπο καθώς εξ ορισμού η $V_i^S$ εγγράφει την τιμή $v_i$ στον $post[seq_S][i] \neq post[t][i]$. ∎

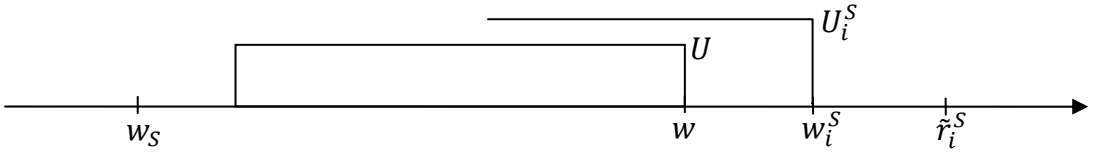
Έστω $S$ μια οποιαδήποτε SCAN τέτοια ώστε η $w_i^S$ έπεται της $w_S$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, όλες οι UPDATE που εκτελούν τη write τους στον $pre[i]$ μεταξύ της $w_S$ και της $w_i^S$, σειριοποιούνται στη $w_S$. Στη συνέχεια αποδεικνύουμε ότι κάθε τέτοια UPDATE ξεκινά την εκτέλεσή της πριν την $w_S$. Αυτό μας επιτρέπει να συμπεράνουμε (Λήμμα 5.4) ότι το σημείο σειριοποίησης κάθε τέτοιας UPDATE είναι εντός του διαστήματος εκτέλεσής της.

**Λήμμα 5.3** *Έστω μία οποιαδήποτε SCAN $S$. Για κάθε $i \in \{1, \ldots, m\}$ τέτοιο ώστε η $w_i^S$ να έπεται της $w_S$, ισχύει ότι κάθε UPDATE στη συνιστώσα $A_i$ που εγγράφει τιμή στον $pre[i]$ μεταξύ των εκτελέσεων της $w_S$ και της $w_i^S$ (με την $U_i^S$ να συμπεριλαμβάνεται), ξεκινά την εκτέλεσή της πριν την $w_S$.*

**Απόδειξη.** Υποθέτουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι υπάρχει μία UPDATE $U$ στη συνιστώσα $A_i$ που ξεκινά την εκτέλεσή της έπειτα από την $w_S$ και επιτελεί την write της στον $pre[i]$ (έστω $w$ η εν λόγω write) μεταξύ της $w_S$ και της $w_i^S$ (βλ. Σχήμα 5.3). Από το Λήμμα 5.1, η $w_i^S$ προηγείται της $\tilde{r}_i^S$ και έτσι η $U$ τελειώνει την εκτέλεσή της πριν το τέλος της $S$. Αφού η $U$ ξεκινά την εκτέλεσή της έπειτα από την $w_S$, η $U$ θα αναγνώσει τιμή ίση με $seq_S$ στον $seq$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $U$ πρώτα διαβάζει τον $pre[i]$ και έπειτα τον

$post[seq_S][i]$. Επιπρόσθετα αν η $U$ αναγνώσει τιμή ίση με $null$ στον $post[seq_S][i]$, εγγράφει σε αυτόν την τιμή που διάβασε στον $pre[i]$.

Αφού η $w_i^S$ προηγείται της $\tilde{r}_i^S$ και η $U$ τερματίζει πριν την $w_i^S$, η εκτέλεση των γραμμών 4-5 από την $U$ προηγείται της εκτέλεσης της εντολής $\tilde{r}_i^S$. Άρα η $\tilde{r}_i^S$ διαβάζει τιμή διάφορη του $null$ στον $post[seq_S][i]$ και έτσι η $V_i^S$ είναι καλά ορισμένη. Από τους ορισμούς για την $V_i^S$ και την $w_i^S$ ισχύουν τα ακόλουθα: (1) η $V_i^S$ διαβάζει $null$ στον $post[seq_S][i]$ (γραμμή 3) και $seq_S$ στον καταχωρητή $seq$ (αφού εγγράφει στον $post[seq_S][i]$) και (2) η εντολή read της $V_i^S$ στον $pre[i]$ έπεται της $w_i^S$, αφού η $V_i^S$ διαβάζει στον καταχωρητή $pre[i]$ την τιμή που ενέγραψε σε αυτόν η $w_i^S$. Έτσι προκύπτει ότι η εντολή read της $V_i^S$ στον $post[seq_S][i]$ (από τον ψευδοκώδικα η εν λόγω read έπεται της read της $V_i^S$ στον $pre[i]$) έπεται της εκτέλεσης των γραμμών 4-5 και της πιθανής εγγραφής στον $post[seq_S][i]$ από την $U$. Έτσι η $V_i^S$ διαβάζει μια τιμή διάφορη του $null$ στον $post[seq_S][i]$, το οποίο και είναι άτοπο. ■




Σχήμα 5.3 Η $U$ ξεκινά την εκτέλεσή της έπειτα από την $w_S$.

**Λήμμα 5.4** *Έστω $\alpha$ μία οποιαδήποτε εκτέλεση του T-Opt. Το σημείο σειριοποίησης κάθε SCAN και κάθε UPDATE βρίσκεται εντός του διαστήματος εκτέλεσής της.*

**Απόδειξη.** Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, κάθε SCAN σειριοποιείται εντός του διαστήματος εκτέλεσής της. Το ίδιο ισχύει και για τις UPDATE που σειριοποιούνται στην write τους στον $pre$.

Έστω $U$ μια από τις υπόλοιπες UPDATE στη συνιστώσα $A_i$. Από το Λήμμα 5.3 προκύπτει ότι η $U_i^S$ ξεκινά την εκτέλεσή της πριν την $w_S$ και τελειώνει την εκτέλεσή της έπειτα από την $w_S$. Έτσι η $U$ σειριοποιείται εντός του διαστήματος εκτέλεσής της. ■

Στη συνέχεια αποδεικνύεται ότι κάθε SCAN επιστρέφει ένα συνεπές διάνυσμα τιμών. Αρχικά αποδεικνύεται ότι για κάθε συνιστώσα $A_i, 1 \leq i \leq m$, τα σημεία σειριοποίη-

σης όλων των UPDATE στην $A_i$ σέβονται τη διάταξη που καθορίζεται από την εκτέλεση των write τους στον $pre[i]$.

**Λήμμα 5.5** *Έστωσαν $U_1$, $U_2$ δύο UPDATE σε κάποια συνιστώσα $A_i$, $1 \leq i \leq m$. Έστω ότι $w_1$ είναι η εντολή write στον $pre[i]$ από τη λειτουργία $U_1$ και $w_2$ είναι η εντολή write στον $pre[i]$ από τη λειτουργία $U_2$. Αν η $w_1$ προηγείται της $w_2$, τότε το σημείο σειριοποίησης της $U_1$ προηγείται του σημείου σειριοποίησης της $U_2$.*

**Απόδειξη.** Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι ο ισχυρισμός του λήμματος δεν ισχύει. Αν οι λειτουργίες $U_1$, $U_2$ σειριοποιούνται στην εντολή write τους στον $pre[i]$, τότε προφανώς ο ισχυρισμός του λήμματος ισχύει. Έτσι υποθέτουμε ότι τουλάχιστον μία από τις λειτουργίες $U_1$, $U_2$ δε σειριοποιείται στην εντολή write της στον $pre[i]$. Διακρίνουμε τις ακόλουθες περιπτώσεις:

1) Η $U_2$ σειριοποιείται στην $w_2$. Από το Λήμμα 5.4 προκύπτει ότι η $U_1$ σειριοποιείται εντός του διαστήματος εκτέλεσής της, έτσι η $U_1$ σειριοποιείται το αργότερο στο σημείο όπου εκτελείται η $w_1$ εντολή της. Αφού η $w_1$ προηγείται της $w_2$, η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο.
2) Η $U_1$ σειριοποιείται στην $w_1$. Αφού έχουμε υποθέσει ότι η $U_2$ έχει σειριοποιηθεί πριν την $U_1$, η $U_2$ δε σειριοποιείται στην $w_2$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S$ τέτοια ώστε η $w_2$ να έχει επιτελεσθεί μεταξύ της $w_S$ και της $w_i^S$. Αφού η $w_1$ προηγείται της $w_2$, η $w_1$ έχει επιτελεσθεί πριν την $w_i^S$. Αν η $w_1$ έπεται της $w_S$, η $U_1$ και η $U_2$ σειριοποιούνται ακριβώς πριν την $w_S$ με τη σειρά με την οποία επιτέλεσαν τις write τους στον $pre[i]$. Έτσι η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο. Έτσι υποθέτουμε ότι η $w_1$ προηγείται της $w_S$. Από το Λήμμα 5.4 προκύπτει ότι η $U_1$ σειριοποιείται το αργότερο στο σημείο όπου επιτελείται η $w_1$. Επίσης από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $U_2$ σειριοποιείται ακριβώς πριν την $w_S$. Άρα η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο.
3) Καμία από τις λειτουργίες $U_1$, $U_2$ δε σειριοποιείται στην write της στον $pre[i]$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχουν δύο SCAN $S_1$ και $S_2$, τέτοιες ώστε η $w_1$ να έχει επιτελεσθεί ανάμεσα στην $w_{S_1}$ και στην $w_i^{S_1}$, και η $w_2$ να έχει επιτελεσθεί ανάμεσα στην $w_{S_2}$ και στην $w_i^{S_2}$. Επιπρόσθετα η $U_1$ σει-

ριοποιείται ακριβώς πριν την $w_{S_1}$, ενώ η $U_2$ σειριοποιείται ακριβώς πριν την $w_{S_2}$. Από το Λήμμα 5.1 συνεπάγεται ότι η $w_i^{S_1}$ προηγείται του τέλους της $S_1$, ενώ η $w_i^{S_2}$ προηγείται του τέλους της $S_2$.

Αν $S_1 = S_2$, τότε οι $U_1$, $U_2$ σειριοποιούνται ακριβώς πριν την $w_{S_1} = w_{S_2}$ με τη σειρά με την οποία επιτέλεσαν τις write τους στον $pre[i]$. Έτσι η $U_1$ σειριοποιείται ακριβώς πριν την $U_2$, το οποίο και είναι άτοπο.

Αν η $S_1$ προηγείται της $S_2$, το σημείο σειριοποίησης της $U_1$ που έχει τοποθετηθεί ακριβώς πριν την $w_{S_1}$, προηγείται του σημείου σειριοποίησης της $U_2$ που έχει τοποθετηθεί ακριβώς πριν την $w_{S_2}$. Άρα και σε αυτή την περίπτωση έχουμε άτοπο.

Έστω ότι η $S_1$ έπεται της $S_2$. Υπενθυμίζεται ότι η $w_i^{S_1}$ εκτελείται έπειτα την $w_{S_1}$ και πριν το τέλος της εκτέλεσης της $S_1$. Ομοίως η $w_i^{S_2}$ εκτελείται έπειτα από την $w_{S_2}$ και πριν το τέλος της εκτέλεσης της $S_2$. Έτσι η $w_1$ που εκτελείται ανάμεσα στις $w_{S_1}$ και $w_i^{S_1}$, έπεται της $w_2$ που εκτελείται ανάμεσα στην $w_{S_2}$ και την $w_i^{S_2}$, το οποίο και είναι άτοπο.

Σε όλες τις περιπτώσεις οδηγηθήκαμε σε άτοπο. Έτσι δύναται να συμπεράνουμε ότι το σημείο σειριοποίησης της $U_1$ προηγείται του σημείου σειριοποίησης της $U_2$, όπως και απαιτείται από τον ισχυρισμό του λήμματος. ■

Χρησιμοποιώντας το Λήμμα 5.5, αποδεικνύουμε στη συνέχεια ότι κάθε SCAN επιστρέφει ένα συνεπές διάνυσμα τιμών.

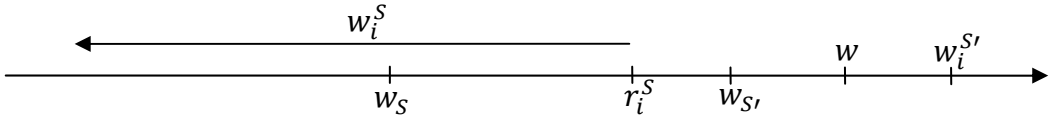
**Λήμμα 5.6** *Έστω $a$ μία οποιαδήποτε εκτέλεση του T-Opt. Κάθε SCAN που επιτελείται στην $a$ επιστρέφει ένα συνεπές διάνυσμα τιμών.*

**Απόδειξη.** Έστω ότι η $S$ επιστρέφει το διάνυσμα τιμών $\vec{v} = \langle v_1, \ldots, v_m \rangle$. Υπενθυμίζεται ότι για κάθε $i \in \{1, \ldots, m\}$, η $U_i^S$ εγγράφει την τιμή $v_i$ στον $pre[i]$ και έτσι χρησιμοποιεί την $v_i$ ως παράμετρο. Στην περίπτωση όπου η $w_i^S$ προηγείται της $w_S$, το Λήμμα 5.4 συνεπάγεται ότι η $U_i^S$ σειριοποιείται πριν την $S$. Στην περίπτωση όπου η $w_i^S$ έπεται της $w_S$, από τον τρόπο ανάθεσης των σημείων σειριοποίησης, το σημείο

σειριοποίησης της $U_i^S$ προηγείται του σημείου σειριοποίησης της $S$. Θα αποδείξουμε ότι δεν υπάρχει κάποια UPDATE στη συνιστώσα $A_i$ που να σειριοποιείται ανάμεσα στην $U_i^S$ και στην $S$, και άρα η $S$ επιστρέφει συνεπή τιμή για τη συνιστώσα $A_i$.

Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι υπάρχει ένας ακέραιος $i \in \{1,..,m\}$, τέτοιος ώστε η τελευταία UPDATE στη συνιστώσα $A_i$ που έχει σειριοποιηθεί πριν την $S$ να μην είναι η $U_i^S$. Συμβολίζουμε με $U$ την εν λόγω UPDATE και με $w$ την write της $U$ στον $pre[i]$. Διακρίνουμε τις ακόλουθες περιπτώσεις:

1) Έστω ότι η $S$ διαβάζει τιμή ίση με $null$ στον $post[seq_S][i]$ (και την τιμή $v_i$ στον $pre[i]$). Στην περίπτωση όπου η $w$ προηγείται της $w_i^S$, από το Λήμμα 5.5 προκύπτει ότι η $U$ σειριοποιείται πριν την $U_i^S$, το οποίο και είναι άτοπο. Έτσι υποθέτουμε ότι η $w$ έπεται της $w_i^S$. Από τον τρόπο ορισμού της $w_i^S$, η $r_i^S$ διαβάζει την τιμή που η $w_i^S$ εγγράφει στον $pre[i]$. Έτσι η εντολή $w$ πρέπει να έπεται της $r_i^S$ (βλ. Σχήμα 5.4). Αφού η $U$ σειριοποιείται πριν την $S$, και η $S$ σειριοποιείται στην $w_S$ εντολή, η $U$ δε θα σειριοποιηθεί στην $w$ εντολή της. Οπότε υπάρχει μία SCAN $S'$, τέτοια ώστε η $w$ να έχει επιτελεσθεί ανάμεσα στην $w_{S'}$ και στην $w_i^{S'}$, και η $U$ να σειριοποιείται ακριβώς πριν την $w_{S'}$. Αφού η $w$ έπεται της $w_i^S$, προκύπτει ότι $S \neq S'$. Από το Λήμμα 5.1 προκύπτει ότι η $w_i^{S'}$ προηγείται του τέλους του διαστήματος εκτέλεσης της $S'$. Αν η $S'$ προηγείται της $S$, τότε η $w$ που επιτελείται μεταξύ της $w_{S'}$ και της $w_i^{S'}$, προηγείται της $w_S$ και άρα και της $S$, το οποίο και είναι άτοπο. Αν η $S'$ έπεται της $S$, το σημείο σειριοποίησης της $U$ (το οποίο τοποθετείται στην $w_{S'}$) έπεται του σημείου σειριοποίησης της $S$ (το οποίο τοποθετείται στην $w_S$). Αυτό είναι άτοπο καθώς έχουμε υποθέσει πως το σημείο σειριοποίησης της $U$ προηγείται του σημείου σειριοποίησης της $S$.

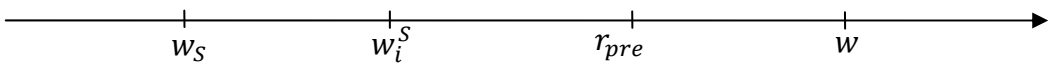



Σχήμα 5.4 Η $S$ διαβάζει $null$ στον $post[seq_S][i]$ και η $w$ έπεται της $w_i^S$.

2) Έστω ότι η $S$ διαβάζει την τιμή $v_i \neq null$ στον $post[seq_S][i]$. Σε αυτή την περίπτωση η $V_i^S$ είναι καλά ορισμένη. Συμβολίζουμε με $r_{pre}$ την εντολή read της $V_i^S$ στον $pre[i]$. Από τον τρόπο ορισμού της $V_i^S$ και της $U_i^S$, η $S$ επιστρέφει την τιμή την οποία η $U_i^S$ χρησιμοποίησε ως παράμετρο, δηλαδή η $S$ επιστρέφει την τιμή που η $U_i^S$ έχει εγγράψει στον $pre[i]$. Στην περίπτωση όπου η $w$ προηγείται της $w_i^S$, από το Λήμμα 5.5 προκύπτει ότι η $U$ σειριοποιείται πριν την $U_i^S$, το οποίο και είναι άτοπο. Έτσι υποθέτουμε ότι η $w$ έπεται της $w_i^S$. Σε αυτή την περίπτωση η $w$ πρέπει να έπεται της $r_{pre}$, καθώς η $V_i^S$ (από τον ορισμό της $U_i^S$) διαβάζει την τιμή που η $w_i^S$ εγγράφει στον καταχωρητή $pre[i]$ (βλ. Σχήμα 5.5). Από το Λήμμα 5.2 προκύπτει ότι η $r_{pre}$ έπεται της $w_S$, άρα η $w$ έπεται της $w_S$. Αφού η $U$ σειριοποιείται πριν την $S$, και η $S$ σειριοποιείται στην $w_S$, η $U$ δε δύναται να σειριοποιηθεί στην $w$. Έτσι υπάρχει μία SCAN $S'$, τέτοια ώστε η $w_i^{S'}$ να έπεται της $w_{S'}$, η $w$ να έχει επιτελεσθεί ανάμεσα στην $w_{S'}$ και στην $w_i^{S'}$, και η $U$ να έχει σειριοποιηθεί ακριβώς πριν την $w_{S'}$. Αφού η $w$ επιτελείται έπειτα από την $w_i^S$, θα ισχύει $S' \neq S$.

   Αν η $S'$ έπεται της $S$, το σημείο σειριοποίησης της $U$, το οποίο έχει τοποθετηθεί στην $w_{S'}$, έπεται του σημείου σειριοποίησης της $S$, το οποίο έχει τοποθετηθεί στην $w_S$. Άρα σε αυτή την περίπτωση έχουμε άτοπο.

   Αν η $S'$ προηγείται της $S$, το Λήμμα 5.1 συνεπάγεται ότι η $w_i^{S'}$ προηγείται του τέλους του διαστήματος εκτέλεσης της $S'$. Έτσι η $w$ που έχει επιτελεσθεί ανάμεσα στην $w_{S'}$ και στην $w_i^{S'}$, προηγείται της $w_S$ και άρα και της $r_{pre}$, το οποίο και είναι άτοπο.




Σχήμα 5.5 Η $S$ διαβάζει $v_i \neq null$ στον $post[seq_S][i]$ και η $w$ έπεται της $w_i^S$.

Σε όλες τις περιπτώσεις καταλήξαμε σε άτοπο. Έτσι συμπεραίνουμε ότι δεν υπάρχει UPDATE στη συνιστώσα $A_i$ που να σειριοποιείται ανάμεσα στην $U_i^S$ και στην $S$. Οπότε η $S$ επιστρέφει ένα συνεπές διάνυσμα τιμών. ■

***Θεώρημα 5.7*** *Ο T-Opt είναι μία σειριοποιήσιμη, Single-Scanner υλοποίηση ατομικών στιγμιοτύπων που πληροί την ιδιότητα ελεύθερη-αναμονής και επιτυγχάνει χρονική πολυπλοκότητα* $O(\mathrm{m})$ *για τη SCAN και* $O(1)$ *για την UPDATE, χρησιμοποιώντας* $O(mk)$ *κοινούς καταχωρητές μεγέθους* $O(\log k)$ *bit, όπου* $k$ *είναι ο μέγιστος αριθμός SCAN που μπορούν να επιτελεσθούν σε οποιαδήποτε εκτέλεση.*

# ΚΕΦΑΛΑΙΟ 6. Ο ΑΛΓΟΡΙΘΜΟΣ RT



Σε αυτό το κεφάλαιο παρουσιάζεται ένας Single-Scanner αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται RT. Ο RT επιτυγχάνει χρονική πολυπλοκότητα $O(n)$ για τη SCAN και $O(1)$ για την UPDATE, χρησιμοποιώντας $O(mn)$ κοινούς καταχωρητές. Από τους καταχωρητές που χρησιμοποιεί ο RT, ο μεγαλύτερος σε μέγεθος είναι ο καταχωρητής $seq$ που αποτελείται από $O(\log n)$ bit.

### 6.1. Περιγραφή του αλγορίθμου

Στον αλγόριθμο RT (Αλγόριθμος 7.1) γίνεται μία πρώτη προσπάθεια ώστε να μειωθεί ο αριθμός των καταχωρητών που χρησιμοποιεί ο T-Opt. Ο RT αντί να χρησιμοποιεί μη πεπερασμένο αριθμό γραμμών για τον πίνακα $post$, χρησιμοποιεί $n + 2$ γραμμές. Για να επιτευχθεί αυτό χρησιμοποιείται και ένας νέος πίνακας $n$ διαμοιραζόμενων καταχωρητών που ονομάζεται $state$. Κάθε καταχωρητής του πίνακα $state$ αντιστοιχίζεται σε μία διεργασία και ενημερώνεται κατά την επιτέλεση μίας UPDATE από τη διεργασία που έχει αντιστοιχιστεί στον καταχωρητή. Μία UPDATE που εκτελείται από κάποια διεργασία $p$ εγγράφει στον καταχωρητή $state[p]$ την τιμή που ανέγνωσε στον καταχωρητή $seq$ (γραμμή 2). Κάθε SCAN $S$ διαβάζει τους $n$ καταχωρητές του πίνακα $state$ και επιλέγει μια νέα τιμή $seq_S$ που θα εγγράψει στον $seq$, μια τιμή που δεν αναγνώσθηκε στους καταχωρητές του πίνακα $state[p]$ (γραμμές 8-11).

Ο κύριος στόχος του RT είναι να εγγυηθεί ότι μόνο οι UPDATE που επιτελούν το μεγαλύτερο τμήμα τους έπειτα από την εγγραφή του $seq$ από την $S$ (βλ. γραμμή 15),

εγγράφουν στους καταχωρητές της γραμμής $post[seq_S]$. Αυτό επιτυγχάνεται μέσω της εφαρμογής μίας read-write-read τεχνικής από τις UPDATE. Κάθε φορά που μία διεργασία $p$ επιτελεί μία UPDATE $U$, χρησιμοποιεί τον καταχωρητή $state[p]$ ώστε να πληροφορήσει τη SCAN για την τιμή που έχει αναγνώσει στον καταχωρητή $seq$ (γραμμές 1-2). Έπειτα η $U$ διαβάζει ξανά τον καταχωρητή $seq$ (γραμμή 3) και μόνο αν διαβάσει την ίδια τιμή στον καταχωρητή $seq$ (γραμμή 6) επιχειρεί να εγγράψει στον καταχωρητή $post$ (γραμμή 7).

Αλγόριθμος 6.1 Ψευδοκώδικας για τον αλγόριθμο RT.

```
  shared int state[1..n]={1, ..., 1};
  shared data post[1..(n+2)][1..m]={null, ..., null};
  shared data pre[1..m]={null, ..., null};
  shared int seq=1;

  void UPDATE(data v, int i){
     int curr_seq1, curr_seq2;
     data d1, d2;

1.   curr_seq1=seq;
2.   state[p]=curr_seq1;
3.   curr_seq2=seq;
4.   d1=pre[i];
5.   d2=post[curr_seq1][i];
6.   if(d2==null && curr_seq1==curr_seq2)
7.      post[curr_seq1][i]=d1;
8.   pre[i]=v;
  }

   data[] SCAN(void){
     data view[1..m], d1, d2;
     set act_set=∅;
     int lseq, j;

9.   act_set=act_set∪{seq};
10.  for(j=1;j≤n;j++)
11.     act_set=act_set ∪state[j];
12.  lseq=any integer of {1, ..., n+2}-act_set;
13.  for(j=1;j≤m;j++)
14.     post[lseq][j]=null;
15.  seq=lseq;
16.  for(j=1;j≤m;j++){
17.     d1=pre[j];
18.     d2=post[seq][j];
19.     if(d2==null)
20.        view[j]=d1;
21.     else
22.        view[j]=d2;
     }
     return view;
  }
```

Αν η $U$ επιτελέσει την write της στον καταχωρητή $state[p]$ πριν η $S$ αναγνώσει τον καταχωρητή $state[p]$, η $S$ θα επιλέξει μία τιμή για τον καταχωρητή $seq$ διαφορετική από αυτή που η $U$ ανέγνωσε στον καταχωρητή $seq$. Η μόνη άλλη ενδιαφέρουσα περίπτωση είναι η περίπτωση όπου η $U$ εγγράφει τον καταχωρητή $state[p]$ αφότου η $S$ τον αναγνώσει, και η $U$ επιτελεί τη δεύτερη εντολή read της στον καταχωρητή $seq$ πριν την $w_S$. Σε αυτή την περίπτωση η $U$ στη δεύτερη ανάγνωση του καταχωρητή $seq$, θα διαβάσει την τιμή που έγγραψε η SCAN που προηγείται της $S$ (ή την αρχική τιμή του $seq$ αν η $S$ είναι η πρώτη SCAN που εκτελείται στο σύστημα). Ωστόσο κάθε SCAN επιλέγει έναν σειριακό αριθμό διαφορετικό από αυτόν που επέλεξε η προηγούμενη SCAN (και από την αρχική του τιμή). Από τα παραπάνω είναι φανερό ότι το σύνολο από το οποίο επιλέγει η SCAN την τιμή που θα εγγράψει στον $seq$ αρκεί να έχει $n + 2$ διαφορετικές τιμές. Άρα ο πίνακα $post$ πρέπει να έχει $n + 2$ γραμμές.

### 6.2. Χρονική και χωρική πολυπλοκότητα του αλγορίθμου

Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι η χρονική πολυπλοκότητα της UPDATE είναι $O(1)$, ενώ η χρονική πολυπλοκότητα της SCAN είναι $O(n)$. Ο RT χρησιμοποιεί $(n + 3)m + n + 1$ καταχωρητές, όπου στους $(n + 3)m$ αποθηκεύεται μία μόνο τιμή του συνόλου $T$, ενώ το μέγεθος των υπολοίπων καταχωρητών είναι $O(\log n)$ bit (σε κάθε έναν από αυτούς αποθηκεύεται μία τιμή που ανήκει στο σύνολο $\{1, \ldots, n + 2\}$).

### 6.3. Απόδειξη της σειριοποιησιμότητας του αλγορίθμου

Σε αυτή την ενότητα θα αποδειχθεί η σειριοποιησιμότητα του RT. Για την απόδειξη της σειριοποιησιμότητας είναι χρήσιμος ο ακόλουθος συμβολισμός. Έστω $a$ μία οποιαδήποτε εκτέλεση του αλγορίθμου RT και έστω $S$ μία οποιαδήποτε SCAN που επιτελείται στην $a$. Συμβολίζουμε με $w_S$ την εντολή write που επιτελεί η $S$ (γραμμή 15), ενώ με $seq_S$ συμβολίζουμε την τιμή που έγγραψε η $S$ στον $seq$. Για κάθε $i \in \{1, \ldots, m\}$ χρησιμοποιούμε τον συμβολισμό $r_i^S, \tilde{r}_i^S, v_i, U_i^S, V_i^S$ και $w_i^S$ με τον ίδιο τρόπο όπως και στον αλγόριθμο T-Opt. Ειδικότερα συμβολίζουμε με $r_i^S$ την εντολή

read που επιτελεί μία SCAN $S$ στη γραμμή 17 του ψευδοκώδικα, με $\tilde{r}_i^S$ την read που επιτελεί η $S$ στη γραμμή 18. Έστω ότι η $S$ επιστρέφει τιμή ίση με $v_i$ για τη συνιστώσα $A_i$. Αν η $S$ έχει αναγνώσει τιμή ίση με $null$ στον $post[seq_S][i]$ και τιμή ίση με $v_i \neq null$ στον $pre[i]$, συμβολίζουμε με $U_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον καταχωρητή $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία write στον εν λόγω καταχωρητή πριν την επιτέλεση της $r_i^S$. Αν η $S$ έχει αναγνώσει τιμή ίση με $v_i \neq null$ στον $post[seq_S][i]$, χρησιμοποιούμε τον ακόλουθο συμβολισμό. Συμβολίζουμε με $V_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον $post[seq_S][i]$ και η write της στον $post[seq_S][i]$ είναι η τελευταία write στον εν λόγω καταχωρητή που προηγείται της $\tilde{r}_i^S$. Από τον ψευδοκώδικα του αλγορίθμου μπορούμε εύκολα να συμπεράνουμε ότι η $V_i^S$ πρέπει να έχει αναγνώσει τιμή ίση με $v_i$ στον $pre[i]$. Συμβολίζουμε με $U_i^S$ την UPDATE στη συνιστώσα $A_i$ που εγγράφει την τιμή $v_i$ στον $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία εντολή write πριν η $V_i^S$ αναγνώσει τον εν λόγω καταχωρητή. Σε κάθε περίπτωση συμβολίζουμε με $w_i^S$ την εντολή write που επιτελεί η $U_i^S$ στον $pre[i]$.

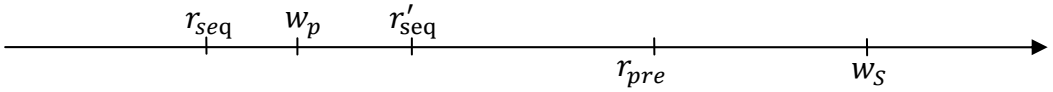
Αναθέτουμε σημεία σειριοποίησης για τις SCAN και τις UPDATE ακριβώς με τον ίδιο τρόπο όπως και στον T-Opt.

- Σε κάθε SCAN $S$ τοποθετούμε το σημείο σειριοποίησής της στην $w_S$.
- Για κάθε UPDATE στη συνιστώσα $A_i$ που επιτέλεσε την write της στον $pre[i]$ ανάμεσα στην $w_S$ και στην $w_i^S$, αναθέτουμε σημείο σειριοποίησης ακριβώς πριν την $w_S$ με τη σειρά που επιτέλεσαν τις write τους στον $pre[i]$.
- Στις υπόλοιπες UPDATE αναθέτουμε σημείο σειριοποίησης στην write τους στον $pre[i]$.

Η απόδειξη της σειριοποιησιμότητας του RT είναι κατά ένα μεγάλο μέρος ίδια με αυτή του T-Opt. Όλα τα λήμματα που αποδείχθηκαν για τον T-Opt ισχύουν και για τον RT, ενώ οι αποδείξεις των λημμάτων αυτών είναι ακριβώς ίδιες εκτός των Λημμάτων 5.2 και 5.3, τα οποία και παρατίθενται στη συνέχεια.

**Λήμμα 6.1** *Έστω ότι η S διαβάζει την τιμή $v_i$ (την οποία επιστρέφει για τη συνιστώσα $A_i$) στον καταχωρητή $post[seq_S][i]$, και έστω $r_{pre}$ η read της $V_i^S$ στον καταχωρητή $pre[i]$. Η $r_{pre}$ επιτελείται έπειτα από την $w_S$.*

**Απόδειξη.** Υποθέτουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι η $r_{pre}$ εκτελείται πριν την $w_S$. Συμβολίζουμε με $r_{seq}$ και $r'_{seq}$την πρώτη και τη δεύτερη read που επιτελεί η $V_i^S$ στον καταχωρητή $seq$ (γραμμές 1 και 3). Έστω $p$ η διεργασία που εκτελεί την $V_i^S$, έστω $w_p$ η write της $V_i^S$ στον καταχωρητή $state[p]$, και έστω $r_p$ η εντολή read της $S$ στον καταχωρητή $state[p]$ (γραμμή 11). Αφού έχουμε υποθέσει ότι η $r_{pre}$ προηγείται της $w_S$, το ίδιο θα ισχύει και για την $r'_{seq}$ (η $r'_{seq}$ προηγείται της $r_{pre}$, βλ. Σχήμα 6.1).




Σχήμα 6.1 Η $r'_{seq}$ προηγείται της $r_{pre}$.

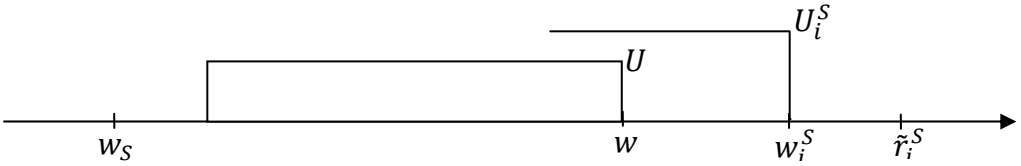
Έστω ότι η $r_p$ έπεται της $r'_{seq}$. Αφού η $w_p$ προηγείται της $r'_{seq}$, προκύπτει ότι η $w_p$ προηγείται της $r_p$. Έτσι η τιμή $t$ που έγγραψε η $w_p$ στον καταχωρητή $state[p]$, θα αναγνώσθηκε από την $r_p$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 11) προκύπτει ότι $t \neq seq_S$. Αφού η $V_i^S$ ανέγνωσε τιμή ίση με $t$ στον καταχωρητή $seq$, από τον ψευδοκώδικα προκύπτει ότι η $V_i^S$ δε δύναται να εγγράψει σε κάποιον καταχωρητή εκτός του $post[t][i] \neq post[seq_S][i]$, το οποίο και είναι άτοπο (αφού εξ ορισμού η $V_i^S$ εγγράφει στον $post[seq_S][i]$).

Έστω ότι η $r_p$ προηγείται της $r'_{seq}$. Αφού η $r'_{seq}$ προηγείται της $r_{pre}$ και έχουμε υποθέσει ότι η $r_{pre}$ προηγείται της $w_S$, η $r'_{seq}$ θα προηγείται της $w_S$. Έστω $S'$ η SCAN που εκτελείται ακριβώς πριν την $S$ στην $a$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμές 9-12), προκύπτει ότι η $S'$ εγγράφει διαφορετική τιμή στον $seq$ από την $S$ (δηλαδή θα ισχύει $seq_{S'} \neq seq_S$). Αφού η $r'_{seq}$ έπεται της $r_p$ και προηγείται της $w_S$, η $r'_{seq}$ διαβάζει τιμή ίση με $seq_{S'} \neq seq_S$ στον καταχωρητή $seq$. Έτσι η $V_i^S$ δεν εγγράφει στον καταχωρητή $post[seq_S][i]$, το οποίο και είναι άτοπο. ∎

Η απόδειξη του επόμενου λήμματος είναι παρόμοια με αυτή του Λήμματος 5.3. Ωστόσο παρουσιάζεται στη συνέχεια επειδή υπάρχουν μικρές διαφορές μεταξύ των δύο αποδείξεων. Στην απόδειξη του παρακάτω λήμματος χρησιμοποιείται το Λήμμα 5.1, το οποίο ισχύει και για τον αλγόριθμο RT (η απόδειξή του είναι ακριβώς η ίδια με αυτή που παρουσιάστηκε στην ενότητα 5).

**Λήμμα 6.2** *Έστω μία οποιαδήποτε SCAN $S$. Για κάθε $i \in \{1, \ldots, m\}$ τέτοιο ώστε η $w_i^S$ να έπεται της $w_S$, ισχύει ότι κάθε UPDATE στην συνιστώσα $A_i$ που επιτελεί την write της στον $pre[i]$ μεταξύ της $w_S$ και της $w_i^S$ (με την $U_i^S$ να συμπεριλαμβάνεται), ξεκινά την εκτέλεσή της πριν την $w_S$.*

**Απόδειξη.** Υποθέτουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι υπάρχει μία UPDATE στη συνιστώσα $A_i$ που ξεκινά την εκτέλεσή της έπειτα από την $w_S$ και επιτελεί την write της στον $pre[i]$ (την οποία την συμβολίζουμε με $w$) πριν την $w_i^S$ (βλ. Σχήμα 6.2). Από το Λήμμα 6.1 προκύπτει ότι η $w_i^S$ προηγείται της $\tilde{r}_i^S$ και ως εκ τούτου η $U$ ολοκληρώνει την εκτέλεσή της πριν το τέλος της $S$. Αφού το διάστημα εκτέλεσης της $U$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$, η $U$ θα έχει αναγνώσει την τιμή που έγγραψε η $S$ (έστω $seq_S$) στον καταχωρητή $seq$ και τις δύο φορές (γραμμές 1 και 3). Έτσι η δεύτερη συνθήκη του ελέγχου $if$ της γραμμής 6 θα αποτιμηθεί ως $true$. Έτσι στην περίπτωση που η $U$ αναγνώσει τιμή ίση με $null$ στον καταχωρητή $post[seq_S][i]$, θα εγγράψει στον καταχωρητή $post[seq_S][i]$ την τιμή που ανέγνωσε στον καταχωρητή $pre[i]$.




Σχήμα 6.2 Η $U$ ξεκινά έπειτα από την $w_S$ και ολοκληρώνεται πριν την $w_i^S$.

Από τον ψευδοκώδικα του αλγορίθμου (γραμμές 12-13), η $S$ αρχικοποιεί τους $m$ καταχωρητές του $post[seq_S]$ σε $null$ πριν την επιτέλεση της $w_S$. Αφού η $U$ ξεκίνησε την εκτέλεσή της έπειτα από την $w_S$, η εκτέλεση των γραμμών 6-7 του ψευδοκώδικα (ο έλεγχος $if$ και η πιθανή εγγραφή του καταχωρητή $post[seq_S][i]$) από την $U$ έπεται της αρχικοποίησης των καταχωρητών του $post[seq_S]$ σε $null$ από την $S$. Αφού η

$w_i^S$ προηγείται της $\tilde{r}_i^S$ και η $U$ επιτελεί την write της στον $pre[i]$ πριν την $w_i^S$, η εκτέλεση των γραμμών 6-7 από την $U$ προηγείται της $\tilde{r}_i^S$. Άρα η $\tilde{r}_i^S$ διαβάζει τιμή διάφορη του $null$ στον καταχωρητή $post[seq_S][i]$ και έτσι η λειτουργία $V_i^S$ είναι καλά ορισμένη. Από τον τρόπο ορισμού της $V_i^S$, η $V_i^S$ εγγράφει στον καταχωρητή $post[seq_S][i]$ και έτσι διαβάζει $null$ στον $post[seq_S][i]$ και $seq_S$ στον $seq$ (στις γραμμές 1 και 3). Επίσης η εντολή read της $V_i^S$ στον καταχωρητή $pre[i]$ έπεται της $w_i^S$, αφού η $V_i^S$ διαβάζει στον καταχωρητή $pre[i]$ την τιμή που εγγράφηκε από την $w_i^S$. Οπότε προκύπτει ότι η εντολή read της $V_i^S$ στον καταχωρητή $post[seq_S][i]$, η οποία έπεται της read στον $pre[i]$, επιτελείται έπειτα από την εκτέλεση των γραμμών 6-7. Έτσι η $V_i^S$ διαβάζει μία τιμή διαφορετική του $null$ στον καταχωρητή $post[seq_S][i]$ από την $U$, το οποίο και είναι άτοπο. ■

Η απόδειξη των υπόλοιπων λημμάτων είναι ακριβώς η ίδια με αυτή που παρουσιάζεται στο προηγούμενο κεφάλαιο.

***Θεώρημα 6.3*** *Ο αλγόριθμος RT είναι μία σειριοποιήσιμη, Single-Scanner υλοποίηση ατομικών στιγμιοτύπων που πληροί την ιδιότητα ελεύθερη-αναμονής και επιτυγχάνει χρονική πολυπλοκότητα $O(n)$ για τη SCAN και $O(1)$ για την UPDATE, χρησιμοποιώντας $O(mn)$ κοινούς καταχωρητές μεγέθους $O(log\, n)$ bit.*

# ΚΕΦΑΛΑΙΟ 7. Ο ΑΛΓΟΡΙΘΜΟΣ RT-OPT



Σε αυτό το κεφάλαιο παρουσιάζεται ένας Single-Scanner αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται RT-Opt. Ο RT-Opt επιτυγχάνει χρονική πολυπλοκότητα $O(m)$ για τη SCAN και $O(1)$ για την UPDATE, χρησιμοποιώντας $O(nm)$ καταχωρητές, μεγέθους $O(\max\{\log n, \log |T|\})$ bit.

## 7.1. Περιγραφή του αλγορίθμου

Ο ψευδοκώδικας του αλγορίθμου RT-Opt παρουσιάζεται στον Αλγόριθμο 7.1. Η λειτουργία UPDATE του RT-Opt είναι ακριβώς ίδια με αυτή του RT. Ο κύριος στόχος της SCAN του RT αλλά και του RT-Opt είναι να παρακολουθούν σε ποιες γραμμές του πίνακα $post$ πρόκειται να γράψουν κάποιες «παλιές» UPDATE (δηλαδή UPDATE που έχουν επιτελέσει ένα σημαντικό μέρος της εκτέλεσής τους πριν την εντολή write της $S$ στον $seq$). Η $S$ πρέπει να επιλέξει ως νέα τιμή για τον $seq$, έναν σειριακό αριθμό που θα δείχνει σε μια γραμμή του πίνακα $post$, όπου καμία «παλιά» UPDATE δεν πρόκειται να γράψει. Έτσι θα είναι σίγουρο ότι αν η $S$ διαβάσει στη γραμμή αυτή μια τιμή διαφορετική του $null$, αυτή η τιμή θα έχει εγγραφεί από κάποια UPDATE που έχει εκτελέσει το μεγαλύτερο τμήμα της λειτουργίας της έπειτα από την εντολή write της $S$ στον $seq$. Προκειμένου να επιτευχθεί αυτό, κάθε SCAN διαβάζει και τους $n$ καταχωρητές του πίνακα $state$ και επιλέγει κάποια τιμή διαφορετική από αυτές που ανέγνωσε. Δυστυχώς αυτό έχει ως αποτέλεσμα μια μη επιθυμητή αύξηση της χρονικής πολυπλοκότητας της SCAN σε $O(n)$. Για να διατηρηθεί η πολυπλοκότητα της SCAN χαμηλή ($O(m)$), κάθε SCAN του αλγορίθμου RT-Opt διαβάζει μόνο $m$ από τους $n$ καταχωρητές του πίνακα $state$. Έτσι χρειάζονται $\lceil n/m \rceil$

διαδοχικές SCAN για να διαβαστούν και οι $n$ καταχωρητές του πίνακα $state$. Κάθε εκτέλεση του αλγορίθμου RT-Opt δύναται να χωριστεί σε τμήματα εκτέλεσης (*εποχές*), όπου κάθε εποχή περιέχει ακριβώς $\lceil n/m \rceil$ διαδοχικές SCAN. Επιπρόσθετα ο καταχωρητής $seq$ παίρνει τιμές από το σύνολο $\{1, \ldots, n + 2\lceil n/m \rceil + 1\}$ (το οποίο προφανώς είναι μεγαλύτερο από το σύνολο $\{1, \ldots, n + 2\}$ του αλγορίθμου RT).

Το σύνολο $free$ χρησιμοποιείται για την παρακολούθηση των τιμών που είναι δυνατό να χρησιμοποιηθούν ως σειριακοί αριθμοί από τις SCAN κάθε εποχής. Ουσιαστικά το σύνολο $free$ περιέχει τις γραμμές του πίνακα $post$ όπου κατά τη διάρκεια εκτέλεσης της τρέχουσας SCAN, δεν πρόκειται να γράψει κάποια UPDATE. Κατά τη διάρκεια οποιασδήποτε εποχής $E_j, j > 1$, όλοι οι σειριακοί αριθμοί που επιλέγονται από τις SCAN της $E_j$ είναι διαφορετικοί μεταξύ τους (γραμμή 14). Για την πρώτη εποχή $E_1$, όλοι οι σειριακοί αριθμοί είναι διαφορετικοί και επιπρόσθετα είναι διαφορετικοί της αρχικής τιμής του $seq$.

Έστω μία εποχή $E_j, j > 1$. Υπενθυμίζεται ότι όλοι οι καταχωρητές του πίνακα $state$ έχουν αναγνωσθεί τουλάχιστον μία φορά στην εποχή $E_{j-1}$. Όλες οι τιμές που αναγνώσθηκαν στον πίνακα $state$ ενδέχεται να δεικτοδοτούν γραμμές του πίνακα $post$, στις οποίες παλιές UPDATE δύναται να γράψουν. Έτσι καμία από τις προαναφερθείσες τιμές δεν πρέπει να επιλεχθεί ως σειριακός αριθμός από κάποια SCAN της εποχής $E_j$. Ωστόσο δεν είναι αρκετό να εξαιρεθούν μόνο οι προαναφερθείσες τιμές από το σύνολο των διαθέσιμων σειριακών αριθμών της εποχής $E_j$. Πρέπει επιπρόσθετα να εξαιρεθούν και οι τιμές που χρησιμοποιήθηκαν και στην εποχή $E_{j-1}$. Ο λόγος είναι ότι κάποια UPDATE (που εκτελείται από κάποια διεργασία $p$) δύναται να αναγνώσει στον καταχωρητή $seq$ την τιμή που εγγράφηκε από κάποια SCAN στην εποχή $E_{j-1}$ και να εγγράψει τον καταχωρητή $state[p]$, έπειτα από την ανάγνωση του $state[p]$ στην εποχή $E_{j-1}$. Μία τέτοια UPDATE κατά την επιτέλεση της δεύτερης ανάγνωσής της στον $seq$ θα διαβάσει την ίδια τιμή που εγγράφηκε τώρα από κάποια SCAN στην εποχή $E_j$. Έτσι οι τιμές που επιλέχθηκαν ως σειριακοί αριθμοί από SCAN στην εποχή $E_j$ μπορεί να δείχνουν γραμμές του $post$ που μπορούν να εγγραφούν από «παλιές» UPDATE. Άρα οι τιμές που επιλέχθηκαν ως σειριακοί αριθμοί στην εποχή $E_{j-1}$ εξαι-

ρούνται από το σύνολο των διαθέσιμων σειριακών αριθμών για τις SCAN της εποχής $E_j$.

Αλγόριθμος 7.1 Ψευδοκώδικας για τον αλγόριθμο RT-Opt.

```
      constant PACE=m;
      constant PERIODS=⌈n/PACE⌉;
      shared int seq=1;
      shared int state[1..(PERIODS*m)]={1, ..., 1};
      shared data pre[1..m]={null, ..., null};
      shared data post[1..(n+2*PERIODS+1)][1..m]={null, ..., null};

      void UPDATE(data v, int i){
           int curr_seq1, curr_seq2;
           data d1, d2

1.         curr_seq1=seq;
2.         state[p]=curr_seq1;
3.         curr_seq2=seq;
4.         d1=pre[i];
5.         d2=post[curr_seq1][i];
6.         if(d2==null && curr_seq1==curr_seq2)
7.              post[curr_seq1][i]=d1;
8.         pre[i]=v;
      }

      data[] SCAN(){
           data view[1..m], d1, d2;
           int lseq, j;
           static int cur_period=0;
           static set free=∅;
           static set candidates={2, ..., n+2*PERIODS+1};

9.         if(cur_period==0){
10.             free=free ∪ candidates;
11.             candidates={1, ..., n+2*PERIODS+1};
           }
12.        lseq=any element of set free;
13.        for(j=1;j≤m;j++)
                post[lseq][j]=null;
14.        free=free-lseq;
15.        candidates=candidates-lseq;
16.        cur_period=(cur_period+1)%PERIODS;
17.        seq=lseq;
18.        for(j=1;j≤PACE;j++)
19.             candidates=candidates-state[cur_period*PACE+j];
20.        for(j=1;j≤m;j++){
21.             d1=pre[i];
22.             d2=post[seq][j];
23.             if(d2==null) view[j]=d1;
24.             else view[j]=d2;
           }
25.        return view;
       }
```

Το σύνολο $candidates$ καταγράφει όλες τις τιμές που επιτρέπονται να επιλεχθούν ως σειριακοί αριθμοί από τις SCAN της επόμενης εποχής. Παρατηρούμε ότι κατά την έναρξη κάθε εποχής, το σύνολο $candidates$ αρχικοποιείται έτσι ώστε να περιέχει όλους τους πιθανούς σειριακούς αριθμούς (γραμμή 11). Κατά τη διάρκεια εκτέλεσης των $\lceil n/m \rceil$ SCAN μίας εποχής, όλες οι τιμές που διαβάζονται στον πίνακα $state$ καθώς και οι τιμές που χρησιμοποιήθηκαν ως σειριακοί αριθμοί, αφαιρούνται από το σύνολο $candidates$ (γραμμές 15, 19). Κατά την έναρξη της επόμενης εποχής, οι τιμές που έχουν απομείνει στο σύνολο $candidates$ μετακινούνται στο σύνολο $free$, όπου αποθηκεύονται οι διαθέσιμοι σειριακοί αριθμοί για την εποχή που ξεκινά. Σημειώνεται ότι δεν προστίθενται νέα στοιχεία στο σύνολο $free$ έπειτα από την έναρξη μίας εποχής. Έτσι το σύνολο $free$ καταγράφει σωστά το σύνολο των διαθέσιμων σειριακών αριθμών για τις SCAN κάθε εποχής.

Κατά την έναρξη μίας οποιασδήποτε εκτέλεσης $a$ του RT-Opt, το σύνολο $candidates$ περιέχει $n + 2 * PERIODS$ διαφορετικούς σειριακούς αριθμούς, όπου $PERIODS = \lceil n/m \rceil$. Κατά τη διάρκεια της $E_1$, το πολύ $n + PERIODS + 1$ σειριακοί αριθμοί εξάγονται από το σύνολο $candidates$ ($n$ από τον $state$, $PERIODS$ για τις τιμές που χρησιμοποιήσαν οι SCAN στην $E_1$, και η αρχική τιμή του $s$). Έτσι στο τέλος της $E_1$, το σύνολο $candidates$ περιέχει τουλάχιστον $PERIODS$ τιμές, οι οποίες προστίθενται στο σύνολο $free$ κατά την έναρξη της εποχής $E_2$. Άρα το σύνολο $free$ περιέχει αρκετούς σειριακούς αριθμούς για τις $PERIODS$ SCAN που εκτελούνται στην εποχή $E_2$. Έστω μία οποιαδήποτε εποχή $E_j, j > 1$. Κατά την έναρξη της εποχής $E_j$, το σύνολο $candidates$ περιέχει $n + 2 * PERIODS + 1$ διαφορετικούς σειριακούς αριθμούς. Κατά τη διάρκεια της εποχής $E_j$, το πολύ $n + PERIODS$ σειριακοί αριθμοί αφαιρούνται από το σύνολο $candidates$. Έτσι τουλάχιστον $PERIODS + 1$ σειριακοί αριθμοί προστίθενται στο σύνολο $free$ κατά την έναρξη της $E_{j+1}$, που είναι αρκετοί για τις $PERIODS$ SCAN της εποχής $E_{j+1}$. Από τα παραπάνω προκύπτει ότι ο RT-Opt απαιτεί τη χρήση $n + 2 * \lceil \mathrm{n/m} \rceil + 1$ διαφορετικών σειριακών αριθμών.

### 7.2. Χρονική και χωρική πολυπλοκότητα του αλγορίθμου

Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι η χρονική πολυπλοκότητα του RT-Opt για την UPDATE είναι $O(1)$. Η χρονική πολυπλοκότητα της SCAN είναι $O(m)$ αφού κάθε SCAN προσπελάζει $3m + 1$ κοινούς καταχωρητές, οι οποίοι είναι: α) $m$ καταχωρητές του πίνακα $state$ (αφού ισχύει $PACE = m$), β) $m$ καταχωρητές του πίνακα $post$, γ) $m$ καταχωρητές του πίνακα $pre$, και δ) τον καταχωρητή $seq$[3]. Ο αλγόριθμος RT-Opt χρησιμοποιεί $O(mn)$ καταχωρητές, όπου όλοι εκτός του καταχωρητή $seq$ αποθηκεύουν μόνο μία τιμή του συνόλου $T$, ενώ το μέγεθος του $seq$ είναι $O(\log n)$ bit. Είναι αξιοσημείωτο ότι το $PACE$ δύναται να πάρει οποιαδήποτε τιμή μεταξύ του 1 και του $n$, δίχως να επηρεάζεται η ορθότητα του αλγορίθμου. Αν $PACE = n$, ο αλγόριθμος RT-Opt λειτουργεί με τον ίδιο ακριβώς τρόπο όπως ο αλγόριθμος RT, ενώ αν ισχύει $PACE \leq m$ ο RT-Opt επιτυγχάνει βέλτιστη χρονική πολυπλοκότητα.

### 7.3. Απόδειξη της ορθότητας του αλγορίθμου

Για να αποδείξουμε την ορθότητα του αλγορίθμου RT-Opt θα χρησιμοποιήσουμε τον ακόλουθο συμβολισμό. Έστω $a$ μία οποιαδήποτε εκτέλεση του RT-Opt και έστω $S$ μία οποιαδήποτε SCAN στην $a$. Συμβολίζουμε με $w_S$ την εγγραφή του καταχωρητή $seq$ από την $S$ στη γραμμή 7 του ψευδοκώδικα, ενώ με $seq_S$ συμβολίζουμε την τιμή που έγγραψε η $S$ στον $seq$. Για κάθε $i \in \{1, \ldots, m\}$ χρησιμοποιούμε τον ακόλουθο συμβολισμό $r_i^S, \tilde{r}_i^S, v_i, U_i^S, V_i^S$ και $w_i^S$ με τον ίδιο τρόπο όπως και στον αλγόριθμο T-Opt. Ειδικότερα με $r_i^S$ συμβολίζουμε την εντολή read της $S$ στον $pre[i]$ (γραμμή 9), ενώ με $\tilde{r}_i^S$ συμβολίζουμε την εντολή read της $S$ στον $post[seq_S][i]$ (γραμμή 10). Έστω ότι η $S$ επιστρέφει τιμή ίση με $v_i$ για τη συνιστώσα $A_i$. Αν η $S$ έχει αναγνώσει τιμή ίση με $null$ στον $post[seq_S][i]$ και τιμή ίση με $v_i \neq null$ στον $pre[i]$, συμβολί-

[3] Υπενθυμίζουμε ότι το κλασικό μοντέλο ενός κατανεμημένου συστήματος κοινής μνήμης (το οποία χρησιμοποιείται σε αυτή την εργασία) δεν προσμετρά τους τοπικούς υπολογισμούς που επιτελούν οι διεργασίες.

ζουμε με $U_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον καταχωρητή $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία εντολή write στον εν λόγω καταχωρητή πριν την επιτέλεση της $r_i^S$. Αν η $S$ έχει αναγνώσει τιμή ίση με $v_i \neq null$ στον $post[seq_S][i]$, χρησιμοποιούμε τον ακόλουθο συμβολισμό. Συμβολίζουμε με $V_i^S$ την UPDATE που εγγράφει την τιμή $v_i$ στον $post[seq_S][i]$ και η write της στον $post[seq_S][i]$ είναι η τελευταία εντολή write στον εν λόγω καταχωρητή που προηγείται της $\tilde{r}_i^S$. Από τον ψευδοκώδικα του αλγορίθμου μπορούμε εύκολα να συμπεράνουμε ότι η $V_i^S$ πρέπει να έχει αναγνώσει τιμή ίση με $v_i$ στον $pre[i]$. Συμβολίζουμε με $U_i^S$ την UPDATE στη συνιστώσα $A_i$ που εγγράφει την τιμή $v_i$ στον $pre[i]$ και η write της στον $pre[i]$ είναι η τελευταία εντολή write πριν η $V_i^S$ αναγνώσει τον εν λόγω καταχωρητή. Σε κάθε περίπτωση συμβολίζουμε με $w_i^S$ την εντολή write που επιτελεί η $U_i^S$ στον $pre[i]$.

Χωρίζουμε την εκτέλεση $a$ του αλγορίθμου RT-Opt σε εποχές, έτσι ώστε κάθε εποχή να περιέχει ακριβώς $\lceil n/m \rceil$ SCAN. Συμβολίζουμε με $E_i$ την i-οστή εποχή της εκτέλεσης $a$, $i \geq 1$. Η εποχή $E_1$ ξεκινά με την εκτέλεση της πρώτης εντολής της $a$ και τελειώνει με την εκτέλεση της τελευταίας εντολής της $\lceil n/m \rceil$-οστής SCAN (ή τελειώνει με την τελευταία εντολή της $a$, αν υπάρχουν λιγότερες από $\lceil n/m \rceil$ SCAN). Για κάθε $i > 1$, η εποχή $E_i$ ξεκινά από το σημείο στο οποίο η $((i-1)\lceil n/m \rceil)$-οστή SCAN τερματίζει τη λειτουργία της και τελειώνει με την εκτέλεση της τελευταίας εντολής της $(i\lceil n/m \rceil)$-οστής SCAN (ή τελειώνει με την τελευταία εντολή της $a$, αν υπάρχουν λιγότερες από $(i\lceil n/m \rceil)$ SCAN). Έστω ο μεγαλύτερος δυνατός ακέραιος $c_1 \geq 0$, τέτοιος ώστε η $a$ να περιέχει $c_1\lceil n/m \rceil + c_2$ SCAN όπου η $0 \leq c_2 < \lceil n/m \rceil$ είναι ακέραια σταθερά. Αξίζει να σημειωθεί ότι αν $c_2 = 0$, τότε η $a$ περιέχει $c_1$ εποχές. Ωστόσο αν $c_2 > 0$, τότε η $a$ περιέχει $c_1 + 1$ εποχές, όπου η $(c_1 + 1)$-οστή εποχή περιέχει λιγότερες από $\lceil n/m \rceil$ SCAN. Επίσης υπενθυμίζουμε ότι αν το διάστημα εκτέλεσης της $a$ είναι άπειρο, τότε το διάστημα εκτέλεσης και της $(c_1 + 1)$-οστής εποχής είναι άπειρο.

Έστω ότι η $a$ περιέχει $q$ εποχές. Για κάθε $i \in \{1, \ldots, q\}$, συμβολίζουμε με $SN_i$ το σύνολο τιμών που εγγράφηκαν στον καταχωρητή $seq$ από οποιαδήποτε SCAN στην εποχή $E_i$. Συμβολίζουμε με $free_i$, τα στοιχεία που περιέχει το σύνολο $free$ στο τέλος

της εποχής $E_i$, και με $candidates_i$ τα στοιχεία που περιέχει το σύνολο $candidates$ στο τέλος της εποχής $E_i$.

Για να αποδείξουμε τη σειριοποιησιμότητα του αλγορίθμου πρέπει αρχικά να αποδώσουμε σημεία σειριοποίησης σε κάθε SCAN και σε κάθε UPDATE. Έπειτα πρέπει να αποδείξουμε ότι το σημείο σειριοποίησης κάθε SCAN ή UPDATE βρίσκεται εντός του διαστήματος εκτέλεσής της και εν τέλει αποδεικνύουμε ότι κάθε SCAN επιστρέφει ένα συνεπές διάνυσμα τιμών.

Έτσι αρχικά αποδίδουμε σημεία σειριοποίησης στις SCAN και στις UPDATE.

- Τοποθετούμε το σημείο σειριοποίησης μιας SCAN $S$, στο σημείο όπου επιτέλεσε την $w_S$.
- Για κάθε $i \in \{1, .., m\}$, αν η $w_i^S$ επιτελέσθηκε έπειτα από την $w_S$, τοποθετούμε σημείο σειριοποίησης στην $U_i^S$ ακριβώς πριν την $w_S$. Ακόμη για κάθε UPDATE στη συνιστώσα $A_i$ που επιτέλεσε την write της ανάμεσα στην $w_S$ και στην $w_i^S$, τοποθετούμε σημείο σειριοποίησης ακριβώς πριν την $w_S$. Τα σημεία σειριοποίησης των προαναφερθεισών λειτουργιών τοποθετούνται με τη σειρά με την οποία επιτελέσθηκαν οι write τους στον καταχωρητή $pre[i]$.
- Αφότου αναθέσουμε σημεία σειριοποίησης σε όλες τις SCAN αλλά και σε μερικές UPDATE (με τον τρόπο που περιγράφηκε παραπάνω), αναθέτουμε σημεία σειριοποίησης στις υπόλοιπες UPDATE στο σημείο όπου επιτέλεσαν την write τους στον καταχωρητή $pre[i]$.

Η απόδειξη της σειριοποιησιμότητας του RT-Opt είναι κατά ένα μεγάλο μέρος ίδια με αυτή του T-Opt. Όλα τα λήμματα που αποδείχθηκαν για τον T-Opt ισχύουν και για τον RT, ενώ οι αποδείξεις των λημμάτων αυτών είναι ακριβώς ίδιες εκτός των Λημμάτων 5.2 και 5.3, τα οποία και παρατίθενται στη συνέχεια. Αρχικά παρατίθενται μερικά τεχνικά λήμματα (7.1-7.5). Έπειτα αποδεικνύεται το αντίστοιχο λήμμα του Λήμματος 5.2 του αλγορίθμου T-Opt. Το αντίστοιχο λήμμα με το Λήμμα 5.3 του αλγορίθμου T-Opt είναι ακριβώς ίδιο με αυτό που παρουσιάστηκε στην προηγούμενη ενότητα για τον αλγόριθμο RT (Λήμμα 6.2).

**Λήμμα 7.1** *Για κάθε* $j \in \{1,..,q\}$ *και για κάθε* $p \in \{1,\ldots,n\}$*, υπάρχει μία και μοναδική SCAN, η οποία διαβάζει την τιμή του καταχωρητή* $state[p]$ *στην εποχή* $E_j$*.*

**Απόδειξη.** Από τον τρόπο ορισμού της $E_j$, ακριβώς $\lceil n/m \rceil$ SCAN επιτελούνται στην $E_j$. Κάθε μία από τις προαναφερθείσες SCAN διαβάζει $m$ διαφορετικούς καταχωρητές του πίνακα $state$. Άρα και οι $\lceil n/m \rceil * m$ καταχωρητές του πίνακα $state$ διαβάζονται ακριβώς μία φορά στην $E_j$. ∎

**Λήμμα 7.2** *Για κάθε* $j \in \{1,\ldots,q\}$*, ισχύει:*

1. $free_j \cap SN_j = \emptyset$*, και*
2. $candidates_j \cap SN_j = \emptyset$*.*

**Απόδειξη.** Από τον ψευδοκώδικα του αλγορίθμου δύναται να συμπεράνουμε ότι κάθε τιμή που εγγράφεται στον $seq$ εξάγεται από το σύνολο $free$ (γραμμή 14). Το ίδιο ισχύει και για το σύνολο $candidates$ (γραμμή 15). Έτσι στο τέλος της εποχής $E_j$ ισχύει ότι $free_j \cap SN_j = \emptyset$ και $candidates_j \cap SN_j = \emptyset$. ∎

Από τον ψευδοκώδικα του αλγορίθμου (γραμμές 8 και 16) είναι φανερό ότι οι γραμμές 9-10 εκτελούνται μόνο από την πρώτη SCAN κάθε εποχής, όπως αναφέρεται στο ακόλουθο λήμμα.

**Λήμμα 7.3** *Για κάθε* $j \in \{1,\ldots,q\}$*, οι παρακάτω ισχυρισμοί ισχύουν για την πρώτη SCAN* $S$ *της εποχής* $E_j$*:*

1. *η* $S$ *είναι η μοναδική SCAN που προσθέτει στοιχεία στο σύνολο* $free$ *στην εποχή* $E_j$*, και*
2. *η* $S$ *είναι η μοναδική SCAN που επιτελεί την γραμμή 10 του ψευδοκώδικα αρχικοποιώντας το σύνολο* $candidates$*.*

Στο επόμενο λήμμα αποδεικνύεται ότι κάθε SCAN μιας οποιασδήποτε εποχής εγγράφει μία διαφορετική τιμή στον καταχωρητή $seq$ από τις τιμές που έχουν ήδη εγγραφεί από τις SCAN της εν λόγω εποχής.

**Λήμμα 7.4** *Για κάθε* $j \in \{1, \ldots, q\}$*, κάθε SCAN της εποχής* $E_j$ *εγγράφει μία διαφορετική τιμή στον καταχωρητή* $seq$ *από τις τιμές που έχουν ήδη εγγραφεί από προηγούμενες SCAN της* $E_j$*.*

**Απόδειξη.** Από το Λήμμα 7.3 συμπεραίνουμε ότι μόνο η πρώτη SCAN της $E_j$ προσθέτει νέα στοιχεία στο σύνολο $free$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 11) κάθε SCAN $S$ επιλέγει την τιμή που θα εγγράψει στον καταχωρητή $seq$ από το σύνολο $free$. Την εν λόγω τιμή, η SCAN την αφαιρεί έπειτα από το σύνολο $free$ (γραμμή 14). Έτσι οι SCAN που ακολουθούν και ανήκουν στην εποχή $E_j$ θα επιλέξουν μία διαφορετική τιμή από αυτή που επέλεξε η $S$. ∎

Στο επόμενο λήμμα θα δειχθεί ότι οι SCAN μίας εποχής επιλέγουν διαφορετικές τιμές για τον καταχωρητή $seq$ από τις SCAN της προηγούμενης εποχής.

**Λήμμα 7.5** *Για κάθε* $j \in \{2, \ldots, q\}$*, ισχύει ότι* $SN_{j-1} \cap SN_j = \emptyset$*.*

**Απόδειξη.** Έστω ότι $j \in \{2, \ldots, q\}$. Από το Λήμμα 7.2 ισχύει ότι $free_{j-1} \cap SN_{j-1} = \emptyset$ και $candidates_{j-1} \cap SN_{j-1} = \emptyset$. Έτσι στο τέλος της εποχής $E_{j-1}$, το σύνολο $free$ και το σύνολο $candidates$ δεν έχουν κοινά στοιχεία με το σύνολο $SN_{j-1}$. Από το Λήμμα 7.3, η μοναδική SCAN της εποχής $E_j$ που προσθέτει στοιχεία στο σύνολο $free$ (γραμμή 9) είναι η πρώτη SCAN της εν λόγω εποχής (έστω $S$ αυτή). Η $S$ προσθέτει τα στοιχεία του συνόλου $candidates_{j-1}$ στο σύνολο $free_{j-1}$. Συμβολίζουμε με $free_j^S$, το σύνολο $free$ έπειτα από την εκτέλεση της γραμμής 9 του ψευδοκώδικα από την $S$. Είναι προφανές ότι ισχύει $free_j^S = free_{j-1} \cup candidates_{j-1}$. Αφού ισχύει $free_{j-1} \cap SN_{j-1} = \emptyset$ και $candidates_{j-1} \cap SN_{j-1} = \emptyset$, συνεπάγεται ότι $free_j^S \cap SN_{j-1} = \emptyset$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 11), όλα τα στοιχεία του συνόλου $SN_j$ επιλέγονται από το σύνολο $free_j^S$. Έτσι θα ισχύει $SN_{j-1} \cap SN_j = \emptyset$, όπως και απαιτείται από τον ισχυρισμό του λήμματος. ∎

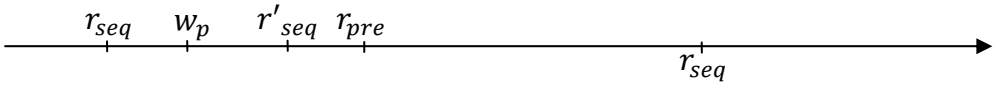
Σε αυτό το σημείο θα αποδειχθεί με τη χρήση των παραπάνω τεχνικών λημμάτων το αντίστοιχο λήμμα του Λήμματος 5.2 του αλγορίθμου T-Opt.

**Λήμμα 7.6** *Έστω ότι η S διαβάζει την τιμή $v_i \neq null$ στον καταχωρητή $post[seq_S][i]$ και έστω $r_{pre}$ η εντολή read της $V_i^S$ στον καταχωρητή $pre[i]$. Η $r_{pre}$ εκτελείται έπειτα από την $w_S$.*

**Απόδειξη.** Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $r_{pre}$ εκτελείται πριν από την $w_S$. Συμβολίζουμε με $r_{seq}$ την πρώτη read που επιτελεί η $V_i^S$ στον $seq$ (βλ. γραμμή 1), και με $r'_{seq}$ τη δεύτερη read που επιτελεί η $V_i^S$ στον $seq$ (βλ. γραμμή 3). Έστω $p$ η διεργασία που εκτελεί την $V_i^S$ και έστω $w_p$ η write που επιτελεί η $V_i^S$ στον καταχωρητή $state[p]$ (βλ. γραμμή 2). Αφού η $r_{pre}$ προηγείται της $w_S$, το ίδιο θα ισχύει και για την $r'_{seq}$ (αφού η $r'_{seq}$ προηγείται της $r_{pre}$). Έστω ότι η $S$ έχει εκτελεστεί σε κάποια εποχή $E_j$, $j \geq 1$. Διακρίνουμε τις ακόλουθες περιπτώσεις.

1) Έστω ότι $j = 1$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 10) κατά τη διάρκεια της πρώτης εποχής, το σύνολο $free$ δεν περιέχει την αρχική τιμή του καταχωρητή $seq$. Αφού οι SCAN μιας εποχής επιλέγουν μεταξύ τους διαφορετικά στοιχεία του συνόλου $free$, η $S$ επιλέγει έναν αριθμό διαφορετικό από την αρχική τιμή του καταχωρητή $seq$. Από το Λήμμα 7.4 δύναται να συμπεράνουμε ότι καμία SCAN που προηγείται της $S$, δεν επιλέγει την ίδια τιμή με την $S$. Εξ ορισμού, η $V_i^S$ εγγράφει στον καταχωρητή $post[seq_S][i]$. Από τον ψευδοκώδικα του αλγορίθμου, η $V_i^S$ εγγράφει στον $post[seq_S][i]$ μόνο αν οι εντολές $r_{seq}$, $r'_{seq}$ ανέγνωσαν την ίδια τιμή $seq_S$ στον $seq$. Έτσι προκύπτει ότι οι εντολές $r_{seq}$, $r'_{seq}$ επιτελέσθηκαν έπειτα από την $w_S$. Αφού η $r_{pre}$ έπεται της $r'_{seq}$, η $r_{pre}$ έπεται της $w_S$, το οποίο και είναι άτοπο.
2) Έστω ότι $j > 1$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $S$ αρχικοποιεί τον καταχωρητή $post[seq_S][i]$, εκχωρώντας του τιμή ίση με $null$. Εξ ορισμού η $V_i^S$ εγγράφει στον $post[seq_S][i]$ την τιμή που επιστρέφει η $S$. Άρα η $V_i^S$ εγγράφει στον καταχωρητή $post[seq_S][i]$, έπειτα από την αρχικοποίηση αυτού από την $S$. Αφού ο καταχωρητής $state[p]$ εγγράφεται μόνο από τη διεργασία $p$, ο εν λόγω καταχωρητής θα περιέχει την τιμή που εγγράφηκε από την $V_i^S$ από το σημείο που επιτελέσθηκε η $w_p$ έως τουλάχιστον το σημείο όπου ο $post[seq_S][i]$ αρχικοποιήθηκε από την $S$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $r_{pre}$ εκτελείται έπειτα από την $r'_{seq}$ (βλ. Σχήμα 7.1). Από το Λήμμα 7.2 προκύ-

πτει ότι υπάρχει μία SCAN $S'$, η οποία διαβάζει τον καταχωρητή $state[p]$ στην εποχή $E_{j-1}$. Συμβολίζουμε με $r_p$ την εντολή read της $S'$ στον καταχωρητή $state[p]$, ενώ με $w_S$, συμβολίζουμε την εγγραφή του καταχωρητή $seq$ από την $S'$. Διαχωρίζουμε τις ακόλουθες περιπτώσεις.




Σχήμα 7.1 Η $r_{pre}$ εκτελείται έπειτα από την $r'_{seq}$.

a) Έστω ότι η $r'_{seq}$ έπεται της $r_p$. Εξ ορισμού, η $V_i^S$ εγγράφει στον καταχωρητή $post[seq_S][i]$, έτσι η $r'_{seq}$ πρέπει να έχει αναγνώσει τιμή ίση με $seq_S$ στον καταχωρητή $seq$. Το Λήμμα 7.5 συνεπάγεται ότι $SN_{j-1} \cap SN_j = \emptyset$. Αφού ισχύει $seq_S \in SN_j$, θα ισχύει $seq_S \notin SN_{j-1}$ (δηλαδή καμία SCAN δεν εγγράφει την τιμή $seq_S$ στον καταχωρητή $seq$ στην εποχή $E_{j-1}$). Από το Λήμμα 7.5 συμπεραίνουμε ότι κάθε SCAN της εποχής $E_j$ εγγράφει μία ξεχωριστή τιμή στον καταχωρητή $seq$. Έτσι καμία SCAN που προηγείται της $S$ στην εποχή $E_j$, δεν εγγράφει τιμή ίση με $seq_S$ στον καταχωρητή $seq$. Αφού η $r'_{seq}$ έπεται της $r_p$ και η $r_p$ έπεται της $w_S$, ο μόνος τρόπος για να αναγνώσει η $r'_{seq}$ την τιμή $seq_S$ στον καταχωρητή $seq$, είναι η $r'_{seq}$ να επιτελεσθεί μετά την $w_S$. Αφού η $r_{pre}$ έπεται της $r'_{seq}$, συνεπάγεται πως η $r_{pre}$ έπεται της $w_S$, το οποίο αντιτίθεται στην υπόθεση ότι η $r_{pre}$ προηγείται της $w_S$.

b) Έστω τώρα ότι η $r'_{seq}$ προηγείται της $r_p$. Υπενθυμίζεται ότι η $r'_{seq}$ ανέγνωσε τιμή ίση με $seq_S$ στον καταχωρητή $seq$. Υποθέτουμε ότι η τιμή $seq_S$ που αναγνώσθηκε από την $r'_{seq}$, έχει εγγραφεί στον καταχωρητή $seq$ από κάποια SCAN $S_l$ που έχει εκτελεσθεί στην εποχή $E_l$, $l \geq 1$. Αν καμία τέτοια SCAN δεν υπάρχει, η $r'_{seq}$ έχει αναγνώσει την αρχική τιμή του καταχωρητή $seq$ και έτσι θα ισχύει $seq_S = 1$. Επιπρόσθετα τότε η $r'_{seq}$ έχει εκτελεσθεί προτού την εγγραφή του καταχωρητή $seq$ από την πρώτη SCAN της εκτέλεσης (η εν λόγω SCAN είναι η πρώτη SCAN της εποχής $E_1$). Σε αυτή την περίπτωση υποθέτουμε ότι $l = 0$, $free_0 = \emptyset$ και $candidates_0 = \{2, \ldots, n + 2 * PERIODS + 1\}$ (δηλαδή με $free_0$ και $candidates_0$ συμβολίζουμε τα σύνολα $free$ και $candidates$ στην αρχική κατάσταση του συστήματος). Αφού ισχύει

$seq_S \in SN_j$, το Λήμμα 7.5 συνεπάγεται ότι $seq_S \notin SN_{j-1}$. Έτσι θα ισχύει $1 \leq l < j-1$. Έστω $w_{S_l}$ η write της $S_l$ στον καταχωρητή $seq$. Αφού η $r'_{seq}$ ανέγνωσε την τιμή που έγγραψε η $w_{S_l}$ στον καταχωρητή $seq$, η $r'_{seq}$ πρέπει να έχει επιτελεσθεί ανάμεσα στην $w_{S_l}$ και στην επόμενη εγγραφή του καταχωρητή $seq$ από τη SCAN που έπεται της $S_l$ (αφού $l < j-1$, μία τέτοια SCAN υπάρχει). Έτσι η $r'_{seq}$ εκτελείται είτε κατά τη διάρκεια της εποχής $E_l$, είτε κατά την έναρξη της εποχής $E_{l+1}$ (αυτή η περίπτωση δύναται να εμφανισθεί όταν η $S_l$ είναι η τελευταία SCAN της $E_l$). Αξίζει να σημειωθεί ότι αφού $l < j-1$, η $E_{l+1}$ είναι είτε η εποχή $E_{j-1}$ ή είτε μια πρότερη εποχή. Σε αυτό το σημείο θα αποδείξουμε τους ακόλουθους ισχυρισμούς.

**Ισχυρισμός 7.1.** *Για κάθε* $f \in \{l, \ldots, j-1\}$*, ισχύει ότι* $seq_S \notin conadidates_f$.

**Απόδειξη.** Έστω ότι $f = l$. Στην περίπτωση όπου $l = 0$ ισχύει ότι $seq_S = 1$ (η αρχική τιμή του $seq$) και $candidates_0 = \{2, \ldots, n + 2 * PERIODS + 1\}$. Έτσι θα ισχύει $seq_S \notin candidates_0$. Έστω ότι $l > 0$. Αφού η SCAN $S_l$ εκτελείται στην εποχή $E_l$ και εγγράφει στον καταχωρητή $seq$ την τιμή $seq_S$, θα ισχύει $seq_S \in SN_l$. Το Λήμμα 7.6 συνεπάγεται ότι $candidates_l \cap SN_l = \emptyset$. Έτσι θα ισχύει $seq_S \notin candidates_l$. Έστω ότι $f > l$. Υπενθυμίζεται ότι η τιμή του καταχωρητή $state[p]$ δεν μεταβάλλεται από το σημείο εκτέλεσης της εντολής $w_p$ έως το σημείο όπου η $S$ ξεκινά τη λειτουργία της. Υπενθυμίζεται επίσης ότι η $r'_{seq}$ (και επομένως η $w_p$ που προηγείται της $r'_{seq}$) εκτελείται πριν την εντολή write στον καταχωρητή $seq$ της πρώτης SCAN της εποχής $E_{l+1}$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμές 13-15), κάθε SCAN αρχικά εγγράφει τον καταχωρητή $seq$ και έπειτα διαβάζει μερικούς από τους καταχωρητές του πίνακα $state$. Το Λήμμα 7.2 συνεπάγεται ότι ο καταχωρητής $state[p]$ έχει αναγνωσθεί από μία μόνο SCAN στην εποχή $E_f$. Έτσι η τιμή $seq_S$ αναγνώσθηκε στον καταχωρητή $state[p]$ κατά τη διάρκεια της εποχής $E_f$. Έτσι από τον ψευδοκώδικα του αλγορίθμου προκύπτει πως η τιμή $seq_S$ αφαιρέθηκε από το σύνολο $candidates$ κατά τη διάρκεια της εποχής $E_f$. Από Λήμμα 7.4 προκύπτει ότι δεν προστέθηκαν στοιχεία στο σύνολο $free$ έπειτα από την εκτέλεση της γραμμής 12 από την πρώτη SCAN της εποχής $E_f$. Αφού η γραμμή 16 έπεται της γραμμής 19, συμπεραίνουμε ότι $seq_S \notin candidates_f$. ▲

**Ισχυρισμός 7.2.** *Για κάθε* $f \in \{l, \ldots, j-1\}$*, ισχύει ότι* $seq_S \notin free_f$.

Η απόδειξη θα γίνει με επαγωγή στο $f$. Αρχικά ο ισχυρισμός θα αποδειχθεί για $f = l$. Στην περίπτωση όπου ισχύει $l = 0$, έχουμε $free_0 = \emptyset$, οπότε προφανώς $seq_S \notin free_0$. Έστω ότι $l > 0$. Αφού η $S_l$ επιλέγει την τιμή $seq_S$ για να την εγγράψει στον καταχωρητή $seq$, θα ισχύει $seq_S \in SN_l$. Από το Λήμμα 7.3 δύναται να συμπεράνουμε ότι $free_l \cap SN_l = \emptyset$. Έτσι θα ισχύει $seq_S \notin free_l$.

Έστω ότι $l < f \leq j-1$, και έστω ότι ο ισχυρισμός ισχύει για $f-1$. Θα αποδείξουμε ότι ο ισχυρισμός ισχύει για $f$.

Από την επαγωγική υπόθεση ισχύει ότι $seq_S \notin free_{f-1}$. Από τον Ισχυρισμό 7.1 προκύπτει ότι $seq_S \notin candidates_{f-1}$. Συμβολίζουμε με $free_f^S$ το σύνολο $free$ έπειτα από την εκτέλεση της γραμμής 11 από την πρώτη SCAN της εποχής $E_f$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 11) προκύπτει ότι $free_f^S = free_{f-1} \cup candidates_{f-1}$. Οπότε συμπεραίνουμε πως $seq_S \notin free_f^S$. Αφού κανένα άλλο στοιχείο δεν προστίθεται στο σύνολο $free$ μέχρι το τέλος της $E_f$, θα ισχύει $seq_S \notin free_f^S$, όπως και απαιτείται. ▲

Για $f = j-1$, από τον Ισχυρισμό 7.1 συνεπάγεται ότι $seq_S \notin candidates_{j-1}$ ενώ από τον Ισχυρισμό 7.2 συνεπάγεται ότι $seq_S \notin free_{j-1}$. Από το Λήμμα 7.3 προκύπτει ότι μόνο η πρώτη SCAN της εποχής $E_j$ προσθέτει στοιχεία στο σύνολο $free$ εκτελώντας τη γραμμή 10. Συμβολίζουμε με $free_j^S$ το σύνολο $free$ έπειτα από την εκτέλεση της προαναφερθείσας εντολής. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι $free_j^S = free_{j-1} \cup candidates_{j-1}$. Έτσι θα ισχύει $seq_S \notin free_j^S$. Όλες οι SCAN της εποχής $E_j$ (συμπεριλαμβανομένης και της $S$) επιλέγουν στοιχεία στο σύνολο $free_j^S$. Ωστόσο η $S$ επιλέγει το στοιχείο $seq_S$, το οποίο δεν υπάρχει στο σύνολο $free_j^S$, το οποίο και είναι άτοπο.

Σε κάθε περίπτωση καταλήξαμε σε άτοπο, άρα το λήμμα ισχύει. ■

Η απόδειξη των υπόλοιπων λημμάτων είναι ακριβώς η ίδια με αυτή που παρουσιάζεται στο Κεφάλαιο 5.

**Θεώρημα 7.9** *Ο αλγόριθμος RT είναι μία σειριοποιήσιμη, Single-Scanner υλοποίηση ατομικών στιγμιοτύπων που πληροί την ιδιότητα ελεύθερη-αναμονής και επιτυγχάνει χρονική πολυπλοκότητα* $O(m)$ *για τη SCAN και* $O(1)$ *για την UPDATE, χρησιμοποιώντας* $O(mn)$ *καταχωρητές μεγέθους* $O(max\{log\, n\, , log|T|\})$ *bit.*

# ΚΕΦΑΛΑΙΟ 8. Ο ΑΛΓΟΡΙΘΜΟΣ C-SNAP



Σε αυτό το κεφάλαιο παρουσιάζεται ένας αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται C-Snap. Ο C-Snap αναφέρεται στη γενικότερη περίπτωση των ατομικών στιγμιοτύπων, όπου μπορούν πολλές SCAN να εκτελούνται ταυτόχρονα (Multi-Scanner). Ο C-Snap επιτυγχάνει χρονική πολυπλοκότητα $O(m)$ για τη SCAN και $O(1)$ για την UPDATE, χρησιμοποιώντας $2m+1$ καταχωρητές μεγέθους $O(m\log(|\mathrm{T}|)+\log k)$ bit, όπου $k$ είναι ο μέγιστος αριθμός λειτουργιών SCAN που εκτελούνται. Από τους καταχωρητές που χρησιμοποιεί C-Snap οι $m$ είναι ανάγνωσης-εγγραφής, ενώ οι $m+1$ είναι καταχωρητές CAS.

## 8.1. Περιγραφή του αλγορίθμου

Ο αλγόριθμος C-Snap (βλ. Αλγόριθμος 8.1 και Αλγόριθμος 8.2) βασίζεται στον αλγόριθμο T-Opt. Οι πίνακες $pre$ και $post$ παίζουν τον ίδιο ρόλο με τους πίνακες $pre$ και $post$ του αλγορίθμου T-Opt. Ωστόσο, ο πίνακας $post$ του C-snap αποτελείται από μόνο μία γραμμή $m$ καταχωρητών CAS, όπου κάθε καταχωρητής περιέχει μία τιμή και έναν σειριακό αριθμό. Η χρήση των σειριακών αριθμών αλλά και η χρήση καταχωρητών CAS μας εξασφαλίζει ότι μία UPDATE στη συνιστώσα $A_i, 1 \leq i \leq m$ θα εγγράψει μία τιμή στον $post[i]$, μόνο αν είναι αρκετά πρόσφατη. Σημειώνεται ότι η UPDATE του C-Snap λειτουργεί με παρόμοιο τρόπο με αυτή του T-Opt. Επίσης είναι αξιοσημείωτο ότι η εντολή CAS σε έναν καταχωρητή $post[i]$ από κάποια UPDATE επιτυγχάνει μόνο αν ισχύει $post[i] = null$ (γραμμές 3 και 4).

Ο αλγόριθμος C-Snap είναι μία υλοποίηση της γενικότερης περίπτωσης ατομικών στιγμιοτύπων όπου δύνανται να επιτελούνται ταυτόχρονα πολλές SCAN. Το πιο σημαντικό τμήμα μιας SCAN επιτελείται από τη συνάρτηση $grab\_scan$. Κάθε κλήση μιας $grab\_scan$ προσπαθεί να υπολογίσει ένα συνεπές διάνυσμα τιμών (γραμμές 6 και 7), με τον ίδιο ακριβώς τρόπο με αυτόν του T-Opt. Έπειτα η $grab\_scan$ προσπαθεί να αποθηκεύσει το διάνυσμα τιμών που υπολόγισε στον $seq$ (στον C-Snap ο $seq$ χρησιμοποιείται για την αποθήκευση περισσότερων πληροφοριών) επιτελώντας μία εντολή CAS (γραμμή 18). Οι εντολές CAS που εκτελούνται στη γραμμή 12 αποκαλούνται CAS τύπου 0. Ως τελευταία εργασία η $grab\_scan$ προσπαθεί να αυξήσει τον σειριακό αριθμό του καταχωρητή $seq$ χρησιμοποιώντας μία εντολή CAS. Η εν λόγω εντολή CAS (γραμμή 18) αποκαλείται CAS τύπου 1. Ο C-Snap έχει σχεδιαστεί έτσι ώστε μία επιτυχημένη CAS τύπου 0 να εναλλάσσεται με μία επιτυχημένη CAS τύπου 1. Αυτό επιτυγχάνεται χρησιμοποιώντας το πεδίο ενός bit που ονομάζεται $grab$, του καταχωρητή $seq$. Η CAS τύπου 0 επιτυγχάνει μόνο αν η τιμή του $grab$ είναι ίση με $true$ (γραμμή 11), και η CAS την αλλάζει σε $false$ (γραμμή 12). Αντίθετα μία CAS τύπου 1 επιτυγχάνει μόνο αν η τιμή του πεδίου $grab$ είναι ίση με $false$ (γραμμή 17), και η CAS την αλλάζει σε $true$ (γραμμή 18). Επιπρόσθετα μία CAS τύπου 0 τροποποιεί μόνο το πεδίο $view$ του καταχωρητή $seq$, ενώ μία CAS τύπου 1 τροποποιεί μόνο το πεδίο $tm$ του καταχωρητή $seq$ εγγράφοντας έναν νέο σειριακό αριθμό (γραμμές 14, 18).

Κάθε εκτέλεση του C-Snap χωρίζεται σε φάσεις ως εξής. Η πρώτη φάση ξεκινά με την αρχική καθολική κατάσταση του συστήματος. Μία φάση τελειώνει πριν την εγγραφή ενός νέου σειριακού αριθμού στο $seq.tm$ από μία επιτυχημένη CAS τύπου 1, ενώ η επόμενη φάση ξεκινά από την εν λόγω CAS. Ο ψευδοκώδικας του αλγορίθμου έχει σχεδιαστεί έτσι ώστε κάθε φάση να «προσομοιώνει» την εκτέλεση μίας SCAN toy Single-Scanner αλγορίθμου T-Opt.

Αλγόριθμος 8.1 Ψευδοκώδικας για τις SCAN και UPDATE του αλγορίθμου C-Snap.

```
   struct sequence{
       int tm;
       boolean grab;
       data view[1..m];
   }

   struct post_data{
       int tm;
       data value;
   }

   shared sequence seq=<1, true, <null, ..., null>>;
   shared post_data post[1..m]={<0, null>, ..., <0, null>};
   shared data pre[1..m]={null, ..., null};

   void UPDATE(data v, int i){
       sequence curr_seq;
       data p1;
       post_data p2, lpost;

1.     curr_seq=seq;
2.     p1=pre[i];
3.     lpost=<curr_seq.tm-1, null>;
4.     CAS(post[i], lpost, <lpost.tm, p1>);
5.     pre[i]=v;
   }

   data[] SCAN(void)
6.     grab_scan();
7.     grab_scan();
8.     return seq.view;
   }

   void grab_scan(void){
       sequence curr_seq;
       data view[1..m], lview[1..m];
       int ltm;

9.     curr_seq=seq;
10.    view=collect_view();
11.    if(curr_seq.grab==true)
12.        CAS(seq, curr_seq, <curr_seq.tm, false, view>);
13.    clear_registers(curr_seq);
14.    ltm=curr_seq.tm+1;
15.    curr_seq.grab=false;
16.    lview=seq.view;
17.    curr_seq.view=lview;
18.    CAS(seq, curr_seq, <ltm, true, lview>);
   }
```

**Αλγόριθμος 8.2** Ψευδοκώδικας για την $clear\_registers$ και την $collect\_view$.

```
   void clear_registers(sequence s){
       int i;
       post_data p, lpost;

19.    for(i=1;i≤m;i++){
20.        p=post[i];
21.        lpost=<s.tm-1, p.value>;
22.        CAS(post[i], lpost, <s.tm, null>);
23.        p=post[i];
24.        lpost=<s.tm-1, p.value>;
25.        CAS(post[i], lpost, <s.tm, null>);
       }
}

   data[] collect_view(void){
       int i;
       data view[1..m], p1, p2;

26.    for(i=1;i≤m;i++){
27.        p1=pre[i];
28.        p2=post[i].value;
29.        if(p2==null) view[i]=p1;
30.        else view[i]=p2;
      }
31.    return view;
   }
```

Η χρήση σειριακών αριθμών και εντολών CAS εξασφαλίζει ότι καμία SCAN και καμία UPDATE που ξεκίνησε τη λειτουργία της σε μία πρωτύτερη φάση δε θα επιτύχει να τροποποιήσει τα περιεχόμενα κάποιου καταχωρητή ($post$ ή $seq$) σε κάποια επόμενη φάση. Μόνο SCAN που ξεκίνησαν τη λειτουργία τους στην τρέχουσα φάση δύναται να εγγράψουν στον καταχωρητή $seq$ και μόνο μία από αυτές θα επιτελέσει επιτυχώς μία εντολή CAS (λόγω του πεδίου $grab$) εγγράφοντας ένα διάνυσμα τιμών στον $seq$. Αφότου επιτελεστούν τα προαναφερθέντα, το σύστημα μπαίνει σε μία φάση αρχικοποίησης των καταχωρητών του πίνακα $post$, η οποία τελειώνει με την έναρξη της επόμενης φάσης. Αφότου το πεδίο $value$ κάποιου καταχωρητή $post$ αλλάξει σε $null$ κατά τη διάρκεια της τρέχουσας φάσης, η χρήση σειριακών αριθμών και εντολών CAS μας εξασφαλίζουν ότι καμία εντολή CAS δε θα τροποποιήσει τον καταχωρητή έως ότου ξεκινήσει μία νέα φάση. Έτσι οι $m$ καταχωρητές του πίνακα $post$ έχουν αποθηκευμένη την $null$ στο πεδίο $value$ πριν την έναρξη της νέας φάσης. Α-

φότου εγγραφεί ο σειριακός αριθμός της νέας φάσης στον $seq$ είναι εγγυημένο ότι δε θα επιτύχει κάποια εντολή CAS κάποιας συνάρτησης $clear\_registers$ της τρέχουσας ή παλαιότερης φάσης, ώστε να πανωγράψει χρήσιμες τιμές στο $post[i].value$. Έτσι αν η συνάρτηση $grab\_scan$ επιτύχει να εγγράψει ένα νέο διάνυσμα τιμών στην τρέχουσα φάση και διαβάσει μία τιμή διαφορετική του $null$ στον καταχωρητή $post[i]$, τότε η εν λόγω τιμή έχει εγγραφεί από κάποια UPDATE στη συνιστώσα $A_i$ που έχει ξεκινήσει την εκτέλεσή της στην τρέχουσα φάση. Για τέτοιες συνιστώσες, η $grab\_scan$ χρησιμοποιεί τις τιμές που βρέθηκαν στον καταχωρητή $post[i]$ (όπως γίνεται από τον T-Opt).

Μία SCAN $S$ επιστρέφει το διάνυσμα τιμών που περιέχεται στον καταχωρητή $seq$ εκτελώντας τη γραμμή 8. Το εν λόγω διάνυσμα τιμών έχει εγγραφεί από την τελευταία επιτυχημένη CAS τύπου 0 (συμβολίζουμε ως $C_0^S$ την εν λόγω εντολή CAS) που προηγείται της εκτέλεσης της γραμμής 8 από την $S$. Αφού η $C_0^S$ είναι τύπου 0, δεν τροποποιεί το $seq.tm$, του οποίου η τιμή έχει εγγραφεί από την τελευταία CAS τύπου 1 που προηγείται της $C_0^S$ (συμβολίζουμε με $C_1^S$ την εν λόγω εντολή CAS). Για τον αλγόριθμο C-Snap, η $C_1^S$ παίζει τον ίδιο ρόλο με την εντολή $w_S$ του T-Opt. Για να εξασφαλισθεί ότι η $C_1^S$ είναι εντός του διαστήματος εκτέλεσης της $S$, η $grab\_scan$ καλείται δύο φορές από την $S$ (γραμμές 6, 7). Αποδεικνύεται ότι στο διάστημα εκτέλεσης κάθε $grab\_scan$ εμπεριέχει μία επιτυχημένη CAS τύπου 1. Επίσης αποδεικνύεται ότι ανάμεσα στις δύο επιτυχημένες CAS τύπου 1, έχει εκτελεσθεί επιτυχώς μία CAS τύπου 0. Έτσι από τον ορισμό της $C_1^S$, προκύπτει ότι η $C_1^S$ επιτελείται εντός του διαστήματος εκτέλεσης της S.

### 8.2. Χρονική και χωρική πολυπλοκότητα του αλγορίθμου

Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι η χρονική πολυπλοκότητα της UPDATE είναι $O(1)$, ενώ η χρονική πολυπλοκότητα της SCAN είναι $O(m)$. Ο αλγόριθμος C-Snap χρησιμοποιεί $m + 1$ καταχωρητές CAS και $m$ καταχωρητές ανάγνωσης-εγγραφής. Όλοι οι καταχωρητές ανάγνωσης-εγγραφής αποθηκεύουν μόνο μία λέξη ενός συνόλου $T$ μεγέθους $O(\log|T|)$ bit. Οι $m$ καταχωρητές CAS του πίνακα $post$ αποθηκεύουν μία λέξη του συνόλου $T$ και έναν σειριακό αριθμό, ο οποίος δύνα-

ται να είναι τόσο μεγάλος όσο και το πλήθος $k$ των SCAN που θα εκτελεστούν. Άρα οι $m$ καταχωρητές CAS του πίνακα $post$ θα έχουν μέγεθος $O(\log|T| + \log k)$ bit. Ο καταχωρητής $seq$ αποθηκεύει ένα διάνυσμα $m$ τιμών του συνόλου $T$, έναν σειριακό αριθμό ανάλογο του $k$ και το πεδίο $grab$ που έχει μέγεθος ενός bit. Άρα το συνολικό μέγεθος του $seq$ θα είναι $O(m\log|T| + \log k)$ bit.

### 8.3. Απόδειξη της σειριοποιησιμότητας του αλγορίθμου

Έστω $a$ μία οποιαδήποτε εκτέλεση του C-Snap και έστω $S$ μία οποιαδήποτε SCAN που επιτελείται στην $a$. Υπενθυμίζουμε ότι αποκαλούμε τις εντολές CAS της γραμμής 12 του ψευδοκώδικα ως CAS τύπου 0, ενώ τις εντολές CAS της γραμμής 18 τις αποκαλούμε ως CAS τύπου 1. Η $S$ επιστρέφει το διάνυσμα τιμών που ανέγνωσε στον $seq$ (γραμμή 8). Συμβολίζουμε με $C_0^S$ την τελευταία επιτυχημένη CAS τύπου 0 που έγγραψε το διάνυσμα τιμών που επέστρεψε η $S$, ενώ με $C_1^S$ συμβολίζουμε την επιτυχημένη CAS τύπου 1 που προηγείται της $C_0^S$. Συμβολίζουμε με $g_0^S$ τη συνάρτηση $grab\_scan$ που επιτελεί την $C_0^S$. Έστω $i$ ένας οποιοσδήποτε ακέραιος, $1 \leq i \leq m$. Στην περίπτωση όπου η $g_0^S$ διάβασε τιμή ίση με $null$ στον $post[i]$ και μια τιμή $v_i$ στον $pre[i]$, συμβολίζουμε με $U_i^S$ την UPDATE στη συνιστώσα $A_i$ που εγγράφει τελευταία την $v_i$ στον $pre[i]$ (γραμμή 5) πριν η $g_0^S$ αναγνώσει τον εν λόγω καταχωρητή. Στην περίπτωση όπου η $g_0^S$ διάβασε τιμή ίση με $v_i \neq null$ στον $post[i]$, συμβολίζουμε με $V_i^S$ την UPDATE της οποίας η εντολή CAS στον $post[i]$, είναι η τελευταία επιτυχημένη εντολή CAS στον $post[i]$ πριν την ανάγνωση αυτού από την $g_0^S$. Συμβολίζουμε με $U_i^S$ την UPDATE στη συνιστώσα $A_i$ που εγγράφει τελευταία στον $pre[i]$, πριν η $V_i^S$ τον αναγνώσει. Συμβολίζουμε με $w_i^S$ την εντολή write στον $pre[i]$ που επιτελείται από την $U_i^S$. Συμβολίζουμε με $r_i^S$ την εντολή read στον $pre[i]$ από την $g_0^S$ και με $\tilde{r}_i^S$ τη εντολή read της $g_0^S$ στον $post[i]$. Αν η $g_0^S$ ανέγνωσε τιμή ίση με $v_i \neq null$ στο $post[i].value$, συμβολίζουμε με $r_{pre}$ τη εντολή read της $V_i^S$ στον $pre[i]$ (γραμμή 2). Συμβολίζουμε με $a^C$ την υπακολουθία της $a$ που περιέχει όλες τις επιτυχημένες εντολές CAS (τύπου 0 και τύπου 1) στον καταχωρητή $seq$ με τη σειρά που εμφανίσθηκαν. Συμβολίζουμε με $|a^C|$ τον αριθμό των εντολών CAS που περιέχει η $a^C$. Συμβολίζουμε με $a_0^C$ ( $a_1^C$) την υπακολουθία της $a$ που περιέχει όλες τις επιτυ-

χημένες CAS τύπου 0 (τύπου 1 αντίστοιχα) στον καταχωρητή $seq$ με τη σειρά που εμφανίσθηκαν. Συμβολίζουμε με $|a_0^C|$ ( $|a_1^C|$ αντίστοιχα) τον αριθμό των εντολών CAS που περιέχει η $a_0^C$ ( $a_1^C$ αντίστοιχα).

Σε αυτό το σημείο θα θέσουμε σημεία σειριοποίησης για τις SCAN και UPDATE.

- Κάθε SCAN $S$ σειριοποιείται στην $C_1^S$.
- Για κάθε $i \in \{1, \ldots, m\}$, αν η $w_i^S$ έπεται της $C_1^S$ (και η $U_i^S$ δεν έχει ήδη σειριοποιηθεί)[4], τοποθετούμε σημείο σειριοποίησης στην $U_i^S$ ακριβώς πριν την $C_1^S$. Επίσης τοποθετούμε σημείο σειριοποίησης σε κάθε UPDATE στη συνιστώσα $A_i$ που επιτελεί την write της στον καταχωρητή $pre[i]$ μεταξύ της $C_1^S$ και της $w_i^S$ (και δεν έχει ήδη σειριοποιηθεί), ακριβώς πριν την $C_1^S$. Αν σειριοποιούνται ακριβώς πριν την $C_1^S$ περισσότερες από μία UPDATE, τότε αυτές σειριοποιούνται με τη σειρά με την οποία επιτέλεσαν την write τους στον $pre$.
- Αφότου αναθέσουμε σημεία σειριοποίησης σε όλες τις SCAN, καθώς και σε κάποιες UPDATE (όπως περιγράφηκε παραπάνω), τοποθετούμε σημείο σειριοποίησης σε όλες τις υπόλοιπες UPDATE στο σημείο που επιτέλεσαν την write τους στον $pre$.

Για να αποδειχθεί η ορθότητα του αλγορίθμου C-Snap είναι απαραίτητο να αποδειχθούν όλοι οι ισχυρισμοί που αναφέρθηκαν κατά την περιγραφή του αλγορίθμου. Έτσι η απόδειξη της ορθότητας του αλγορίθμου C-Snap είναι αρκετά πιο πολύπλοκη και μεγάλη σε σχέση με αυτές των υπόλοιπων αλγορίθμων.

Το ακόλουθο λήμμα είναι άμεση συνεπαγωγή του ορισμού της $U_i^S$. Από το εν λόγω λήμμα προκύπτει ότι η $w_i^S$ προηγείται της $C_0^S$, άρα η $w_i^S$ προηγείται και του τέλους της $g_0^S$.

[4]Δύναται για δύο SCAN $S$ και $S'$ να ισχύει $g_0^S = g_0^{S'}$, έτσι ώστε και οι δύο SCAN να καθορίζουν τα σημεία σειριοποίησης των ίδιων UPDATE. Για το λόγο αυτό η παρένθεση είναι απαραίτητη.

**Λήμμα 8.1** *Για κάθε* $i \in \{1, \ldots, m\}$ *ισχύει:*

1. *η* $\tilde{r}_i^S$ *έπεται της* $w_i^S$*, και*
2. *η* $C_0^S$ *έπεται της* $w_i^S$*.*

**Απόδειξη.** Αρχικά υποθέτουμε ότι η $S$ διάβασε τιμή ίση με $null$ στον καταχωρητή $post[i]$. Από τον τρόπο ορισμού της $U_i^S$ προκύπτει ότι η $w_i^S$ προηγείται της $r_i^S$ (αφού η $r_i^S$ ανέγνωσε την τιμή που η $w_i^S$ έγγραψε στον καταχωρητή $pre[i]$). Αφού η $r_i^S$ προηγείται της $\tilde{r}_i^S$ και η $\tilde{r}_i^S$ προηγείται της $C_0^S$, είναι προφανές ότι και οι δύο ισχυρισμοί του λήμματος ισχύουν.

Ας υποθέσουμε ότι η $S$ ανέγνωσε τιμή ίση με $v_i \neq null$ στον καταχωρητή $post[i]$. Σε αυτή την περίπτωση η λειτουργία $V_i^S$ είναι καλά ορισμένη. Έστω $r_{pre}$ η εντολή read της λειτουργίας $V_i^S$ στον καταχωρητή $pre[i]$. Από τον τρόπο ορισμού της $V_i^S$ και της $U_i^S$ τα ακόλουθα ισχύουν: (1) η $V_i^S$ διαβάζει στον καταχωρητή $pre[i]$ την τιμή που η $w_i^S$ έγγραψε σε αυτόν, έτσι η $r_{pre}$ επιτελείται έπειτα από την $w_i^S$, και (2) η $\tilde{r}_i^S$ διαβάζει στον καταχωρητή $post[i]$ την τιμή που έγγραψε η $V_i^S$ σε αυτόν, έτσι η $\tilde{r}_i^S$ έπεται της εντολή CAS που η $V_i^S$ επιτέλεσε στον καταχωρητή $post[i]$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η εντολή CAS της $V_i^S$ έπεται της $r_{pre}$. Έτσι η $\tilde{r}_i^S$ έπεται της $w_i^S$, όπως και απαιτείται από τον ισχυρισμό (1). Αφού η $g_0^S$ εκτελεί πρώτα την $\tilde{r}_i^S$ και έπειτα την $C_0^S$, προκύπτει ότι η $C_0^S$ έπεται της $w_i^S$, όπως και απαιτείται από τον ισχυρισμό (2). ∎

Έστω $a$ μία οποιαδήποτε εκτέλεση του C-Snap και έστω $a^C = C_0^S C_1^S C_2^S \ldots C_k^S$, όπου $k = |a^C| - 1$. Με το ακόλουθο λήμμα αποδεικνύουμε ότι στην εκτέλεση $a$, οι CAS τύπου 0 εναλλάσσονται με CAS τύπου 1, ενώ η πρώτη εντολή CAS της εκτέλεσης $a$ είναι τύπου 0.

**Λήμμα 8.2** *Για κάθε* $i \in \{0, \ldots, k\}$*, η* $C_i^S$ *είναι τύπου* $(i \bmod 2)$.

**Απόδειξη.** Η απόδειξη του λήμματος θα γίνει με επαγωγή στο $i$.

**Βάση Επαγωγής.** Θα αποδείξουμε ότι η $C_0^S$ είναι τύπου 0. Η $C_0^S$ είναι η πρώτη επιτυχημένη εντολή CAS στον $seq$ της εκτέλεσης $a$. Χρησιμοποιώντας τη μέθοδο της εις

άτοπο απαγωγής υποθέτουμε ότι η $C_0^S$ είναι τύπου 1. Πριν την εκτέλεση της $C_0^S$ εκτελείται η εντολή $curr\_seq.grab = false$ (γραμμή 15). Αφού η $C_0^S$ είναι μία επιτυχημένη CAS τύπου 1, πρέπει να ισχύει $seq.grab = false$ ακριβώς πριν την $C_0^S$. Όμως η αρχική τιμή του $seq.grab$ είναι ίση με $true$, ενώ τα δεδομένα του $seq$ μεταβάλλονται μόνο από επιτυχημένες εντολές CAS. Έτσι η $C_0^S$ δε μπορεί να είναι μία επιτυχημένη CAS τύπου 1, το οποίο και είναι άτοπο.

**Επαγωγική υπόθεση.** Έστω $i, 1 \leq i \leq k$. Υποθέτουμε ότι το λήμμα ισχύει για $i-1$, δηλαδή η $C_{i-1}^S$ είναι τύπου $((i-1) mod 2)$.

**Επαγωγικό Βήμα.** Θα αποδείξουμε ότι το λήμμα ισχύει για $i$. Έστω $p'$ η διεργασία που επιτελεί την $C_{i-1}^S$ και έστω $p$ η διεργασία που επιτελεί την $C_i^S$ (είναι δυνατόν να ισχύει $p = p'$). Συμβολίζουμε με $g_p$ την $grab\_scan$ που επιτελεί την $C_i^S$. Διακρίνουμε τις ακόλουθες δύο περιπτώσεις.

1) Έστω ότι η $C_{i-1}^S$ είναι τύπου 0. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $C_i^S$ είναι και αυτή τύπου 0. Σε αυτή την περίπτωση πρέπει η $g_p$ να διάβασε τιμή ίση με $true$ στο $seq.grab$ (γραμμή 9), γιατί σε αντίθετη περίπτωση η συνθήκη $if$ της γραμμής 11 θα είχε αποτιμηθεί σε $false$ (και έτσι η $C_i^S$ δε θα είχε επιτελεσθεί). Άρα πρέπει να ισχύει $curr\_seq.grab = true$ κατά την εκτέλεση της $C_i^S$. Αφού η $C_{i-1}^S$ είναι τύπου 0, όλες οι επιτυχημένες CAS τύπου 0 εγγράφουν τιμή ίση με $false$ στο $seq.grab$, και καμία επιτυχημένη εντολή CAS δεν επιτελείται στον $seq$ μεταξύ της $C_{i-1}^S$ και της $C_i^S$, η τιμή του $seq.grab$ είναι ίση με $false$ όταν η $C_i^S$ εκτελείται. Έτσι η $C_i^S$ δε δύναται να είναι μία επιτυχημένη CAS τύπου 0, το οποίο αντιτίθεται στην υπόθεση ότι η $C_i^S$ είναι τύπου 0.
2) Έστω ότι η $C_{i-1}^S$ είναι τύπου 1. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $C_i^S$ είναι και αυτή τύπου 1. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 15) προκύπτει ότι ισχύει $curr\_seq.grab = false$ όταν η $C_i^S$ εκτελείται. Αφού η $C_{i-1}^S$ είναι τύπου 1, όλες οι επιτυχημένες CAS τύπου 1 εγγράφουν τιμή ίση με $true$ στο $seq.grab$, και αφού καμία επιτυχημένη CAS στον $seq$ δεν εκτελείται ανάμεσα στην $C_{i-1}^S$ και στη $C_i^S$, ισχύει ότι η τιμή του $seq.grab$ θα είναι ίση με $true$ κατά την εκτέλεση της $C_i^S$. Έτσι η $C_i^S$ δε δύναται

να είναι μία επιτυχημένη εντολή CAS, το οποίο αντιτίθεται στην υπόθεση ότι η $C_i^S$ είναι τύπου 0.

Και στις δύο περιπτώσεις οδηγηθήκαμε σε άτοπο, άρα το λήμμα ισχύει. ■

Με το ακόλουθο λήμμα αποδεικνύεται ότι το πεδίο $tm$ του $seq$ αλλάζει μόνο από επιτυχημένες CAS τύπου 1, ενώ το πεδίο $view$ του ιδίου καταχωρητή αλλάζει μόνο από επιτυχημένες CAS τύπου 0.

**Λήμμα 8.3** *Ισχύουν οι ακόλουθοι ισχυρισμοί:*

1. *Η $C_0^S$ εγγράφει στο στο $seq.tm$ την αρχική τιμή του.*
2. *Για κάθε $i, 1 \leq i \leq k$ ισχύει:*
   a. *Αν $i \bmod 2 = 0$, η $C_i^S$ αποθηκεύει στο $seq.tm$ την ίδια τιμή με την $C_{i-1}^S$.*
   b. *Αν $i \bmod 2 = 1$, η $C_i^S$ αποθηκεύει στο $seq.view$ το ίδιο διάνυσμα τιμών με την $C_{i-1}^S$.*

**Απόδειξη.** Αρχικά θα αποδείξουμε τον ισχυρισμό 1. Υπενθυμίζουμε ότι η $C_0^S$ είναι η πρώτη εντολή CAS στον $seq$ στην εκτέλεση $a$, άρα ο $seq$ έχει την αρχική του τιμή πριν η $C_0^S$ εκτελεστεί. Συμβολίζουμε με $p$ τη διεργασία που εκτελεί την $C_0^S$. Η $p$ διαβάζει στον καταχωρητή $seq$ την αρχική τιμή του (γραμμή 9). Από το Λήμμα 8.2 προκύπτει ότι η $C_0^S$ είναι τύπου 0. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 9), η $C_0^S$ εγγράφει στο $seq.tm$ την τιμή του $curr\_seq.tm$. Έτσι η $C_0^S$ εγγράφει στο $seq.tm$ την αρχική τιμή, όπως και απαιτείται από τον ισχυρισμό 1.

Σε αυτό το σημείο θα αποδειχθεί ο ισχυρισμός 2/a του λήμματος. Έστω οποιοδήποτε $1 \leq i \leq k$, τέτοιο ώστε να ισχύει $i \bmod 2 = 0$. Από το Λήμμα 8.2 προκύπτει ότι η $C_i^S$ είναι τύπου 0. Υποθέτουμε ότι η $C_{i-1}^S$ εγγράφει τιμή ίση με $t$ στο $seq.tm$. Αφού δεν υπάρχουν επιτυχημένες CAS ανάμεσα στην $C_{i-1}^S$ και στην $C_i^S$, κατά την εκτέλεση της $C_i^S$ θα ισχύει $seq.tm = t$. Επειδή η $C_i^S$ είναι μία επιτυχημένη εντολή CAS τύπου 0, από τον ψευδοκώδικα του αλγορίθμου (γραμμή 9) προκύπτει ότι θα ισχύει $curr\_seq.tm = t$, όπως και απαιτείται από τον ισχυρισμό 2/a του λήμματος.

Τέλος θα αποδειχθεί ο ισχυρισμός 2/b του λήμματος. Έστω οποιοδήποτε $1 \leq i \leq k$, τέτοιο ώστε να ισχύει $i \bmod 2 = 1$. Από το Λήμμα 8.2 προκύπτει ότι η εντολή $C_i^S$

είναι τύπου 1. Υποθέτουμε ότι η $C_{i-1}^S$ εγγράφει το διάνυσμα τιμών $\vec{v}$ στο $seq.view$. Αφού δεν υπάρχουν επιτυχημένες εντολές CAS ανάμεσα στην $C_{i-1}^S$ και στην $C_i^S$, κατά την εκτέλεση της $C_i^S$ θα ισχύει $seq.view = \vec{v}$. Επειδή η $C_i^S$ είναι μία επιτυχημένη εντολή CAS, από τον ψευδοκώδικα του αλγορίθμου (γραμμή 18) προκύπτει ότι κατά την εκτέλεση της $C_i^S$ ισχύει $curr\_seq.view = seq.view$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 16) ισχύει ότι $curr\_seq.view = lview$. Έτσι η $C_i^S$ δε μεταβάλλει την τιμή του $seq.view$, όπως και απαιτείται από τον ισχυρισμό 2/b του λήμματος. ∎

Στη συνέχεια θα αποδειχθεί ότι κάθε επιτυχημένη CAS τύπου 1 αυξάνει την τιμή του $seq.tm$ κατά 1.

**Λήμμα 8.4** *Έστωσαν $C_1$ και $C'_1$ δύο επιτυχημένες CAS τύπου 1 στην a, τέτοιες ώστε να μην υπάρχει καμία CAS τύπου 1 ανάμεσα στην $C_1$ και στην $C'_1$. Αν η $C_1$ εγγράψει την τιμή $t$ στο $seq.tm$, τότε η $C'_1$ εγγράφει την τιμή $t+1$ στο $seq.tm$.*

**Απόδειξη.** Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι η $C'_1$ δεν εγγράφει την τιμή $t+1$ στο $seq.tm$. Έστω $g'$ η $grab\_scan$ που επιτελεί την $C'_1$ και έστω $t'$ η τιμή που η $g'$ αναγνώσκει στο $seq.tm$ (γραμμή 9). Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $C'_1$ εγγράφει την τιμή $t'+1 \neq t+1$ στο $seq.tm$. Από το Λήμμα 8.3 προκύπτει ότι το $seq.tm$ αλλάζει μόνο από CAS τύπου 1. Αφού καμία CAS τύπου 1 δεν έχει επιτελεσθεί μεταξύ της $C_1$ και της $C'_1$, κατά την επιτέλεση της $C'_1$ ισχύει ότι $seq.tm = t$. Έτσι η $C'_1$ δε δύναται να είναι μία επιτυχημένη εντολή CAS, το οποίο αντιτίθεται στην υπόθεση ότι η $C'_1$ δεν εγγράφει την τιμή $t+1$ στο $seq.tm$. ∎

Υποθέτουμε ότι ισχύει $|a_1^c| = k_1$. Έστω $a_1^c = C_1^1 C_1^2 \dots C_1^{k_1}$. Συμβολίζουμε με $C_1^0$ την αρχική καθολική κατάσταση (configuration) του συστήματος. Υπενθυμίζουμε ότι η αρχική τιμή του $seq.tm$ είναι ίση με 1. Από το Λήμμα 8.3 προκύπτει ότι η εν λόγω τιμή δεν αλλάζει μέχρι και την επιτέλεση της $C_1^1$. Από το Λήμμα 8.3 προκύπτει ότι το $seq.tm$ αλλάζει μόνο από επιτυχημένες CAS τύπου 1. Τα προαναφερθέντα καθώς και το Λήμμα 8.4 συνεπάγονται το ακόλουθο πόρισμα.

**Πόρισμα 8.5** Για κάθε $j, 1 \leq j \leq k_1$, ισχύει ότι $seq.tm = j$ σε κάθε χρονική στιγμή ανάμεσα στην $C_1^{j-1}$ και στην $C_1^j$.

Έστω $S$ μία οποιαδήποτε SCAN που έχει εκτελεστεί από κάποια διεργασία $p$ και έστω $g$ οποιαδήποτε από τις δύο $grab\_scan$ συναρτήσεις που η $S$ εκτελεί. Συμβολίζουμε με $r_{seq}$ την εντολή read που επιτελεί η $g$ στον καταχωρητή $seq$ (γραμμή 9), ενώ με $C_1$ συμβολίζουμε την CAS τύπου 1 που εκτελείται από την $g$ (γραμμή 18). Προκειμένου να αποδειχθεί ότι κάθε SCAN σειριοποιείται εντός του διαστήματος εκτέλεσής της, αρχικά αποδεικνύουμε ότι μία τουλάχιστον CAS τύπου 1 επιτελείται στο διάστημα εκτέλεσης της $g$.

**Λήμμα 8.6** *Το διάστημα εκτέλεσης της $g$ περιέχει τουλάχιστον μία επιτυχημένη CAS τύπου 1.*

**Απόδειξη.** Υπενθυμίζουμε ότι η $g$ ξεκινά τη λειτουργία της εκτελώντας την $r_{seq}$ και τερματίζει εκτελώντας την $C_1$. Αν η $C_1$ είναι μία επιτυχημένη εντολή CAS, τότε το λήμμα ισχύει. Υποθέτουμε ότι η $C_1$ είναι μία αποτυχημένη εντολή CAS. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι δεν υπάρχει καμία επιτυχημένη CAS τύπου 1 μεταξύ της $r_{seq}$ και της $C_1$. Διακρίνουμε τις ακόλουθες περιπτώσεις.

Έστω ότι η $C_1$ αποτυγχάνει λόγω του ότι κατά την εκτέλεσή της ισχύει $curr\_seq.tm \neq seq.tm$. Από το Λήμμα 8.3 προκύπτει ότι η τιμή του $seq.tm$ μεταβάλλεται μόνο από την εκτέλεση κάποιας επιτυχημένης εντολή CAS τύπου 1. Έτσι πρέπει να υπάρχει κάποια επιτυχημένη CAS τύπου 1 μεταξύ της $r_{seq}$ και της $C_1$. Αυτό είναι άτοπο καθώς υποθέσαμε πως δεν υπάρχει κάποια επιτυχημένη CAS τύπου 1 μεταξύ της $r_{seq}$ και της $C_1$.

Υποθέτουμε ότι ισχύει $curr\_seq.tm = seq.tm$. Έστω ότι η $C_1$ αποτυγχάνει επειδή κατά την εκτέλεσή της ισχύει $curr\_seq.grab \neq seq.grab$. Αφού η $C_1$ είναι μία CAS τύπου 1, από τον ψευδοκώδικα του αλγορίθμου (γραμμή 15), προκύπτει ότι κατά την εκτέλεση της $C_1$ ισχύει $curr\_seq.grab = false$. Έτσι κατά την εκτέλεση της $C_1$ πρέπει να ισχύει $seq.grab = true$. Από τον ψευδοκώδικα του αλγορίθμου συ-

μπεραίνουμε ότι μόνο οι CAS τύπου 1 εγγράφουν τιμή ίση με $true$ στο $seq.grab$. Έτσι η τελευταία εντολή CAS που εκτελέστηκε στον $seq$ πριν την $C_1$ πρέπει να είναι τύπου 1. Έστω $C'_1$ η εν λόγω εντολή CAS. Αν η $C'_1$ εκτελέστηκε μεταξύ της $r_{seq}$ και της $C_1$, τότε το λήμμα ισχύει. Έτσι ας υποθέσουμε ότι η $C'_1$ εκτελέστηκε πριν την $r_{seq}$. Άρα η $C'_1$ είναι η τελευταία CAS τύπου 1 που εκτελέστηκε πριν από την $C_1$, η $r_{seq}$ πρέπει να έχει διαβάσει τιμή ίση με $true$ στο $seq.grab$ (γραμμή 9). Άρα η συνθήκη $if$ της γραμμής 11 πρέπει να έχει αποτιμηθεί σε $true$ και η CAS τύπου 0 της γραμμής 12 να έχει εκτελεσθεί από την $g$ (έστω $C_0$ η εν λόγω εντολή CAS). Αφού η $C'_1$ είναι η τελευταία επιτυχημένη CAS στον καταχωρητή $seq$ πριν την $C_1$ και είναι τύπου 1, καμία CAS τύπου 0 δεν έχει εκτελεστεί επιτυχώς ανάμεσα στην $r_{seq}$ και στην $C_0$, έτσι κατά την εκτέλεση της $C_0$ πρέπει να ισχύει $curr\_seq = seq$. Από τον ψευδοκώδικα προκύπτει ότι η $C_0$ πρέπει να είναι μία επιτυχημένη εντολή CAS. Αυτό είναι άτοπο καθώς η τελευταία επιτυχημένη εντολή CAS που έχει εκτελεστεί στον καταχωρητή $seq$ πριν την $C_1$ είναι η $C'_1$.

Ας υποθέσουμε τέλος ότι ισχύει $curr\_seq.grab = seq.grab$ και $curr\_seq.tm = seq.tm$ κατά την εκτέλεση της $C_1$. Αφού η $C_1$ αποτυγχάνει, πρέπει να ισχύει $curr\_seq.view \neq seq.view$. Από το Λήμμα 8.3 προκύπτει ότι η τιμή του $seq.view$ αλλάζει μόνο από επιτυχημένες CAS τύπου 0. Αφού η $C_1$ αποτυγχάνει, μία επιτυχημένη CAS τύπου 0 (έστω $C'_0$) εκτελείται μεταξύ των γραμμών 16 και 18 που εκτελεί η $g$. Είναι προφανές ότι η $C'_0$ εκτελείται από κάποια διεργασία $p' \neq p$. Το Λήμμα 8.2 συνεπάγεται ότι η τελευταία επιτυχημένη εντολή CAS στον $seq$ πριν την $C'_0$ είναι τύπου 1 (έστω $C'_1$ η εν λόγω εντολή CAS). Αν η $C'_1$ έχει εκτελεστεί μεταξύ της $r_{seq}$ και της $C_1$, τότε το λήμμα ισχύει. Έτσι υποθέτουμε ότι η $C'_1$ έχει εκτελεστεί πριν την $r_{seq}$. Υπενθυμίζουμε ότι δεν υπάρχει καμία επιτυχημένη εντολή CAS στον $seq$ μεταξύ της $r_{seq}$ και της $C'_0$ (αφού η $C'_1$ εγγράφει τιμή ίση με $true$ στο $seq.grab$, η $r_{seq}$ πρέπει να έχει αναγνώσει τιμή ίση με $true$ στο $seq.grab$). Έτσι η συνθήκη $if$ της γραμμής 11 αποτιμάται σε $true$ και η CAS τύπου 0 της γραμμής 12 (έστω $C_0$ αυτή) εκτελείται από την $p$. Επιπρόσθετα αφού η $C_0$ προηγείται της $C'_0$, δεν υπάρχει κάποια επιτυχημένη εντολή CAS στον καταχωρητή $seq$ μεταξύ της $r_{seq}$ και της $C_0$. Έτσι κατά την εκτέλεση της $C_0$ ισχύει $curr\_seq = seq$. Επομένως η $C_0$ είναι μία επιτυχημένη CAS τύπου 0. Η $C_0$ επιτελείται πριν από την $C'_0$, αφού η $C'_0$ επιτελείται μεταξύ της

εκτέλεσης της γραμμής 16 και της γραμμής 18 από την $p$. Αυτό είναι άτοπο καθώς η τελευταία επιτυχημένη CAS στον καταχωρητή $seq$ που προηγείται της $C'_0$, είναι η $C'_1$ που είναι μία CAS τύπου 1. ∎

Στο ακόλουθο λήμμα αποδεικνύεται ότι κάθε SCAN σειριοποιείται εντός του διαστήματος εκτέλεσής της.

**Λήμμα 8.7** *Έστω $S$ μία οποιαδήποτε SCAN, η $S$ σειριοποιείται εντός του διαστήματος εκτέλεσής της.*

**Απόδειξη.** Από τον ψευδοκώδικα του αλγορίθμου (γραμμές 6-7), η $S$ εκτελεί δύο φορές τη συνάρτηση $grab\_scan$. Συμβολίζουμε με $g$ την πρώτη $grab\_scan$ που εκτέλεσε η $S$, και με $g'$ τη δεύτερη $grab\_scan$ που εκτέλεσε η $S$. Συμβολίζουμε με $r_{seq}$ και $r'_{seq}$ τις εντολές read που η $g$ και η $g'$ εκτέλεσαν στη γραμμή 9 του ψευδοκώδικα. Συμβολίζουμε με $C_1$ και $C'_1$ τις CAS που εκτέλεσαν η $g$ και η $g'$ στη γραμμή 18 του ψευδοκώδικα. Από το Λήμμα 8.5 προκύπτει ότι το διάστημα εκτέλεσης που ξεκινά με την $r_{seq}$ και τελειώνει με την $C_1$, εμπεριέχει μία επιτυχημένη CAS τύπου 1 (έστω $C_{suc}$ η εν λόγω εντολή). Ομοίως το διάστημα εκτέλεσης που ξεκινά με την $r'_{seq}$ και τελειώνει με την $C'_1$ εμπεριέχει μία επιτυχημένη CAS τύπου 1 (έστω $C'_{suc}$ η εν λόγω εντολή). Έτσι από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $S$ σειριοποιείται στην τελευταία CAS τύπου 1 που επιτελέσθηκε πριν η $S$ εκτελέσει την εντολή της γραμμής 8 του ψευδοκώδικα. Άρα η $S$ θα σειριοποιηθεί είτε στο σημείο που επιτελέσθηκε η $C_{suc}$ είτε σε κάποιο σημείο που έπεται του σημείου όπου επιτελέσθηκε η $C_{suc}$. Άρα η $S$ σειριοποιείται έπειτα από την έναρξη του διαστήματος εκτέλεσής της. Επίσης από τον τρόπο ανάθεσης των σημείων σειριοποίησης, είναι προφανές ότι η $S$ σειριοποιείται πριν από το τέλος του διαστήματος εκτέλεσή της. Άρα συμπεραίνουμε ότι η $S$ σειριοποιείται εντός του διαστήματος εκτέλεσής της. ∎

Με το επόμενο λήμμα αποδεικνύεται ότι σε κάθε φάση μία ακριβώς εντολή CAS στον καταχωρητή $post[i]$, $1 \leq i \leq m$, εγγράφει τιμή ίση με $null$ στο πεδίο $value$ του καταχωρητή $post[i]$. Έπειτα από την εκτέλεση της προαναφερθείσας εντολής CAS, η τιμή του πεδίου $value$ του καταχωρητή $post[i]$ παραμένει $null$ έως ότου ξεκινήσει η επόμενη φάση. Επίσης στο επόμενο λήμμα αποδεικνύονται και μερικές ιδι-

ότητες που διέπουν τις τιμές του $seq.tm$, αλλά και του $post[i].tm$. Υπενθυμίζεται ότι με $C_1^0$ έχουμε συμβολίσει την αρχική καθολική κατάσταση του συστήματος.

**Λήμμα 8.8** *Έστω οποιοσδήποτε ακέραιος $1 \leq i \leq m$. Ισχύουν οι ακόλουθοι ισχυρισμοί:*

1. *ανάμεσα στην $C_1^{j-1}$ και στην $C_1^j$ ακριβώς μία επιτυχημένη εντολή CAS $C_{post}$ επιτελείται στον καταχωρητή $post[i]$ από κάποια συνάρτηση $clear\_registers$,*
2. *$post[i].tm = j - 1$ μεταξύ της $C_1^{j-1}$ και της $C_{post}$, και*
3. *$post[i].tm = j$ και $post[i].value = null$ μεταξύ της $C_{post}$ και της $C_1^j$.*

**Απόδειξη.** Η απόδειξη θα γίνει με επαγωγή στο $j, 1 \leq j \leq k_1$. Έστω ένας οποιοσδήποτε ακέραιος $j, 1 < j \leq k_1$. Υποθέτουμε ότι το λήμμα ισχύει για $j - 1$, θα αποδείξουμε την ορθότητα του λήμματος για $j$.

Από το Λήμμα 8.3 προκύπτει ότι μόνο οι CAS τύπου 1 αλλάζουν την τιμή του $seq.tm$. Στην περίπτωση όπου $j = 1$, αφού η $C_1^1$ είναι η πρώτη εντολή CAS που εκτελείται στην $a$ (το $seq.tm$ έχει αρχική τιμή ίση με 1 σε όλες τις χρονικές στιγμές μεταξύ της $C_1^0$ και της $C_1^1$). Έτσι οποιαδήποτε διεργασία ξεκινά την εκτέλεσή της πριν από την $C_1^1$, διαβάζει τιμή ίση με 1 στο $seq.tm$ (γραμμές 1, 9). Υπενθυμίζεται ότι για κάθε $i$, $1 \leq i \leq m$, αρχικά ισχύει $post[i].tm = 0$ και $post[i].value = null$.

Έστω $j > 1$. Το Πόρισμα 8.5 συνεπάγεται ότι αμέσως μετά την εκτέλεση της $C_1^{j-1}$, ισχύει $seq.tm = j$. Από το Λήμμα 8.3 προκύπτει ότι σε όλες τις χρονικές στιγμές μεταξύ της $C_1^{j-1}$ και της $C_1^j$θα ισχύει $seq.tm = j$. Επιπρόσθετα από το Λήμμα 8.4 συνεπάγεται ότι όλες οι λειτουργίες που έχουν ξεκινήσει την εκτέλεσή τους πριν την $C_1^{j-1}$ διαβάζουν μία τιμή μικρότερη ή ίση του $j$ στο $seq.tm$ (γραμμές 1, 9). Επίσης από την επαγωγική υπόθεση αμέσως μετά την $C_1^{j-1}$ ισχύει $post[i].tm = j$ και $post[i].value = null$.

Έτσι κάθε περίπτωση ($j = 0$ ή $j > 1$) ισχύει $seq.tm = j$ σε κάθε χρονική στιγμή μεταξύ της $C_1^{j-1}$ και της $C_1^j$, και όλες οι λειτουργίες που ξεκινούν την εκτέλεσή τους

πριν την $C_1^j$ διαβάζουν τιμή μικρότερη ή ίση του $j$ στο $seq.tm$. Ακόμη αμέσως μετά την $C_1^{j-1}$ ισχύει $post[i].tm = j - 1$ και $post[i].value = null$.

Έστω $p$ η διεργασία που επιτελεί την $C_1^j$. Αφού η $C_1^j$ είναι μία επιτυχημένη εντολή CAS και ισχύει $seq.tm = j$ κατά την εκτέλεση της $C_1^j$ (από τον ψευδοκώδικα του αλγορίθμου, γραμμή 9) η $p$ διαβάζει τιμή ίση με $j$ στο $seq.tm$.

Έστω ένας οποιοσδήποτε ακέραιος $i, 1 \leq i \leq m$. Αποδεικνύουμε τους δύο ακόλουθους ισχυρισμούς.

**Ισχυρισμός 8.1.** *Έστω $C$ η πρώτη επιτυχημένη εντολή CAS στον $post[i]$ που εκτελείται ανάμεσα στην $C_1^{j-1}$ και στην $C_1^j$ από κάποια συνάρτηση $clear\_registers$. Ισχύουν τα ακόλουθα:*

1. *Όλες οι επιτυχημένες εντολές CAS που επιτελούνται στον $post[i]$ μεταξύ της $C_1^{j-1}$ και της $C$ δεν μεταβάλλουν την τιμή του $post[i].tm$, και*
2. *καμία εντολή CAS στον $post[i]$ δεν επιτυγχάνει μεταξύ της $C$ και της $C_1^j$.*

**Απόδειξη.** Αρχικά θα αποδείξουμε το 1. Αφού η $C$ είναι η πρώτη επιτυχημένη εντολή CAS στον $post[i]$ που εκτελείται από μία $clear\_registers$ μεταξύ της $C_1^{j-1}$ και της $C_1^j$, οποιαδήποτε επιτυχημένη εντολή CAS εκτελείται μεταξύ της $C_1^{j-1}$ και της $C_1^j$, επιτελείται από μία UPDATE. Υπενθυμίζουμε ότι ισχύει $post[i].tm = j - 1$ αμέσως μετά την $C_1^{j-1}$. Έστω $U$ η πρώτη UPDATE που επιτελεί μία επιτυχημένη εντολή CAS (έστω $C_U$ η εν λόγω εντολή) στον $post[i]$ μεταξύ της $C_1^{j-1}$ και της $C_1^j$. Τότε θα ισχύει $post[i].tm = j - 1$ κατά την εκτέλεση της $C_U$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 4), η $C_U$ δε μεταβάλλει την τιμή του $post[i].tm$. Έτσι θα ισχύει $seq.tm = j - 1$ αμέσως μετά την $C_U$. Εφαρμόζοντας τα παραπάνω επαγωγικά, προκύπτει ότι καμία εντολή CAS στον καταχωρητή που εκτελείται μεταξύ της $C_1^{j-1}$ και της $C$, δεν αλλάζει την τιμή του $post[i].tm$, όπως και είναι απαραίτητο από το 1.

Στη συνέχεια θα αποδείξουμε τον ισχυρισμό 2. Από τον ισχυρισμό 1 που αποδείχθηκε πρωτύτερα προκύπτει ότι κατά την εκτέλεση της $C$ ισχύει $post[i].tm = j - 1$. Έστω ότι η $C$ εκτελείται από κάποια διεργασία $q$. Αφού η $C$ είναι μία επιτυχημένη εντολή

CAS, από τον ψευδοκώδικα του αλγορίθμου (γραμμές 21-22 και 25-26) προκύπτει ότι η $q$ πρέπει να έχει αναγνώσει τιμή ίση με $j$ στον καταχωρητή $seq$ (γραμμή 9), ενώ πρέπει να έχει γράψει τιμή ίση με $j$ στον καταχωρητή $post[i].tm$. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι υπάρχει μία τουλάχιστον επιτυχημένη εντολή CAS στον καταχωρητή $post[i]$ μεταξύ της $C$ και της $C_1^j$. Έστω $C'$ η πρώτη τέτοια εντολή. Κατά τη διάρκεια εκτέλεσης της $C'$ πρέπει να ισχύει $post[i].tm = j$. Αφού η $C'$ προηγείται της $C_1^j$, η λειτουργία που εκτέλεσε την $C'$ έχει ξεκινήσει την εκτέλεσή της πριν την $C_1^j$. Υπενθυμίζεται ότι όλες οι λειτουργίες που ξεκίνησαν την εκτέλεσή τους πριν την $C_1^j$, ανέγνωσαν στο $seq.tm$ τιμή μικρότερη ή ίση του $j$. Έτσι από τον ψευδοκώδικα του αλγορίθμου (γραμμές 3, 22 και 26), κατά την εκτέλεση της $C'$ ισχύει ότι $lpost.tm \leq j - 1$. Οπότε η $C'$ δε δύναται να είναι μία επιτυχημένη εντολή CAS, το οποίο και είναι άτοπο. Επαγωγικά εφαρμόζοντας ακριβώς τα ίδια επιχειρήματα, προκύπτει ότι καμία εντολή CAS στον καταχωρητή $post[i]$ δεν επιτελείται επιτυχώς μεταξύ της $C$ και της $C_1^j$, όπως και είναι απαραίτητο.

▲

**Ισχυρισμός 8.2.** *Για κάθε $i, 1 \leq i \leq m$, υπάρχει μία επιτυχημένη εντολή CAS στον καταχωρητή $post[i]$ που έχει εκτελεστεί από κάποια συνάρτηση $clear\_registers$.*

**Απόδειξη.** Έστω οποιοσδήποτε ακέραιος $i, 1 \leq i \leq m$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 13), η διεργασία $p$ καλεί τη συνάρτηση $clear\_registers$ προτού εκτελέσει την $C_1^j$. Έστωσαν $C_p^1$ και $C_p^2$, οι δύο εντολές CAS που εκτελέστηκαν από τη διεργασία $p$ στον καταχωρητή $post[i]$ (γραμμές 22 και 25). Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι δεν υπάρχει κάποια εντολή CAS στον καταχωρητή $post[i]$ που να έχει εκτελεστεί επιτυχώς από κάποια $clear\_registers$ μεταξύ της $C_1^{j-1}$ και της $C_1^j$. Από τον Ισχυρισμό 8.1 προκύπτει ότι κατά την εκτέλεση της $C_p^1$ ισχύει $post[i].tm = j - 1$. Υπενθυμίζουμε ότι η διεργασία $p$ διαβάζει τιμή ίση με $j$ στο $seq.tm$ (γραμμή 9), έτσι από τον ψευδοκώδικα του αλγορίθμου κατά την εκτέλεση της $C_p^1$ θα ισχύει $seq.tm = j - 1$. Αφού έχουμε υποθέσει ότι η $C_p^1$ είναι μία αποτυχημένη εντολή CAS και κατά την εκτέλεση της $C_p^1$ ισχύει $seq.tm = lpost.tm$, πρέπει να ισχύει $lpost.value \neq post[i].value$ κατά την εκτέ-

λεση της $C_p^1$. Υπενθυμίζεται ότι αμέσως μετά την εκτέλεση της $C_1^{j-1}$ ισχύει $post[i].value = null$. Έτσι πρέπει να υπάρχει τουλάχιστον μία UPDATE, της οποίας η εντολή CAS στον $post[i]$ να επιτυγχάνει μεταξύ της $C_1^{j-1}$ και της $C_p^1$. Έστω $U$ η πρώτη από τις προαναφερθείσες UPDATE και έστω $C_U$ η εντολή CAS που επιτελέστηκε από την $U$ στον $post[i]$. Ο Ισχυρισμός 8.1 συνεπάγεται ότι η $C_U$ εγγράφει τιμή ίση με $j-1$ στο $post[i].tm$. Έστω $v$ η τιμή την οποία η $C_U$ εγγράφει στο $post[i].value$.

Επιχειρηματολογήσαμε ότι όλες οι εντολές CAS στον $post[i]$ που εκτελούνται μεταξύ της $C_U$ και της $C_1^j$ αποτυγχάνουν. Υπενθυμίζουμε ότι οι εν λόγω εντολές CAS εκτελούνται από UPDATE. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 3), για κάθε μία από τις προαναφερθείσες UPDATE ισχύει $lpost.value = null$. Αξίζει να σημειωθεί ότι έπειτα από την εκτέλεση της $C_U$ ισχύει $post[i].value = v \neq null$, και έτσι η εντολή CAS οποιασδήποτε από τις προαναφερθείσες UPDATE αποτυγχάνει. Έτσι προκύπτει ότι σε οποιαδήποτε χρονική στιγμή μεταξύ της $C_U$ και της $C_1^j$ ισχύει $post[i].tm = j-1$ και $post[i].value = v$. Οπότε όταν η διεργασία $p$ επιτελέσει τη δεύτερη εντολή read της στον $post[i]$ (γραμμή 23), διαβάζει τιμή ίση με $v$ στο $post[i].value$, έτσι κατά την εκτέλεση της $C_p^2$ από τη διεργασία $p$ θα ισχύει $lpost.value = v$. Υπενθυμίζεται ότι η διεργασία $p$ διαβάζει τιμή ίση με $j$ στο $seq.tm$. Έτσι από τον ψευδοκώδικα του αλγορίθμου (γραμμή 24), κατά την εκτέλεση της $C_p^2$ θα ισχύει $lpost.tm = j-1$. Έτσι η $C_p^2$ θα είναι επιτυχής, το οποίο και είναι άτοπο. ▲

Στο σημείο αυτό έγινε φανερό γιατί είναι απαραίτητο να εκτελούνται δύο CAS σε όλους του καταχωρητές του $post$ από την $clear\_registers$.

Ο Ισχυρισμός 8.2 συνεπάγεται ότι υπάρχει μία επιτυχημένη εντολή CAS στον καταχωρητή $post[i]$, η οποία έχει εκτελεστεί από κάποια $clear\_registers$ μεταξύ της $C_1^{j-1}$ και της $C_1^j$. Από τον Ισχυρισμό 8.1 προκύπτει ότι υπάρχει μία ακριβώς τέτοια εντολή CAS (έστω $C_{post}$ η εν λόγω εντολή), όπως και είναι απαραίτητο από το 1. Υπενθυμίζεται ότι αμέσως μετά την $C_1^{j-1}$ ισχύει $post[i].tm = j-1$. Ο Ισχυρισμός 8.1 συνεπάγεται ότι σε όλες τις χρονικές στιγμές μεταξύ της $C_1^{j-1}$ και της $C_{post}$ ισχύει

$post[i].tm = j - 1$, όπως και απαιτείται από το 2. Ακόμη προκύπτει πως κατά την εκτέλεση της $C_{post}$, ισχύει $post[i].tm = j - 1$. Έτσι η $C_{post}$ εγγράφει τιμή ίση με $j$ στο $post[i].tm$ και τιμή ίση με $null$ στο $post[i].value$. Ο Ισχυρισμός 8.1 συνεπάγεται πως οι προαναφερθείσες τιμές δεν αλλάζουν έπειτα από την$C_{post}$ έως την $C_1^j$ όπως και απαιτείται από το 3. ■

Στο επόμενο λήμμα θα αποδειχθεί ότι οι συναρτήσεις $grab\_scan$ που έχουν ξεκινήσει την εκτέλεσή τους σε προηγούμενες φάσεις, δε δύναται να μεταβάλουν τα δεδομένα κανενός Καταχωρητή CAS στην τρέχουσα ή σε επόμενη φάση.

**Λήμμα 8.9** *Έστω $C_1^j, 1 \leq j \leq k_1$, μία οποιαδήποτε CAS τύπου 1 και έστω $g$ η συνάρτηση $grab\_scan$ που διαβάζει τελευταία τον καταχωρητή $seq$ (γραμμή 9) πριν την $C_1^j$. Έπειτα από την εκτέλεση της $C_1^j$, καμία εντολή CAS (στον καταχωρητή $seq$ ή σε οποιονδήποτε καταχωρητή του πίνακα $post$) της συνάρτησης $g$ (καθώς και της συνάρτησης $clear\_registers$ που καλείται από την $g$) δεν είναι επιτυχημένη.*

**Απόδειξη.** Από το Πόρισμα 8.5 προκύπτει ότι η $C_1^j$ εγγράφει τιμή ίση με $j + 1$ στο $seq.tm$. Έστω $q$ η διεργασία η οποία εκτελεί τη συνάρτηση $g$. Αφού η εκτέλεση της $g$ ξεκινά πριν την $C_1^j$, η $q$ διαβάζει τιμή μικρότερη ή ίση του $j$ στο $seq.tm$ (γραμμή 9). Συμβολίζουμε με $C$ οποιαδήποτε από τις δύο εντολές CAS που η $q$ εκτελεί στον $post[i]$ (κατά τη διάρκεια εκτέλεσης της $clear\_registers$). Αρχικά θα αποδειχθεί ότι μετά την εκτέλεση της $C_1^j$, καμία εντολή CAS στον $seq$ που εκτελείται από την $g$ δεν είναι επιτυχής. Από το Πόρισμα 5.8 συνεπάγεται ότι όλες οι τιμές που εγγράφηκαν στο $seq.tm$ έπειτα από την $C_1^j$ είναι μεγαλύτερες του $j + 1$. Υπενθυμίζεται ότι η $g$ διαβάζει στον $seq$ τιμή μικρότερη ή ίση με $j$. Έτσι για την $g$ θα ισχύει $curr\_seq.tm \leq j$, ενώ $seq.tm > j$ σε κάθε χρονική στιγμή έπειτα από την $C_1^j$. Έτσι οι εντολές CAS που εκτελεί $g$ έπειτα από την $C_1^j$, δεν είναι επιτυχημένες.

Έστω οποιοσδήποτε ακέραιος $i, 1 \leq i \leq m$. Στη συνέχεια θα αποδείξουμε ότι έπειτα από την εκτέλεση της $C_1^j$, καμία εντολή CAS της $g$ στον $post[i]$ δεν είναι επιτυχημένη. Έστω $C$ μία από τις προαναφερθείσες εντολές CAS. Από το Λήμμα 8.8 προκύπτει ότι $post[i].tm \geq j$ σε όλα τα χρονικά σημεία έπειτα από την εκτέλεση της $C_1^j$. Υπεν-

θυμίζουμε ότι η $g$ διαβάζει τιμή μικρότερη της $j+1$ στο $seq.tm$. Έτσι κατά την εκτέλεση της $C$ θα ισχύει $lpost.tm < j$. Άρα η $C$ δε θα είναι μία επιτυχημένη εντολή CAS. ∎

Με το επόμενο λήμμα θα αποδειχθεί ότι η αρχικοποίηση των $post$ καταχωρητών σε κάθε φάση, ξεκινά έπειτα από την εκτέλεση της επιτυχημένης εντολής CAS τύπου 0 κάθε φάσης. Έστω οποιοδήποτε $j, 1 \leq j \leq k_1$ και έστω οποιοδήποτε $i, 1 \leq i \leq m$. Συμβολίζουμε με $C_0$ την επιτυχημένη CAS τύπου 0 που εκτελείται μεταξύ της $C_1^{j-1}$ και της $C_1^j$. Από το λήμμα 8.7 προκύπτει ότι υπάρχει μία επιτυχημένη εντολή CAS στον $post[i]$ κάποιας $clear\_registers$ μεταξύ της $C_1^{j-1}$ και της $C_1^j$. Συμβολίζουμε με $C_{post}$ την προαναφερθείσα εντολή CAS.

**Λήμμα 8.10** *Η $C_{post}$ εκτελείται έπειτα από την $C_0$.*

**Απόδειξη.** Ας υποθέσουμε ότι $j = 1$. Έστω $C_0'$ η πρώτη εντολή CAS που εκτελείται στον $seq$ (η οποία είναι προφανώς τύπου 0, βλ. γραμμή 12) που εκτελείται από οποιαδήποτε διεργασία. Από τον ψευδοκώδικα του αλγορίθμου (βλ. γραμμές 9-12) είναι προφανές ότι η εν λόγω εντολή θα είναι μια επιτυχημένη CAS. Από τον ψευδοκώδικα του αλγορίθμου, μία συνάρτηση $grab\_scan$ αρχικά εκτελεί μία CAS τύπου 0 και έπειτα καλεί τη συνάρτηση $clear\_registers$. Έτσι η συνάρτηση $clear\_registers$ ξεκινά την εκτέλεσή της έπειτα από την $C_0'$. Οπότε η $C_{post}$ εντολή έπεται της $C_0'$. Αφού η $C_0'$ είναι η πρώτη επιτυχημένη CAS εντολή, θα ισχύει $C_0 = C_0'$ ή $C_0'$ θα έπεται της $C_0$. Όμως από το Λήμμα 8.2 προκύπτει ότι η μόνη επιτυχημένη CAS τύπου 0 μεταξύ της $C_1^{j-1}$ και της $C_1^j$ είναι η $C_0'$. Έτσι συμπεραίνουμε ότι $C_0 = C_0'$, και άρα η $C_{post}$ εκτελείται έπειτα από την $C_0$.

Έστω τώρα ότι $j > 1$. Από το Λήμμα 8.8 προκύπτει ότι οι εντολές CAS (σε οποιονδήποτε καταχωρητή) μιας συνάρτησης $grab\_scan$ που ξεκίνησε την εκτέλεσή της πριν την $C_1^{j-1}$, δεν είναι επιτυχημένες. Έτσι η $C_0$ και η $C_{post}$ εκτελούνται από $grab\_scan$ συναρτήσεις που ξεκίνησαν την εκτέλεσή τους έπειτα από την $C_1^{j-1}$. Έστω $\mathbb{C}$ σύνολο των $grab\_scan$ που ξεκινούν την εκτέλεσή τους έπειτα από την $C_1^{j-1}$ και έστω $g$ η συνάρτηση $grab\_scan$ που επιτελεί πρώτη την εντολή CAS της τύπου

0 (έστω $C'_0$ η εν λόγω εντολή). Είναι προφανές ότι η $C'_0$ εκτελείται πριν από την $C_0$ ή ισχύει $C_0 = C'_0$. Έτσι η εντολή $C'_0$ προηγείται της εντολής $C_1^j$. Από το Λήμμα 8.3 προκύπτει ότι τα περιεχόμενα του καταχωρητή $seq$ δεν μεταβάλλονται μεταξύ της $C_1^{j-1}$ και της πρώτης επιτυχημένης εντολής CAS τύπου 0 που έπεται της $C_1^{j-1}$. Έτσι κατά την εκτέλεση της $C'_0$ θα ισχύει $curr\_seq = seq$ και η $C'_0$ θα είναι επιτυχημένη. Το Λήμμα 8.2 συνεπάγεται ότι η μόνη επιτυχημένη CAS μεταξύ της $C_1^{j-1}$ και της $C_1^j$ είναι η $C_0$. Έτσι πρέπει να ισχύει $C'_0 = C_0$, οπότε η $C_0$ είναι η πρώτη CAS τύπου 0 που εκτελείται από οποιαδήποτε συνάρτηση $grab\_scan$ του συνόλου $\mathbb{C}$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι όλες οι συναρτήσεις $clear\_registers$ που καλούνται από $grab\_scan$ συναρτήσεις του συνόλου $\mathbb{C}$, ξεκινούν την εκτέλεσή τους έπειτα από τη $C_0$. Έτσι συμπεραίνουμε ότι η $C_{post}$ εκτελείται μετά την $C_0$. ∎

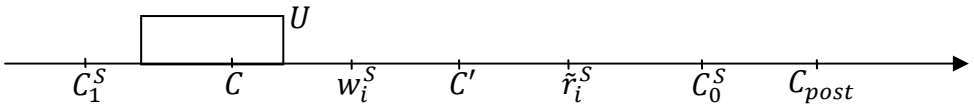
Αποδεικνύουμε στη συνέχεια ότι οι UPDATE που επιτελούν την εντολή write τους στον καταχωρητή $pre[i]$ μεταξύ της $C_1^S$ και της $w_i^S$, έχουν ξεκινήσει την εκτέλεσή τους πριν την $C_1^S$. Ο ισχυρισμός αυτός είναι απαραίτητος για να αποδειχθεί ότι κάθε UPDATE σειριοποιείται εντός του διαστήματος εκτέλεσής της.

**Λήμμα 8.11** *Για κάθε $i \in \{1, \ldots, m\}$, τέτοιο ώστε η $w_i^S$ να έπεται της $C_1^S$, ισχύει ότι οποιαδήποτε UPDATE στη συνιστώσα $A_i$ που επιτελεί την write της μεταξύ της $C_1^S$ και της $w_i^S$ (με την $U_i^S$ να συμπεριλαμβάνεται), έχει ξεκινήσει την εκτέλεσή της πριν την $C_1^S$.*

**Απόδειξη.** Ισχύει $C_1^S = C_1^j$ για κάποιο $j, 1 \leq j \leq k_1$. Έστω $C_{post}$ η πρώτη επιτυχημένη εντολή CAS στον $post[i]$, η οποία έχει εκτελεστεί από μία συνάρτηση $clear\_registers$ έπειτα από την $C_1^S$. Το Λήμμα 8.9 συνεπάγεται ότι η $C_{post}$ εκτελείται έπειτα από την $C_0^S$. Από το Πόρισμα 8.5, η $C_1^S$ εγγράφει στο $seq.tm$ τιμή ίση με $j + 1$. Το Λήμμα 8.7 συνεπάγεται ότι μεταξύ της $C_1^S$ και της $C_{post}$ ισχύει ότι $post[i].tm = j$.

Ας υποθέσουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι υπάρχει μία τουλάχιστον UPDATE στη συνιστώσα $A_i$ που να έχει ξεκινήσει την εκτέλεσή της έπειτα από την $C_1^S$ και να έχει επιτελέσει την write της στον $pre[i]$ πριν την $w_i^S$. Συμβολίζουμε με $Y$ το σύνολο των UPDATE που ξεκίνησαν την εκτέλεσή τους έπειτα από την $C_1^S$

και επιτέλεσαν την write τους στον $pre[i]$ πριν την $w_i^S$. Έστω $U \in Y$ η UPDATE που επιτελεί πρώτη την εντολή CAS της (έστω $C$) στον $post[i]$. Από το Λήμμα 8.1 προκύπτει ότι η $C_0^S$ έπεται της $w_i^S$, έτσι η $C$ προηγείται της $C_0^S$ (βλ. Σχήμα 8.1).




Σχήμα 8.1 Η $U$ έχει ξεκινήσει την εκτέλεσή της έπειτα από την $C_1^S$.

Υπενθυμίζουμε ότι η $C_1^j = C_1^S$ εγγράφει στο $seq.tm$ τιμή ίση με $j+1$. Από το Λήμμα 8.7 συνεπάγεται ότι μεταξύ της $C_1^j$ και της $C_1^{j+1}$ (ή το τέλος της εκτέλεσης αν $j = k_1$) ισχύει $seq.tm = j+1$. Αφού η $U$ ξεκινά την εκτέλεσή της έπειτα από την $C_1^S$ και επιτελεί την $C$ πριν την $C_0^S$, προκύπτει ότι η $U$ διαβάζει τιμή ίση με $j+1$ στο $seq.tm$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 4), κατά την εκτέλεση της $C$, ισχύει $lpost.tm = j$. Έτσι κατά την εκτέλεση της $C$ θα ισχύει $post[i].tm = lpost.tm$. Από το Λήμμα 8.7 ισχύει $post[i].value = null$ αμέσως μετά την επιτέλεση της $C_1^S$. Από το Λήμμα 8.7 προκύπτει ότι η μοναδική εντολή CAS που εκτελέστηκε στον $post[i]$ από μία $clear\_registers$ μεταξύ της $C_1^S$ και της $C_1^{j+1}$ (ή το τέλος της εκτέλεσης αν $j = k_1$) είναι η $C_{post}$ που έπεται της $w_i^S$ και άρα έπεται και της $C$. Αφού η $C$ είναι η πρώτη εντολή CAS στον $post[i]$ εκτελούμενη από μία UPDATE που ξεκίνησε την εκτέλεσή της έπειτα από την $C_1^S$, προκύπτει ότι κατά την εκτέλεση της $C$ ισχύει $post[i].value = null$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 6), κατά την εκτέλεση της $C$ ισχύει $lpost.value = null$. Έτσι προκύπτει ότι η $C$ είναι μία επιτυχημένη εντολή CAS στον $post[i]$.

Από το Λήμμα 8.1 ισχύει ότι η $w_i^S$ προηγείται της $\tilde{r}_i^S$. Έτσι η $C$ προηγείται της $\tilde{r}_i^S$. Το Λήμμα 8.9 συνεπάγεται ότι καμία $clear\_registers$ δεν εγγράφει τιμή ίση με $null$ στον $post[i]$ μεταξύ της $C_1^S$ και της $C_0^S$. Αφού η $\tilde{r}_i^S$ προηγείται της $C_0^S$, η $\tilde{r}_i^S$ διαβάζει μία τιμή που είναι διαφορετική του $null$ στον $post[i]$. Έτσι η $V_i^S$ είναι καλά ορισμένη. Επιπρόσθετα ισχύει $V_i^S \notin Y$, αφού η $V_i^S$ διαβάζει την τιμή που η $w_i^S$ έγγραψε στον $pre[i]$, και έτσι επιτελεί την CAS της $C'$ στον $pre[i]$ έπειτα από την $w_i^S$. Έτσι θα ισχύει $V_i^S \neq U$.

Εξ ορισμού η $V_i^S$ επιτελεί την $C'$ στον $post[i]$ πριν την εκτέλεση της $\tilde{r}_i^S$ (αφού η $\tilde{r}_i^S$ διαβάζει την τιμή που εγγράφηκε στον $pre[i]$ από την $C'$). Έτσι η $C'$ προηγείται της $C_0^S$. Επιπρόσθετα η $C'$ έπεται της $w_i^S$ και έτσι η $C'$ έπεται της $C$. Προκύπτει ότι κατά την εκτέλεση της $C'$ ισχύει $post[i].value \neq null$. Από τον ψευδοκώδικα του αλγορίθμου (γραμμή 6) κατά την εκτέλεση της $C'$ ισχύει $lpost.value = null$. Έτσι η $C'$ δεν είναι μία επιτυχημένη εντολή CAS, το οποίο αντιτίθεται στο ότι η CAS εντολή της $V_i^S$ (εξ ορισμού) στον $post[i]$ είναι επιτυχημένη. ∎

**Λήμμα 8.12** *Έστω U μία οποιαδήποτε UPDATE. Η U σειριοποιείται εντός του διαστήματος εκτέλεσής της.*

**Απόδειξη.** Έστω $U$ μία UPDATE στη συνιστώσα $A_i$ που δεν σειριοποιείται στην write της στον $pre[i]$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S$ τέτοια ώστε η $w_i^S$ της $U_i^S$ να εκτελείται έπειτα από την $C_1^S$, η write της $U$ στον $pre[i]$ να εκτελείται μεταξύ της $C_1^S$ και της $w_i^S$, και η $U$ να σειριοποιείται ακριβώς πριν την $C_1^S$. Προφανώς η εκτέλεση της $U$ τελειώνει μετά την $C_1^S$. Από το Λήμμα 8.10 προκύπτει ότι η $U$ ξεκινά την εκτέλεσή της πριν την $C_1^S$. Έτσι η $U$ σειριοποιείται εντός του διαστήματος εκτέλεσής της. ∎

Σε αυτό το σημείο θα αποδείξουμε ότι οι SCAN επιστρέφουν συνεπή διανύσματα τιμών. Αρχικά αποδεικνύονται μερικά τεχνικά λήμματα. Έστω ότι δύο SCAN επιστρέφουν διανύσματα τιμών που γράφτηκαν από δύο διαφορετικές $grub\_scan$. Σε αυτή την περίπτωση η CAS τύπου 0 που εγγράφει το πρώτο από τα προαναφερθέντα διανύσματα εκτελείται σε μία διαφορετική φάση από την CAS τύπου 0 που εγγράφει το δεύτερο διάνυσμα τιμών (αφού από το Λήμμα 8.2 δεν δύναται να έχουμε δύο επιτυχημένες CAS τύπου 0 στην ίδια φάση).

**Λήμμα 8.13** *Έστωσαν $S_1$ και $S_2$ δύο SCAN τέτοιες ώστε $g_c^{S_1} \neq g_c^{S_2}$. Τότε τα διαστήματα $a(C_1^{S_1}, C_0^{S_1})$, $a(C_1^{S_2}, C_0^{S_2})$ δεν επικαλύπτονται.*

**Απόδειξη.** Αφού $g_c^{S_1} \neq g_c^{S_2}$, θα ισχύει $C_0^{S_1} \neq C_0^{S_2}$. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι η $C_0^{S_1}$ προηγείται της $C_0^{S_2}$. Από το Λήμμα 8.5 προκύπτει ότι υπάρχει μία τουλάχιστον επιτυχημένη CAS τύπου 1 μεταξύ της $C_0^{S_1}$ και της $C_0^{S_2}$. Έτσι θα ι-

σχύει $C_1^{S_1} \neq C_1^{S_2}$ και η $C_1^{S_2}$ θα έπεται της $C_0^{S_1}$. Αφού η $C_1^{S_1}$ προηγείται της $C_0^{S_1}$, προκύπτει ότι τα διαστήματα $a(C_1^{S_1}, C_0^{S_1})$, $a(C_1^{S_2}, C_0^{S_2})$ δεν επικαλύπτονται. ∎

Στο επόμενο λήμμα αποδεικνύεται ότι για κάθε συνιστώσα $A_i, 1 \leq i \leq m$, τα σημεία σειριοποίησης όλων των UPDATE στην $A_i$ σέβονται τη διάταξη που καθορίζεται από την εκτέλεση των write τους στον $pre[i]$.

**Λήμμα 8.14** *Έστω $U_1$, $U_2$ δύο UPDATE σε κάποια συνιστώσα $A_i, 1 \leq i \leq m$. Συμβολίζουμε με $w_1$ την write της $U_1$ στον $pre[i]$, και με $w_2$ την write της $U_2$ στον $pre[i]$. Αν η $w_1$ προηγείται της $w_2$, τότε το σημείο σειριοποίησης της $U_1$ προηγείται του σημείου σειριοποίησης της $U_2$.*

**Απόδειξη.** Υποθέτουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι ο ισχυρισμός δεν ισχύει. Αν οι $U_1$ και $U_2$ σειριοποιούνται στις write τους στον καταχωρητή $pre[i]$, τότε προφανώς το λήμμα ισχύει. Έτσι υποθέτουμε ότι τουλάχιστον μία από τις $U_1, U_2$ δε σειριοποιείται στην write της στον καταχωρητή $pre[i]$. Διακρίνουμε τις ακόλουθες περιπτώσεις.

1) Η $U_2$ σειριοποιείται στην $w_2$. Το Λήμμα 8.12 συνεπάγεται ότι η $U_1$ σειριοποιείται εντός του διαστήματος εκτέλεσής της, έτσι η $U_1$ θα σειριοποιηθεί το πολύ στην $w_1$. Έτσι η $U_1$ σειριοποιείται πριν την $U_2$, καθώς έχουμε υποθέσει ότι η $U_2$ σειριοποιείται πριν την $U_1$.
2) Η $U_1$ σειριοποιείται στην $w_1$. Αφού έχουμε υποθέσει ότι η $U_2$ σειριοποιείται πριν την $U_1$, η $U_2$ δε δύναται να σειριοποιηθεί στην $w_2$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S$ τέτοια ώστε η $w_2$ να έχει επιτελεσθεί μεταξύ της $C_1^S$ και της $w_i^S$. Αφού η $w_1$ προηγείται της $w_2$, η $w_1$ έχει επιτελεσθεί πριν την $w_i^S$. Στην περίπτωση όπου η $w_1$ έπεται της $C_1^S$, η $U_1$ και η $U_2$ σειριοποιούνται ακριβώς πριν την $C_1^S$ με τη σειρά με την οποία επιτέλεσαν τις write τους στον $pre[i]$. Συμπεραίνουμε ότι η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο. Έτσι η $w_1$ πρέπει να προηγείται της $C_1^S$. Από το Λήμμα 8.12 προκύπτει ότι η $U_1$ σειριοποιείται το αργότερο στην $w_1$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $U_2$ σειριοποιείται ακριβώς πριν την $C_1^S$. Έτσι η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο.

3) Καμία από τις $U_1, U_2$ δε σειριοποιείται στην write της στον $pre[i]$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχουν δύο SCAN $S_1$ και $S_2$ τέτοιες ώστε η $w_1$ να έχει επιτελεσθεί μεταξύ της $C_1^{S_1}$ και της $w_i^{S_1}$, ενώ η $w_2$ να έχει επιτελεσθεί μεταξύ της $C_1^{S_2}$ και της $w_i^{S_2}$. Επιπρόσθετα η $U_1$ σειριοποιείται ακριβώς πριν την $C_1^{S_1}$ και η $U_2$ σειριοποιείται ακριβώς πριν την $C_1^{S_2}$. Από το Λήμμα 8.1 προκύπτει ότι η $w_i^{S_1}$ προηγείται της $C_0^{S_1}$, ενώ η $w_i^{S_2}$ προηγείται της $C_0^{S_2}$.

   Αν ισχύει $g_c^{S_1} = g_c^{S_2}$, τότε $C_0^{S_1} = C_0^{S_2}$ και έτσι θα ισχύει $C_1^{S_1} = C_1^{S_2}$. Σε αυτή την περίπτωση και η $U_1$ και η $U_2$ σειριοποιούνται ακριβώς πριν την $C_1^{S_1} = C_1^{S_2}$ με τη σειρά με την οποία επιτέλεσαν την write τους στον καταχωρητή $pre[i]$. Έτσι η $U_1$ σειριοποιείται πριν την $U_2$, το οποίο και είναι άτοπο.

   Αν ισχύει $g_c^{S_1} \neq g_c^{S_2}$, το Λήμμα 8.13 συνεπάγεται ότι τα διαστήματα εκτέλεσης $a(C_1^{S_1}, C_0^{S_1})$, $a(C_1^{S_2}, C_0^{S_2})$ δεν τέμνονται. Αν η $C_1^{S_1}$ προηγείται της $C_1^{S_2}$, το σημείο σειριοποίησης της $U_1$ που τίθεται ακριβώς πριν την $C_1^{S_1}$ προηγείται του σημείου σειριοποίησης της $U_2$ που τίθεται ακριβώς πριν την $C_1^{S_2}$, το οποίο και είναι άτοπο.

   Ας υποθέσουμε επομένως ότι η $C_1^{S_1}$ έπεται της $C_1^{S_2}$. Από το Λήμμα 8.1 προκύπτει ότι η $w_i^{S_1}$ προηγείται της $C_0^{S_1}$ και η $w_i^{S_2}$ προηγείται της $C_0^{S_2}$. Έτσι η $w_1$ που εκτελείται πριν την $C_1^S$ και η $w_i^{S_1}$ έπονται της $w_2$ που εκτελείται μεταξύ της $C_1^{S_2}$ και της $w_i^{S_2}$, το οποίο και είναι άτοπο.

Σε κάθε περίπτωση οδηγηθήκαμε σε άτοπο. Έτσι δύναται να συμπεράνουμε ότι το σημείο σειριοποίησης της $U_1$ προηγείται του σημείου σειριοποίησης της $U_2$. ■

**Λήμμα 8.15** *Για κάθε $i \in \{1, \ldots, m\}$, η $r_i^S$ έπεται της $C_1^S$.*

**Απόδειξη.** Η $r_i^S$ είναι η εντολή read της $g_0^S$ στον $pre[i]$, όπου η $g_0^S$ είναι η $grub\_scan$ που εκτελεί την $C_0^S$. Εξ ορισμού η $C_1^S$ είναι η τελευταία επιτυχημένη CAS τύπου 1 που προηγείται της $C_0^S$. Έστω ότι η $C_1^S$ εγγράφει στο $seq.tm$ τιμή ίση με $j$. Από το Λήμμα 8.3 συνεπάγεται ότι και η $C_0^S$ εγγράφει στο $seq.tm$ τιμή ίση με $j$. Έτσι από τον ψευδοκώδικα του αλγορίθμου, η $g_0^S$ διαβάζει στο $seq.tm$ τιμή ίση με $j$

(γραμμή 9). Το Πόρισμα 8.5 συνεπάγεται ότι σε όλες τις χρονικές στιγμές πριν την $C_1^S$, η τιμή του $seq.tm$ είναι μικρότερη ή ίση με $j-1$. Συμπεραίνουμε ότι η $g_0^S$ επιτελεί την εντολή read της στον $seq$ (έστω $r_{seq}$) έπειτα από την $C_1^S$. Αφού η $r_i^S$ έπεται της $r_{seq}$, προκύπτει ότι η $r_i^S$ έπεται της $C_1^S$, όπως και είναι απαραίτητο. ■

**Λήμμα 8.16** *Έστω $S$ μία SCAN τέτοια ώστε η $V_i^S$ να είναι καλά ορισμένη, και έστω $r_{pre}$ η εντολή read της $V_i^S$ στον καταχωρητή $pre[i]$. Τότε η $r_{pre}$ εκτελείται έπειτα από την $C_1^S$.*

**Απόδειξη.** Ας υποθέσουμε δια της μεθόδου της εις άτοπο απαγωγής, ότι η $r_{pre}$ εκτελείται πριν την $C_1^S$.

Έστω ότι $C_1^S = C_1^j$, για κάποιο $1 \leq j \leq k_1$. Τότε από το Πόρισμα 8.5 προκύπτει ότι η $C_1^S$ εγγράφει τιμή ίση με $j+1$ στο $seq.tm$. Από τον τρόπο ορισμού της $V_i^S$, η εντολή CAS $C$ της $V_i^S$ στον καταχωρητή $post[i]$ είναι μία επιτυχημένη εντολή CAS και η τιμή που εγγράφεται από τη $C$ στο $post[i].value$ διαβάζεται από την $\tilde{r}_i^S$. Από το Λήμμα 8.1 προκύπτει ότι η $r_i^S$ έπεται της $C_1^S$. Αφού η $\tilde{r}_i^S$ έπεται της $r_i^S$, η $\tilde{r}_i^S$ έπεται της $C_1^S$. Από το Λήμμα 8.8 προκύπτει ότι αμέσως μετά την εκτέλεση της $C_1^S$ ισχύει $post[i].tm = null$. Από τον τρόπο ορισμού της $V_i^S$, η τιμή που η $C$ εγγράφει στο $post[i].value$ διαβάζεται από την $\tilde{r}_i^S$. Αφού η $\tilde{r}_i^S$ έπεται της $C_1^S$, η $C$ πρέπει να έχει εκτελεστεί έπειτα από την $C_1^S$. Από το Λήμμα 8.8 προκύπτει ότι σε όλες τις χρονικές στιγμές έπειτα από την εκτέλεση της $C_1^S$ ισχύει $post[i].tm \geq j$. Έτσι κατά την εκτέλεση της $C$ ισχύει $post[i].tm \geq j$.

Αφού η εντολή read $r$ της $V_i^S$ στον καταχωρητή $seq$ προηγείται της $r_{pre}$ και αφού έχουμε υποθέσει ότι η $r_{pre}$ προηγείται της $C_1^S$, η $r$ θα προηγείται της $C_1^S$. Το Πόρισμα 8.5 συνεπάγεται ότι σε όλα τα χρονικά σημεία πριν την $C_1^S$ ισχύει $seq.tm \leq j$. Έτσι η $r$ διαβάζει στο $seq.tm$ τιμή μικρότερη ή ίση του $j$. Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι κατά την εκτέλεση της $C$ ισχύει $lpost.tm < j$. Αφού κατά την εκτέλεση της $C$ ισχύει $post[i].tm \geq j$ και $lpost.tm < j$, προκύπτει ότι η $C$ δεν είναι μία επιτυχημένη εντολή CAS, το οποίο και είναι άτοπο. ■

Στο επόμενο λήμμα θα αποδείξουμε τη συνέπεια του αλγορίθμου C-Snap.

**Λήμμα 8.17** *Έστω $a$ μία οποιαδήποτε εκτέλεση του αλγορίθμου C-Snap. Οποιαδήποτε SCAN $S$ στην $a$ επιστρέφει ένα συνεπές διάνυσμα τιμών.*

**Απόδειξη.** Υποθέτουμε ότι η $S$ επιστρέφει το διάνυσμα τιμών $\vec{v} = \langle v_1, \ldots, v_m \rangle$. Από τον τρόπο ορισμού της $C_0^S$ και από το Λήμμα 8.2, προκύπτει ότι το $\vec{v}$ εγγράφηκε στον καταχωρητή $seq$ από την $C_0^S$. Επομένως το διάνυσμα τιμών που υπολογίζεται από την $g_0^S$ είναι το $\vec{v}$. Έτσι για κάθε συνιστώσα $A_i, 1 \leq i \leq m$, η $g_0^S$ είτε διαβάζει $null$ στο $post[i].value$ (γραμμή 28) και $v_i$ στον $pre[i]$, $1 \leq i \leq m$, είτε διαβάζει $v_i$ στο $post[i].value$. Σε κάθε περίπτωση από τον τρόπο ορισμού της $U_i^S$, η $U_i^S$ πρέπει να έχει εγγράψει τιμή ίση με $v_i$ στον καταχωρητή $pre[i]$. Έτσι η $U_i^S$ χρησιμοποιεί ως παράμετρο την τιμή $v_i$. Στην περίπτωση όπου η $w_i^S$ προηγείται της $C_1^S$, το Λήμμα 8.12 συνεπάγεται ότι η $U_i^S$ σειριοποιείται πριν την $C_1^S$ (όπου σειριοποιείται η $S$). Στην περίπτωση όπου η $w_i^S$ έπεται της $C_1^S$, από τον τρόπο ανάθεσης σημείων σειριοποίησης, το σημείο σειριοποίησης της $U_i^S$ προηγείται του σημείου σειριοποίησης της $S$. Έτσι σε κάθε περίπτωση, η $U_i^S$ αποθηκεύει την τιμή $v_i$ στη συνιστώσα $A_i$, ενώ το σημείο σειριοποίησής της προηγείται του σημείου σειριοποίησης της $S$. Θα αποδειχθεί ότι δεν υπάρχει UPDATE στη συνιστώσα $A_i$ που να σειριοποιείται ανάμεσα στην $U_i^S$ και την $S$, και άρα η $S$ επιστρέφει συνεπή τιμή για τη συνιστώσα $A_i$.

Ας υποθέσουμε δια της μεθόδου της εις άτοπο απαγωγής ότι υπάρχει ένας ακέραιος $i \in \{1, \ldots, m\}$, τέτοιος ώστε η τελευταία UPDATE στην συνιστώσα $A_i$ που σειριοποιείται πριν την $S$ να μην είναι η λειτουργία $U_i^S$. Συμβολίζουμε με $U$ την προαναφερθείσα UPDATE, έστω $v \neq v_i$ η τιμή που η $U$ εγγράφει στον καταχωρητή $pre[i]$ και έστω $w$ η write της $U$ στον καταχωρητή $pre[i]$. Διακρίνουμε τις ακόλουθες περιπτώσεις.

1) Έστω ότι η $g_0^S$ διαβάζει τιμή ίση με $null$ στον καταχωρητή $post[i]$ και τιμή ίση με $v_i$ στον καταχωρητή $pre[i]$ (γραμμή 27). Στην περίπτωση όπου η $w$ προηγείται της $w_i^S$, το Λήμμα 8.15 συνεπάγεται ότι η $U$ σειριοποιείται πριν την $U_i^S$, το οποίο και είναι άτοπο. Έτσι υποθέτουμε ότι η $w$ έπεται της $w_i^S$. Από τον τρόπο ορισμού της $w_i^S$, η $r_i^S$ διαβάζει την τιμή που η $w_i^S$ έγγραψε στον $pre[i]$. Άρα η $w$ πρέπει να έπεται της $r_i^S$. Από το Λήμμα 8.15 προκύπτει ότι η $r_i^S$ έπεται της $C_1^S$.

Έτσι η $w$ έπεται της $C_1^S$. Αφού η $U$ σειριοποιείται πριν την $S$ και η $S$ σειριοποιείται στην $C_1^S$, η $U$ δε δύναται να σειριοποιηθεί στην $w$ εντολή της. Οπότε υπάρχει μία SCAN $S'$ τέτοια ώστε η $w$ να επιτελείται μεταξύ της $C_1^{S'}$ και της $w_i^{S'}$ και η $U$ να σειριοποιείται ακριβώς πριν την $C_1^{S'}$. Αφού η $w$ έπεται της $w_i^S$, ισχύει ότι $g_0^S \neq g_0^{S'}$. Το Λήμμα 8.13 συνεπάγεται ότι τα διαστήματα εκτέλεσης $a(C_1^S, C_0^S)$ και $a(C_1^{S'}, C_0^{S'})$ δεν τέμνονται. Αν η $C_1^S$ προηγείται της $C_1^{S'}$, το σημείο σειριοποίησης της $U$ που τοποθετείται στην $C_1^{S'}$ έπεται του σημείου σειριοποίησης της $S$ που τοποθετείται στην $C_1^S$, το οποίο και είναι άτοπο. Αν η $C_1^S$ έπεται της $C_1^{S'}$, το Λήμμα 8.1 συνεπάγεται ότι η $w_i^{S'}$ προηγείται της $C_0^{S'}$. Έτσι η $w$ που επιτελείται μεταξύ της $C_1^{S'}$ και της $w_i^S$, το οποίο και είναι άτοπο καθώς υποθέσαμε ότι η $w$ έπεται της $w_i^S$.

2) Έστω ότι η $g_0^S$ ανέγνωσε τιμή ίση με $v_i$ στον καταχωρητή $post[i]$ (γραμμή 28). Σε αυτή την περίπτωση η $V_i^S$ είναι καλά ορισμένη. Συμβολίζουμε με $r_{pre}$ την read της $V_i^S$ στον καταχωρητή $pre[i]$ (γραμμή 27). Υπενθυμίζουμε ότι η $S$ επιστρέφει την τιμή που η $U_i^S$ χρησιμοποιεί ως παράμετρο και έτσι η $S$ επιστρέφει την τιμή που η $U_i^S$ έγγραψε στον καταχωρητή $pre[i]$. Στην περίπτωση όπου η $w$ προηγείται της $w_i^S$, το Λήμμα 8.14 συνεπάγεται ότι η $U$ σειριοποιείται πριν την $U_i^S$, το οποίο και είναι άτοπο. Έτσι υποθέτουμε ότι η $w$ έπεται της $w_i^S$. Τότε η $w$ πρέπει να έπεται της $r_{pre}$, αφού (από τον ορισμό της $U_i^S$) η $V_i^S$ διαβάζει την τιμή που η $w_i^S$ εγγράφει στον καταχωρητή $pre[i]$. Από το Λήμμα 8.16 προκύπτει ότι η $r_{pre}$ έπεται της $C_1^S$. Άρα η $w$ έπεται της $C_1^S$. Αφού η $U$ σειριοποιείται πριν την $S$ και η $S$ σειριοποιείται στην $C_1^S$, η $U$ δε δύναται να σειριοποιηθεί στην $w$ εντολή της. Οπότε υπάρχει μία SCAN $S'$ τέτοια ώστε η $w_i^{S'}$ να έπεται της $C_1^{S'}$, η $w$ να επιτελείται μεταξύ της $C_1^{S'}$ και της $w_i^{S'}$, και η $U$ να σειριοποιείται ακριβώς πριν την $C_1^{S'}$. Αφού η $w$ επιτελείται έπειτα από την $w_i^S$, θα ισχύει $g_c^{S'} \neq g_0^S$.

Το Λήμμα 8.13 συνεπάγεται ότι τα διαστήματα εκτέλεσης $a(C_1^S, C_0^S)$ και $a(C_1^{S'}, C_0^{S'})$ δεν επικαλύπτονται. Αν η $C_1^S$ προηγείται της $C_1^{S'}$, το σημείο σειριοποίησης της $U$ που τοποθετείται στην $C_1^{S'}$ έπεται του σημείου σειριοποίησης της $S$ που τοποθετείται στην $C_1^S$, το οποίο και είναι άτοπο. Ας υποθέσουμε επομένως ότι η $C_1^S$ έπεται της $C_1^{S'}$. Το Λήμμα 8.1 συνεπάγεται ότι η $w_i^S$ προηγείται της $C_0^{S'}$. Έ-

τσι η $w$ που επιτελείται μεταξύ της $C_1^{S'}$ και της $w_i^{S'}$, προηγείται της $C_1^S$, το οποίο και είναι άτοπο.

Σε όλες τις περιπτώσεις οδηγηθήκαμε σε άτοπο. Έτσι συμπεραίνουμε ότι δεν υπάρχει UPDATE στη συνιστώσα $A_i$ που να σειριοποιείται μεταξύ της $U_i^S$ και της $S$. Άρα η $S$ επιστρέφει ένα συνεπές διάνυσμα τιμών. ■

***Θεώρημα 8.18*** *Ο αλγόριθμος C-Snap είναι μία σειριοποιήσιμη, Multi-Scanner υλοποίηση ατομικών στιγμιοτύπων που πληροί την ιδιότητα ελεύθερη-αναμονής και επιτυγχάνει χρονική πολυπλοκότητα* $O(m)$ *για τη SCAN και* $O(1)$ *για την UPDATE, χρησιμοποιώντας* $m$ *καταχωρητές ανάγνωσης-εγγραφής μεγέθους* $O(log\,|T|)$ *bit και* $m+1$ *καταχωρητές CAS μεγέθους* $O(m\,log|T| + log\,k)$ *bit, όπου* $k$ *είναι ο μέγιστος αριθμός SCAN που εκτελούνται.*

# ΚΕΦΑΛΑΙΟ 9. Ο ΑΛΓΟΡΙΘΜΟΣ SWEEPLINE



Σε αυτό το κεφάλαιο παρουσιάζεται ένας Single-Scanner αλγόριθμος ατομικών στιγμιοτύπων, που ονομάζεται SweepLine. Ο SweepLine επιτυγχάνει χρονική πολυπλοκότητα $O(m^2)$ για τη SCAN και $O(m^2)$ για την UPDATE. Ο SweepLine χρησιμοποιεί $m+1$ καταχωρητές ανάγνωσης-εγγραφής μεγέθους $O(\mathrm{m} \cdot \log(|\mathrm{T}|) + \log k + \log n)$ bit, όπου $k$ είναι ο μέγιστος αριθμός λειτουργιών SCAN που εκτελούνται.

## 9.1. Περιγραφή του αλγορίθμου

Ο SweepLine χρησιμοποιεί $m+1$ καταχωρητές του οποίους τους συμβολίζουμε με $R_1, \ldots, R_m$ και $seq$ (βλ. Αλγόριθμο 9.1). Μία διεργασία που επιθυμεί να επιτελέσει μία UPDATE στη συνιστώσα $A_i, 1 \leq i \leq m$, εγγράφει μόνο τον καταχωρητή $R_i$. Ο καταχωρητής $seq$ εγγράφεται μόνο από τις SCAN. Αφού ο SweepLine είναι ένας αλγόριθμος Single-Scanner, σε κάθε χρονική στιγμή επιτρέπεται μία μόνο SCAN να είναι ενεργή. Ο $seq$ είναι ένας καταχωρητής ακεραίων, ο οποίος έχει αρχική τιμή ίση με 0 και η τιμή του αυξάνεται κάθε φορά που ξεκινάει την εκτέλεσή της μία νέα SCAN. Για κάθε $i, 1 \leq i \leq m$, ο καταχωρητής $R_i$ αποθηκεύει τις ακόλουθες πληροφορίες: (1) μία τιμή του συνόλου $T$ για τη συνιστώσα $A_i$, (2) το αναγνωριστικό (id) μίας διεργασίας $p$ (που επιτέλεσε την τελευταία εντολή write στον $R_i$), (3) έναν σειριακό αριθμό που χρησιμοποιείται από την $p$ για τη διάκριση των UPDATE της, (4) ένα πεδίο $curr\_seq$ στο οποίο αποθηκεύεται η τιμή του $seq$ που ανέγνωσε η $p$ κατά την έναρξη της επιτέλεσης της UPDATE, και (5) ένα διάνυσμα $m$ τιμών, μία για κάθε συνιστώσα.

Οι SCAN και UPDATE προσπαθούν να υπολογίσουν ένα συνεπές διάνυσμα τιμών καλώντας τη συνάρτηση $get\_vector$. Κάθε φορά που μία SCAN εκτελείται από μία διεργασία $p$, η $p$ αυξάνει την τιμή του $seq$ κατά 1, έπειτα καλεί την $get\_vector$ (με παράμετρο την νέα τιμή του $seq$) και επιστρέφει το διάνυσμα τιμών που υπολογίστηκε από αυτή. Κάθε διεργασία έχει μία τοπική μεταβλητή, την $timestamp$, η οποία έχει αρχική τιμή ίση με 0 και αυξάνεται κάθε φορά που η διεργασία επιτελεί μία UPDATE. Κατά τη διάρκεια εκτέλεσης μίας UPDATE στη συνιστώσα $A_i$ από μία διεργασία $p'$ συμβαίνουν τα ακόλουθα. Η διεργασία $p'$ διαβάζει την τιμή του $seq$ και καλεί την $get\_vector$ με παράμετρο την τιμή που ανέγνωσε στον $seq$. Στο τέλος η $p'$ εγγράφει στον $R_i$ τη νέα τιμή της συνιστώσας $A_i$, την αυξημένη τιμή της μεταβλητής $timestamp$, την τιμή που ανέγνωσε στο $seq$, και το διάνυσμα τιμών που επέστρεψε η $get\_vector$.

Αλγόριθμος 9.1 Ψευδοκώδικας για τον αλγόριθμο SweepLine.

```
  struct register{
      data value;
      int id;
      int timestamp;
      int curr_seq;
      data v[1..m];
  }

  shared struct register R[1..m];
  shared int seq=0;

  void UPDATE(int i, data value, int id, int timestamp){
      data view[1..m];
      int curr_seq;

1.    curr_seq=seq;
2.    timestamp = timestamp +1;
3.    view=get_vector(curr_seq);
4.    R[i]=<curr_seq, id, timestamp, value, view>;
  }

data[] SCAN(){
      data view[1..m];
      int curr_seq;

5.    curr_seq=seq.curr_seq+1;
6.    seq=curr_seq;
7.    view=get_vector(curr_seq);
8.    return view;
  }
```

Αλγόριθμος 9.2 Ψευδοκώδικας για την συνάρτηση get_vector.

```
   data[] get_vector(int curr_seq){
9.    struct h_data{
10.         register r;
11.         boolean change;
      };
      struct h_data history[0..m+2][1..m];
      int epoch=0;
      int i, j, k, SL[1..m];
      data v[m];
      struct register mp;

12.   for(i=1;i≤m;i++){
13.         history[epoch][i]=<R[i], false>;
      }
14.   epoch=epoch+1;

15.   while(true){
16.         for(i=1;i≤m;i++){
17.               mp=R[i];
18.               history[epoch][i]=<mp, false>;
19.               if(mp.curr_seqCurr≥seq)
20.                     return mp.v;
21.               if(history[epoch-1][i].r≠history[epoch][i].r)
22.                     history[epoch][i].r.change=true;
            }
23.         for(i=1;i≤m;i++)
24.               SL[i]=0;
25.         for(i=epoch;i≥1;i--){
26.            for(j=1;j≤m;j++)
27.                     if(history[i][j].change==true)
28.                           SL[j]=SL[j]+1;
29.            if,for each j,(SL[j]≠1 OR history[i][j].change≠true){
30.                     for(k=1;k≤m;k++)
31.                           v[k]=history[i][k].r.value;
32.                     return v;
               }
            }
33.         epoch=epoch+1;
     }
}
```

Μία $get\_vector$ $g$ επιτελεί συνεχώς ομάδες αναγνώσεων στους καταχωρητές $R_1, \ldots, R_m$ έως ότου ικανοποιηθεί μία από τις ακόλουθες συνθήκες.

1) Αν για κάποιο $j \in \{1, \ldots, m\}$, η τιμή του $seq$ είναι μεγαλύτερη ή ίση από την τιμή της παραμέτρου $curr\_seq$ της $g$, τότε η UPDATE που έγγραψε την εν λόγω τιμή

στον $R_j$, έχει ξεκινήσει την εκτέλεσή της έπειτα από την $g$ και έχει τερματίσει πριν την το τέλος της $g$. Σε αυτή την περίπτωση η $g$ επιστρέφει το διάνυσμα τιμών που ανέγνωσε στον καταχωρητή $R_j$.

2) Έστω ότι υπάρχει ένας ακέραιος $i > 0$, τέτοιος ώστε για κάθε καταχωρητή του οποίου τα περιεχόμενα έχουν αλλάξει από την $i-1$ στην $i$ ομάδα αναγνώσεων, η $get\_vector$ έχει δει τουλάχιστον μία ακόμη αλλαγή στον καταχωρητή αυτό στις επόμενες ομάδες αναγνώσεων. Η $g$ επιστρέφει την πρώτη φορά που πληρούται η εν λόγω συνθήκη και επιστρέφει τις τιμές που ανέγνωσε στα $R_1.value, \ldots, R_m.$value στην ομάδα αναγνώσεων $i$.

Για να ανιχνευθεί αν πληρούται η συνθήκη (2) χρησιμοποιείται μία τεχνική σάρωσης (sweeping technique). Κάθε διεργασία χρησιμοποιεί έναν δισδιάστατο πίνακα $m$ στηλών (η $i$-οστή στήλη αντιστοιχίζεται στην $i$-οστή συνιστώσα του στιγμιοτύπου) που ονομάζεται $history$. Στη γραμμή 0 του $history$ (ο $history$ αντίθετα από όλους τους υπόλοιπους πίνακες ξεκινά από τη γραμμή 0) αποθηκεύονται πληροφορίες για το τι αναγνώσθηκε στην πρώτη ομάδα αναγνώσεων (γραμμή 10), ενώ στη γραμμή $i$, $1 \leq i \leq m+1$ αποθηκεύονται πληροφορίες για το τι αναγνώσθηκε στην ομάδα αναγνώσεων $i+1$ (θα αποδειχθεί αργότερα ότι κάθε $get\_vector$ εκτελεί το πολύ $m+2$ ομάδες αναγνώσεων και έτσι το πλήθος των γραμμών του $history$ είναι αρκετό). Το πεδίο $history[i][j].change, 1 \leq j \leq m$ έχει τιμή ίση με $true$ αν τα περιεχόμενα του καταχωρητή $R_j$ έχουν αλλάξει ανάμεσα στην $i-1$ και στην $i$ ομάδα αναγνώσεων. Μία φανταστική γραμμή σάρωσης (sweep line) που υλοποιείται από τον πίνακα $SL$, σαρώνει τις γραμμές του πίνακα $history$ κατευθυνόμενη από την τελευταία γραμμή προς την πρώτη. Ο πίνακας $SL$ υπολογίζει πόσες φορές η τιμή κάθε καταχωρητή έχει αλλάξει (γραμμές 23-24). Έτσι δύναται να βρεθεί η πρώτη ομάδα αναγνώσεων που ενδεχομένως πληροί τη συνθήκη (2) (γραμμή 29) και να επιστραφεί σε αυτή την περίπτωση το κατάλληλο διάνυσμα τιμών (γραμμή 32). Η συνθήκη (2) εγγυάται ότι η $S$ τερματίζει ακόμα και στην περίπτωση που δεν δει μία τέτοια UPDATE. Αξίζει να σημειωθεί ότι αν οι τιμές των συνιστωσών σε δύο συνεχόμενες ομάδες αναγνώσεων δεν αλλάξουν, η συνθήκη (2) θα πληρούται τετριμμένα και η $S$ σε αυτή την περίπτωση επιστρέφει το διάνυσμα τιμών που αναγνώσθηκε σε οποιαδήποτε από αυτές τις δύο ομάδες αναγνώσεων.

### 9.2. Χρονική και χωρική πολυπλοκότητα του αλγορίθμου

Έστω $a$ μία οποιαδήποτε εκτέλεση του SweepLine και έστω $g$ μία εκτέλεση της $get\_vector$ στην $a$. Ας υποθέσουμε ότι η $g$ εκτελεί $m+1$ επαναλήψεις του βρόχου while (γραμμή 15) και δεν έχει τερματίσει. Από την συνθήκη της γραμμής 29 προκύπτει ότι για κάθε $i = m+1, \ldots, 1$ υπάρχει τουλάχιστον ένας ακέραιος $j \in \{1, \ldots, m\}$ για το οποίο ισχύει ότι $SL[j] == 1$ και $history[i][j].change == true$. Έτσι σε κάθε μία από τις εν λόγω επαναλήψεις υπάρχει ένας τουλάχιστον καταχωρητής (και άρα μία συνιστώσα) που αλλάζει για τελευταία φορά μεταξύ της εκτέλεσης της τρέχουσας επανάληψης και της προηγούμενης. Αυτό είναι άτοπο, καθώς το αντικείμενο ατομικών στιγμιοτύπων έχει μόνο $m$ συνιστώσες. Έτσι έπειτα από το πολύ $m+1$ επαναλήψεις του βρόχου while, η $get\_vector$ ολοκληρώνει την εκτέλεσή της. Σε κάθε επανάληψη του βρόχου while, $m$ διαφορετικοί καταχωρητές διαβάζονται. Έτσι η χρονική πολυπλοκότητα της $get\_vector$ είναι $O(m^2)$. Κάθε SCAN ή UPDATE εκτός από την κλήση της $get\_vector$ επιτελεί ένα σταθερό αριθμό προσπελάσεων στην κοινή μνήμη. Άρα η πολυπλοκότητα της SCAN και της UPDATE είναι $O(m^2)$.

Από τον ψευδοκώδικα του αλγορίθμου είναι προφανές ότι ο SweepLine χρησιμοποιεί $m+1$ καταχωρητές κοινής μνήμης. Ο καταχωρητής seq έχει μέγεθος $O(\log k)$ bit, όπου $k$ είναι το πλήθος των SCAN που επιτελούνται σε μία εκτέλεση. Οι καταχωρητές $R_1, \ldots, R_m$ αποθηκεύουν ένα διάνυσμα $m$ τιμών του συνόλου $T$, τον ακέραιο $curr\_seq$ που έχει μέγεθος $\log k$ bit, μία τιμή του συνόλου $T$ (value) και ένα αναγνωριστικό της διεργασίας που ενημέρωσε τον καταχωρητή τελευταία. Άρα το μέγεθος των καταχωρητών $R_1, \ldots, R_m$ θα είναι $O(\log n + \log k + m \log|T|)$ bit.

### 9.3. Απόδειξη της ορθότητας του αλγορίθμου

Έστω $S$ μία οποιοδήποτε SCAN και $g$ η $get\_vector$ που εκτελείται από την $S$. Υποθέτουμε ότι η $g$ τερματίζει τη λειτουργία της στη γραμμή 20 του ψευδοκώδικα. Έστω $k(g)$ ο μεγαλύτερος ακέραιος, τέτοιος ώστε να υπάρχει μία ακολουθία από UPDATE $U_1, \ldots, U_{k(g)}$ τέτοιες ώστε: (1) $U_{k(g)}$ είναι η UPDATE που εγγράφει το διάνυσμα τιμών που επιστρέφει η $g$, και (2) για κάθε $l, 1 < l \leq k(g)$, η $get\_vector$ $g_l$ που εκτε-

λείται από την $U_l$ ολοκληρώνει στη γραμμή 20 του ψευδοκώδικα και επιστρέφει το διάνυσμα τιμών που εγγράφηκε από την $U_{l-1}$. Έστω η ακολουθία των UPDATE $SU(g) = U_1, \ldots, U_{k(g)}$ και έστω η ακολουθία των $get\_vector\ SG(g) = g_1, \ldots, g_{k(g)}$. Στην περίπτωση όπου η $g$ δεν τερματίζει στη γραμμή 20 του ψευδοκώδικα, θα ισχύει $SU(g) = SG(g) = \lambda$ (όπου $\lambda$ είναι η κενή ακολουθία). Αξίζει να σημειωθεί ότι αν ισχύει $SG(g) \neq \lambda$, η $g$ και κάθε μία από τις συναρτήσεις $g_1, \ldots, g_{k(g)} \in SG(g)$ επιστρέφουν το ίδιο διάνυσμα τιμών. Επιπρόσθετα η $g_1$ επιστρέφει εκτελώντας τη γραμμή 32 του ψευδοκώδικα. Έστω $gv(S) = g_1$ αν ισχύει $SG(g) \neq \lambda$, διαφορετικά $gv(S) = g$. Ως άμεση συνέπεια των παραπάνω ορισμών συμπεραίνουμε ότι $gv(S)$ ολοκληρώνει στη γραμμή 32 του ψευδοκώδικα και η $S$ επιστρέφει το ίδιο διάνυσμα τιμών με αυτό που υπολόγισε η $gv(S)$.

Υποθέτουμε ότι μία $get\_vector\ g$ ολοκληρώνεται έπειτα από $epoch \geq 1$ επαναλήψεις του βρόχου while (γραμμή 15). Για κάθε $1 \leq l \leq epoch$, συμβολίζουμε με $SL_l^g$ τις τιμές που έχει ο πίνακας έπειτα από την εκτέλεση της $l$-οστής ομάδας αναγνώσεων (ο εκθέτης δύναται να παραληφθεί και θα καθορίζεται από τα συμφραζόμενα).

Σε αυτό το σημείο θα αναθέσουμε σημεία σειριοποίησης. Έστω $S$ μία οποιαδήποτε SCAN σε μία εκτέλεση $a$. Είναι εύβολο να αναθέσουμε σημείο σειριοποίησης και στην $gv(S)$. Υποθέτουμε ότι η $gv(S)$ επιστρέφει εκτελώντας $epoch \geq 1$ επαναλήψεις του βρόχου while. Υπενθυμίζουμε ότι η $gv(S)$ τερματίζει εκτελώντας τη γραμμή 32 του ψευδοκώδικα. Έτσι υπάρχει ένας ακέραιος $l', 1 \leq l' \leq epoch$, τέτοιος ώστε για κάθε $j \in \{1, \ldots, m\}$ να ισχύει $SL_{l'} \neq 1$ ή να ισχύει $history[l'][j] \neq true$. Έστω $l'$ ο μεγαλύτερος τέτοιος ακέραιος. Υποθέτουμε ότι οι τιμές $d$ καταχωρητών $R_{x1}, \ldots, R_{xd}$ έχουν αλλάξει μεταξύ της $l$-οστής και της $(l+1)$-οστής ομάδας αναγνώσεων. Έστωσαν $v_{x1}, \ldots, v_{xd}$ τα περιεχόμενα των $R_{x1}, \ldots, R_{xd}$ που αναγνώσθηκαν από την $gv(S)$, και έστωσαν $U_{x1}, \ldots, U_{xd}$ οι UPDATE που έγραψαν τις τιμές $v_{x1}, \ldots, v_{xd}$ στους $R_{x1}, \ldots, R_{xd}$. Αναθέτουμε σημεία σειριοποίησης ως εξής.

- Αναθέτουμε σημείο σειριοποίησης στην $gv(S)$ στο σημείο όπου επιτελεί την πρώτη ανάγνωση της $l$-οστής ομάδας αναγνώσεων. Στο ίδιο ακριβώς σημείο εισάγουμε σημείο σειριοποίησης και στην $S$.

- Για τις UPDATE που επιτέλεσαν τις write τους έπειτα από την πρώτη εντολή read της $gv(S)$, εισάγουμε σημείο σειριοποίησης ακριβώς πριν το σημείο σειριοποίησης της $gv(S)$ (με οποιαδήποτε σειρά αφού ενημέρωσαν διαφορετικές συνιστώσες).
- Αφότου αναθέσουμε σημεία σειριοποίησης σε όλες τις SCAN, αλλά και σε μερικές UPDATE με τον τρόπο που περιγράφηκε παραπάνω, σειριοποιούμε τις υπόλοιπες UPDATE στο σημείο όπου επιτέλεσαν την εντολή write τους.

Η κεντρική ιδέα της ορθότητας του αλγορίθμου είναι η ακόλουθη. Προκύπτει ότι όλες οι τιμές που αναγνώσθηκαν από την $gv(S)$ στην $l$-οστή ομάδα αναγνώσεων έχουν εγγραφεί από UPDATE που ξεκίνησαν την εκτέλεσή τους πριν η $S$ εγγράψει τον $seq$. Έστωσαν $r_1,\ldots,r_m$ οι εντολές read που επιτελέσθηκαν από την $gv(S)$ κατά τη διάρκεια της $l$-οστής ομάδας αναγνώσεων. Αν μία τέτοια UPDATE $U$ επιτελέσει την write της $w$ στη συνιστώσα $A_j$ πριν την $r_1$, σειριοποιείται στην $w$. Γνωρίζουμε ότι καμία άλλη UPDATE δεν έχει εγγράψει τον καταχωρητή $R_j$ μεταξύ της $w$ και της $r_1$ (αφού σε κάθε άλλη περίπτωση η $gv(S)$ δε θα επέστρεφε την τιμή της $U$). Αυτό δύναται να χρησιμοποιηθεί ώστε να αποδειχθεί η συνέπεια της τιμής που επιστρέφεται για τη συνιστώσα $A_j$ από την $gv(S)$. Στην περίπτωση όπου η $U$ επιτελεί την $w$ έπειτα από την $r_1$, μετακινούμε το σημείο σειριοποίησης της $U$ ακριβώς πριν την $r_1$ ώστε η τιμή που επιστρέφεται για τη συνιστώσα $A_j$ να είναι συνεπής. Αξίζει να σημειωθεί ότι η $w$ δύναται να πανωγράψει την τιμή που έγραψε μία άλλη UPDATE $U'$ στη συνιστώσα $A_j$ που επιτελέσθηκε μεταξύ της $r_1$ και της $w$. Αφού η $S$ δε διαβάζει την τιμή της $U'$ στην $l$-οστή ομάδα αναγνώσεων, η $U'$ σειριοποιείται στην write της (έπειτα από σημείο σειριοποίησης της $U$). Ωστόσο αυτό δύναται να δημιουργήσει προβλήματα στη συνέπεια των διανυσμάτων που επιστρέφουν σε SCAN που έπονται της $S$, αφού θα δουν στον καταχωρητή $R_j$ την τιμή που έγγραψε η $U$ και όχι την τιμή που έγγραψε η $U'$. Για αυτό τον λόγο η συνθήκη (2) απαιτεί η $S$ να έχει δει περισσότερες από μία UPDATE στη συνιστώσα $A_j$ κατά την επιτέλεση της $l$-οστής ομάδας αναγνώσεων. Η εν λόγω UPDATE σειριοποιείται στην write της και η write της πανωγράφει την τιμή που έγγραψε η $U'$.

Το ακόλουθο λήμμα είναι άμεση συνεπαγωγή του ορισμού της συνάρτησης $gv(S)$.

**Λήμμα 9.1** *Για κάθε SCAN S,*

1. *η $gv(S)$ επιστρέφει εκτελώντας τη γραμμή 32 του ψευδοκώδικα,*
2. *η S επιστρέφει το ίδιο διάνυσμα τιμών με αυτό που επιστρέφει η $gv(S)$.*

Θα αποδειχθεί ότι το σημείο σειριοποίησης κάθε SCAN και UPDATE βρίσκεται εντός του διαστήματος εκτέλεσης της λειτουργίας. Αρχικά θα αποδειχθούν τρία τεχνικά λήμματα.

**Λήμμα 9.2** *Έστω S μία οποιαδήποτε SCAN και έστω $g$ η συνάρτηση $get\_vector$ που εκτελείται από την S. Έστω ότι ισχύει $SU(g) = U_1,\ldots,U_{k(g)} \neq \lambda$. Για κάθε $1 < j < k(g)$ ισχύουν τα ακόλουθα:*

1. *η $U_{j-1}$ επιτελεί την write της πριν η $U_j$ επιτελέσει την write της, και*
2. *η $U_{j-1}$ έχει αναγνώσει στον $seq$ τιμή μεγαλύτερη ή ίση από αυτή που η $U_j$ ανέγνωσε στο $seq$.*

**Απόδειξη.** Έστω $SG(g) = g_1,\ldots,g_{k(g)}$. Από τον τρόπο ορισμού της ακολουθίας $SU(g)$ και της ακολουθίας $SG(g)$, για κάθε $1 < j \leq k(g)$, η $g_j$ επιστρέφει το διάνυσμα τιμών που εγγράφηκε από την $U_{j-1}$. Έτσι η $U_{j-1}$ εκτελεί την write της πριν το τέλος της $g_j$, και άρα πριν το τέλος της $U_j$, όπως και απαιτείται από τον ισχυρισμό 1 του λήμματος.

Επιπρόσθετα η $g_j$ τερματίζει εκτελώντας τη γραμμή 20 του ψευδοκώδικα, έτσι η συνθήκη της γραμμή 19 έχει αποτιμηθεί ως αληθής. Επομένως η $U_{j-1}$ θα έχει αναγνώσει στο $seq$ τιμή μεγαλύτερη ή ίση από αυτή που ανέγνωσε η $U_j$, όπως και απαιτείται από τον ισχυρισμό 2 του λήμματος. ∎

**Λήμμα 9.3** *Έστω S μία οποιοδήποτε SCAN και έστω $g$ η $get\_vector$ που κλήθηκε από την S. Έστω η ακολουθία $SU(g) = U_1,\ldots,U_{k(g)} \neq \lambda$. Για κάθε $1 \leq j < k(g)$, η τιμή που η $U_j$ ανέγνωσε στον καταχωρητή $R_s$ είναι αυτή που εγγράφηκε από την S.*

**Απόδειξη.** Αρχικά θα αποδειχθεί ότι η $U_{k(g)}$ διαβάζει στον $seq$ την τιμή που έγγραψε η $S$. Από τον τρόπο ορισμού της $SU(g)$, η $g$ επιστρέφει το διάνυσμα τιμών που εγγράφηκε από την $U_{k(g)}$. Έτσι η $g$ (και άρα και η $S$) τερματίζουν έπειτα από το πέρας

της $U_{k(g)}$. Επιπρόσθετα η $g$ επιστρέφει εκτελώντας τη γραμμή 20 του ψευδοκώδικα, και έτσι η συνθήκη της γραμμής 19 είναι αληθής. Επομένως η $U_{k(g)}$ έχει αναγνώσει στο $seq$ τιμή μεγαλύτερη ή ίση από αυτή που εγγράφηκε από την $S$ (η οποία ισούται με την παράμετρο $curr\_seq$ με την οποία κλήθηκε η $g$). Αφού ο καταχωρητής $seq$ εγγράφεται μόνο από SCAN, μία μόνο SCAN εκτελείται σε κάθε χρονική στιγμή, και η $S$ τερματίζει έπειτα από το τέλος της $U_{k(g)}$, η $U_{k(g)}$ δε δύναται να ανέγνωσε κάποια τιμή μεγαλύτερη από αυτή που έγγραψε η $S$ στον $seq$. Έτσι η $U_{k(g)}$ διαβάζει στον $seq$ την τιμή που έγγραψε η $S$.

Από τον ισχυρισμό 1 του Λήμματος 9.2 συνεπάγεται ότι για κάθε $1 \leq j < k(g)$, η $U_j$ τερματίζει πριν το τέλος της $U_{k(g)}$. Έτσι προκύπτει ότι η $U_j$ τερματίζει πριν το τέλος της $S$. Έτσι η $U_j$ δε δύναται να έχει αναγνώσει στον $R_s.curr\_seq$ τιμή μεγαλύτερη από αυτή που η $S$ έγγραψε. Από τον ισχυρισμό 2 του λήμματος 9.2 προκύπτει ότι η $U_j$ ανέγνωσε στο $seq$ τιμή μεγαλύτερη ή ίση από αυτή που ανέγνωσε η $U_{k(g)}$. Αφού η τιμή στον $seq$ που διαβάσθηκε από την $U_{k(g)}$ είναι αυτή που έγγραψε η $S$, προκύπτει ότι η $U_j$ ανέγνωσε στο $seq$ την τιμή που εγγράφηκε από την $S$. ∎

**Λήμμα 9.4** *Έστω $S$ μία οποιαδήποτε SCAN $S$ και έστω $g$ η συνάρτηση $get\_vector$ που εκτελέστηκε από την $S$. Για κάθε συνάρτηση $get\_vector$ $g' \in SG(g)$, το διάστημα εκτέλεσης της $g'$ ξεκινά αφότου η $S$ εγγράφει στον καταχωρητή $seq$ και τελειώνει πριν το τέλος της $S$.*

**Απόδειξη.** Έστω $U'$ η UPDATE που καλεί την $g'$. Αφού ισχύει $g' \in SG(g)$, θα ισχύει $U' \in SU(g)$. Από το Λήμμα 9.3 προκύπτει ότι η $U'$ διαβάζει στον $seq$ την τιμή που έγγραψε η $S$. Αφού η $U'$ ξεκίνησε την εκτέλεσή της διαβάζοντας τον καταχωρητή $seq$, προκύπτει ότι η $U'$ (και έτσι και η $g'$) ξεκίνησε την εκτέλεσή της έπειτα από την εγγραφή της $S$ στον $seq$. Από τον τρόπο ορισμού της $SU(g)$, η $g$ επιστρέφει το διάνυσμα τιμών που εγγράφηκε από την $U_{k(g)}$. Έτσι η write της $U_{k(g)}$ προηγείται του τέλους της $g$ (και έτσι του τέλους της $S$). Το Λήμμα 9.2 συνεπάγεται ότι η write της $U'$ προηγείται της write της $U_{k(g)}$. Έτσι προκύπτει ότι η $U'$ (και έτσι και η $g'$) τελειώνει την εκτέλεσή της πριν το τέλος της $S$. Άρα το διάστημα εκτέλεσης της $g'$ ξεκινά

έπειτα από την εγγραφή του καταχωρητή $seq$ από την $S$ και τελειώνει πριν την το τέλος της $S$. ∎

Σε αυτό το σημείο θα αποδειχθεί ότι τα σημεία σειριοποίησης των SCAN και UPDATE βρίσκονται εντός του διαστήματος εκτέλεσης των λειτουργιών τους.

**Λήμμα 9.5** *Έστω $a$ μία οποιαδήποτε εκτέλεση του αλγορίθμου SweepLine. Το σημείο σειριοποίησης οποιασδήποτε SCAN ή UPDATE βρίσκεται εντός του διαστήματος εκτέλεσής της.*

**Απόδειξη.** Έστω $S$ μία οποιοδήποτε SCAN στην $a$ και έστω $g$ η συνάρτηση $get\_vector$ που εκτελείται από την $S$. Υπενθυμίζεται ότι η $S$ σειριοποιείται στο ίδιο σημείο με την $gv(S)$ και επιστρέφει το ίδιο διάνυσμα τιμών με την $gv(S)$. Είναι προφανές ότι το διάστημα εκτέλεσης της $g$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Στην περίπτωση που ισχύει $SG(g) = g_1, \dots, g_{k(g)} \neq \lambda$, το Λήμμα 9.4 συνεπάγεται ότι το διάστημα εκτέλεσης της $g_1$ εμπεριέχεται στο διάστημα εκτέλεσης της $S$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, το σημείο σειριοποίησης της $gv(S)$ βρίσκεται εντός του διαστήματος εκτέλεσής της. Έτσι προκύπτει ότι το σημείο σειριοποίησης της $S$ βρίσκεται εντός του διαστήματος εκτέλεσής της, όπως και απαιτείται.

Έστω $U$ μία οποιαδήποτε UPDATE στην $a$. Αν το σημείο σειριοποίησης της $U$ έχει τεθεί στο σημείο όπου επιτέλεσε την write της, τότε προφανώς το σημείο σειριοποίησης της $U$ βρίσκεται εντός του διαστήματος εκτέλεσής της. Έτσι υποθέτουμε ότι δεν ισχύει η προαναφερθείσα περίπτωση. Έτσι υπάρχει μία SCAN $S'$, τέτοια ώστε το σημείο σειριοποίησης της $U$ να έχει τοποθετηθεί ακριβώς πριν το σημείο σειριοποίησης της $g(S')$. Υποθέτουμε ότι η $g(S')$ επιστρέφει αφότου εκτελέσει $epoch$ επαναλήψεις του βρόχου while (γραμμή 15). Από το Λήμμα 9.1 προκύπτει ότι η $g(S')$ τερματίζει εκτελώντας τη γραμμή 32 του ψευδοκώδικα. Έτσι υπάρχει ένας ακέραιος $l', l \leq l' \leq epoch$ τέτοιος ώστε για κάθε $j \in \{1, \dots, m\}$ να ισχύει $SL_{l'}[j] \neq 1$ ή $history[l'][j].change \neq true$. Έστω $l$ ο μεγαλύτερος τέτοιος ακέραιος. Υποθέτουμε ότι οι τιμές των $d$ καταχωρητών $R_{x1}, \dots, R_{xd}$ έχουν αλλάξει ανάμεσα στην $(l-1)$-οστή και στην $l$-οστή ομάδα αναγνώσεων. Έστωσαν $v_{x1}, \dots, v_{xd}$ τα περιεχόμενα των καταχωρητών $R_{x1}, \dots, R_{xd}$ που αναγνώσθηκαν από την $gv(S')$, και

έστωσαν $U_{x1}, \ldots, U_{xd}$ οι UPDATE που έγραψαν τις τιμές $v_{x1}, \ldots, v_{xd}$ στους καταχωρητές $R_{x1}, \ldots, R_{xd}$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $U$ είναι μία από τις $U_{x1}, \ldots, U_{xd}$. Επιπρόσθετα η $U$ έχει επιτελέσει την write της έπειτα από το σημείο σειριοποίησης της $gv(S)$.

Θα δείξουμε ότι η $U$ έχει ξεκινήσει την εκτέλεσή της πριν την έναρξη της $gv(S')$. Το διάστημα εκτέλεσης της συνάρτησης $get\_vector\ g'$ που εκτελείται από την $S'$, προφανώς εμπεριέχεται στο διάστημα εκτέλεσης της $S'$. Τα προαναφερθέντα και το Λήμμα 9.4 συνεπάγονται ότι το διάστημα εκτέλεσης της $gv(S')$ εμπεριέχονται στο διάστημα εκτέλεσης της $S'$. Αφού η $gv(S')$ διαβάζει την τιμή που εγγράφηκε από την $U$, από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι η $U$ διαβάζει στο $seq$ μία τιμή μεγαλύτερη ή ίση από αυτή που εγγράφηκε από την $S'$. Έτσι η $gv(S')$ θα τερματίσει εκτελώντας τη γραμμή 20 του ψευδοκώδικα, το οποίο και αντιτίθεται στον ισχυρισμό 1 του Λήμματος 9.1.

Έτσι η $U$ διαβάζει στον $seq$ μία τιμή μικρότερη από αυτή που έγγραψε η $S'$, έτσι η $U$ επιτελεί την πρώτη read της πριν η $S'$ εγγράψει τον καταχωρητή $seq$. Συμπεραίνουμε ότι το διάστημα εκτέλεσης της $U$ ξεκινά πριν μία από τις $gv(S')$, $U$ επιτελέσει την write της έπειτα από το σημείο σειριοποίησης της $gv(S')$. Το σημείο σειριοποίησης της $U$ τίθεται ακριβώς πριν το σημείο σειριοποίησης της $gv(S')$. Συμπεραίνουμε ότι το σημείο σειριοποίησης της $U$ βρίσκεται εντός του διαστήματος εκτέλεσής της. ■

Τέλος θα αποδειχθεί ότι οι SCAN επιστρέφουν συνεπή διανύσματα τιμών. Ακολουθεί η απόδειξη ενός τεχνικού λήμματος.

**Λήμμα 9.6** *Έστω U μία οποιοδήποτε UPDATE σε κάποια συνιστώσα $A_j$, τέτοια ώστε το σημείο σειριοποίησης της U να μην έχει τοποθετηθεί στο σημείο όπου η U επιτέλεσε την write της. Σε αυτή την περίπτωση υπάρχει μία SCAN S, ώστε να ισχύουν τα ακόλουθα:*

1. *η S επιστρέφει για τη συνιστώσα $A_j$ την τιμή που έγγραψε η U,*
2. *η εντολή write w της U προηγείται του τέλους της S,*
3. *το σημείο σειριοποίησης της U περιέχεται στο διάστημα εκτέλεσης της S, υπάρχει μία UPDATE $U'$ τέτοια ώστε:*

*a. η $w$ να προηγείται της εντολή write $w'$ της $U'$,*

*b. η $w'$ να προηγείται του τέλους της $S$, και*

*c. το σημείο σειριοποίησης της $U'$ να έπεται του σημείου σειριοποίησης της $S$.*

**Απόδειξη.** Έστω $U$ μία οποιαδήποτε UPDATE στη συνιστώσα $A_j$ που να μην έχει σειριοποιηθεί στην write της. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S$, τέτοια ώστε η $gv(S)$ να επιστρέψει την τιμή της $U$ για τη συνιστώσα $A_j$ και η $U$ να σειριοποιείται ακριβώς πριν το σημείο σειριοποίησης της $gv(S)$. Από το Λήμμα 9.1 προκύπτει ότι η $gv(S)$ τερματίζει εκτελώντας τη γραμμή 32 του ψευδοκώδικα. Υποθέτουμε ότι η $gv(S)$ εκτελεί $epoch$ επαναλήψεις του βρόχου while (γραμμή 15) προτού επιστρέψει. Τότε υπάρχει κάποιος ακέραιος $l', 1 \leq l' \leq epoch$ τέτοιος ώστε για κάθε $j \in \{1, \ldots, m\}$, ισχύει ότι $SL_{l'}[j] \neq 1$ ή $history[l'][j].change \neq true$. Έστω $l$ ο μεγαλύτερος τέτοιος ακέραιος. Έστω $r_1$ η πρώτη εντολή read της $gv(S)$ στην $l$-οστή εποχή και έστω $r_j$ η εντολή read στον καταχωρητή $R_j$ (που αντιστοιχεί στη συνιστώσα $A_j$) της συνάρτησης $gv(S)$ στην $l$-οστή εποχή. Υπενθυμίζεται ότι η $U$ έχει σειριοποιηθεί ακριβώς πριν την $gv(S)$, η οποία σειριοποιείται στην $r_1$. Από τον τρόπο ανάθεσης σημείων σειριοποίησης, η write της $U$ πρέπει να έπεται της $r_1$, έτσι η write της $U$ έπεται του σημείου σειριοποίησης της $gv(S)$. Επιπρόσθετα κατά τη διάρκεια της $l$-οστής εποχής, η $gv(S)$ διαβάζει στον καταχωρητή $R_j$ ότι έγγραψε η $U$ και επιστρέφει την τιμή της $U$ για τη συνιστώσα $A_j$. Από το Λήμμα 9.1, η $S$ επιστρέφει το ίδιο διάνυσμα τιμών με αυτό που επιστρέφει η $gv(S)$. Έτσι η $S$ επιστρέφει την τιμή της $U$ για τη συνιστώσα $A_j$, όπως και απαιτείται από τον ισχυρισμό 1 του λήμματος.

Αφού η $gv(S)$ διαβάζει στον καταχωρητή $R_j$ ότι έγγραψε η $U$, προκύπτει ότι η μοναδική εντολή write της $U$ επιτελείται πριν το τέλος της $gv(S)$. Έστω $g$ η συνάρτηση $get\_vector$ που εκτελέστηκε από την $S$. Είναι προφανές ότι η $g$ τερματίζει πριν το τέλος της $S$. Αυτό και το λήμμα 9.2 συνεπάγονται ότι η $gv(S)$ τερματίζει πριν το τέλος της $S$. Έτσι προκύπτει ότι η write της $U$ προηγείται του τέλους της $S$, όπως και απαιτείται από τον ισχυρισμό 2 του λήμματος.

Το σημείο σειριοποίησης της $U$ εισάγεται ακριβώς πριν από το σημείο σειριοποίησης της $gv(S)$. Αφού η $S$ σειριοποιείται στο ίδιο σημείο με την $gv(S)$, το σημείο σειριοποίησης της $U$ τίθεται ακριβώς πριν από το το σημείο σειριοποίησης της $S$. Από το Λήμμα 9.5 προκύπτει ότι το σημείο σειριοποίησης της $S$ βρίσκεται εντός του διαστήματος εκτέλεσης της $S$. Έτσι το σημείο σειριοποίησης της $U$ βρίσκεται εντός του διαστήματος εκτέλεσης της $S$, όπως και απαιτείται από τον ισχυρισμό 3 του λήμματος.

Τέλος θα αποδείξουμε τον ισχυρισμό 4 του λήμματος. Αφού η write της $U$ επιτελείται έπειτα από την $r_1$ και πριν την $r_j$, η πρώτη φορά που η $gv(S)$ ανέγνωσε την τιμή που έγγραψε η $U$ στον καταχωρητή $R_j$ είναι η στιγμή στην οποία εκτέλεσε την $r_j$ (δηλαδή κατά τη διάρκεια της $l$-οστής ομάδας αναγνώσεων). Από τον ψευδοκώδικα του αλγορίθμου προκύπτει ότι ισχύει $history[l][j] == true$. Επιπρόσθετα ισχύει ότι $SL_l[j] \neq 1$ ή $history[l][j] \neq true$, έτσι θα ισχύει $SL_l[j] > 1$. Προκύπτει ότι υπάρχει κάποιος ακέραιος $l' > l$ τέτοιος ώστε να ισχύει $history[l][j].change == true$ (δηλαδή τα δεδομένα του καταχωρητή $R_j$ να έχουν αλλάξει ανάμεσα στην $(l-1)$-οστή και στην $l$-οστή εποχή). Έτσι υπάρχει μία UPDATE $U'$ που επιτελεί την εντολή write της $w'$ έπειτα από την εντολή write $w$ της $U$ και πριν το τέλος του διαστήματος εκτέλεσης της $gv(S)$ (και έτσι πριν το τέλος της $S$), όπως και απαιτείται από τους ισχυρισμούς 4/a και 4/b του λήμματος.

Από το Λήμμα 9.5 προκύπτει ότι το σημείο σειριοποίησης της $U'$ βρίσκεται εντός του διαστήματος εκτέλεσής της. Αφού η $U'$ ολοκληρώνει τη λειτουργία της στην write της (η οποία εκτελείται πριν το τέλος της $S$), το σημείο σειριοποίησης της της $U'$ δε δύναται να τοποθετηθεί έπειτα από το τέλος του διαστήματος εκτέλεσης της $S$. Σε αυτό το σημείο θα αποδειχθεί ότι το σημείο σειριοποίησης της $U'$ τοποθετείται στην write της και έτσι το σημείο σειριοποίησης της $U'$ έπεται του σημείου σειριοποίησης της $gv(S)$ και της $S$. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι τα προαναφερθέντα δεν ισχύουν. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, υπάρχει μία SCAN $S'$, τέτοια ώστε η $S'$ να επιστρέφει την τιμή της $U'$ για τη συνιστώσα $A_j$ και το σημείο σειριοποίησης της $U'$ να τοποθετείται ακριβώς πριν το σημείο σειριοποίησης της $gv(S')$ (και άρα της $S$). Από το Λήμμα 9.5 προκύπτει ότι το σημείο σειριοποίησης της $S'$ βρίσκεται εντός του διαστήματος εκτέλεσής της, έτσι

το σημείο σειριοποίησης της $U'$ βρίσκεται εντός του διαστήματος εκτέλεσης της $S'$. Αφού η $S$ επιστρέφει την τιμή της $U$ και όχι την τιμή της $U'$ για τη συνιστώσα $A_j$, η $S'$ είναι μία διαφορετική SCAN από την $S$. Αφού η write της $U'$ επιτελέσθηκε πριν το τέλος του διαστήματος εκτέλεσης της $S$, το σημείο σειριοποίησης της $U'$ προηγείται του τέλους της $S$. Έτσι η η $S'$ δεν έπεται τη $S$ και άρα η $S'$ πρέπει να προηγείται της $S$.Αφού η $S'$ επιστρέφει την της $U'$ για τη συνιστώσα $A_j$, η $w'$ προηγείται του τέλους του διαστήματος εκτέλεσης της $S'$. Έτσι η $w'$ προηγείται της $w$, η οποία επιτελείται στο διάστημα εκτέλεσης της $S$ (μεταξύ της $r_1$ και της $r_j$), το οποίο και είναι άτοπο. ■

**Λήμμα 9.7** *Για κάθε SCAN S, το διάνυσμα τιμών που επιστρέφεται από τη συνάρτηση $gv(S)$ είναι συνεπές.*

**Απόδειξη.** Έστω $\vec{v} = v_1,\ldots,v_m$ το διάνυσμα τιμών που η επιστρέφει η συνάρτηση $gv(S)$. Για κάθε $j \in \{1,\ldots,m\}$, $v_j$ είναι η τιμή της τελευταίας UPDATE στη συνιστώσα $A_j$ που σειριοποιήθηκε πριν την $gv(S)$. Χρησιμοποιώντας τη μέθοδο της εις άτοπο απαγωγής υποθέτουμε ότι υπάρχει κάποιος ακέραιος $j \in \{1,\ldots,m\}$, τέτοιος ώστε η παράμετρος της τελευταίας UPDATE $U$ στην $A_j$ που σειριοποιήθηκε πριν από την $gv(S)$ να μην είναι $v_j$. Έστω $v$ η παράμετρος της $U$ και έστω $U_j$ η UPDATE που η παράμετρός της είναι $v_j$.

Το Λήμμα 9.1 συνεπάγεται ότι η $gv(S)$ επιστρέφει εκτελώντας τη γραμμή 32 του ψευδοκώδικα. Υποθέτουμε ότι η $gv(S)$ έχει επιστρέψει αφότου έχει εκτελέσει $epoch$ επαναλήψεις του βρόχου while (γραμμή 15). Τότε υπάρχει κάποιος ακέραιος $l' \in \{1,\ldots,epoch\}$, τέτοιος ώστε για κάθε $j \in \{1,\ldots,m\}$ να ισχύει $SL_{l'}[j] \neq 1$ ή $history[l'][j].change \neq true$. Έστω $l$ ο μεγαλύτερος τέτοιος ακέραιος. Έστω $r_1$ η πρώτη εντολή read της $gv(S)$ στην $l$-οστή ομάδα αναγνώσεων και έστω $r_j$ η εντολή read της $gv(S)$ στον καταχωρητή $R_j$ (που αντιστοιχεί στη συνιστώσα $A_j$) στην $l$-οστή εποχή. Διακρίνουμε τις ακόλουθες περιπτώσεις.

1) Αρχικά υποθέτουμε ότι επιτελεί την write της $w_j$ έπειτα από την $r_1$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, το σημείο σειριοποίησης της $U_j$ έχει εισαχθεί ακριβώς πριν το σημείο σειριοποίησης της $gv(S)$ και καμία άλλη

UPDATE δεν έχει σειριοποιηθεί μεταξύ της $U_j$ και της $S$ (υπενθυμίζεται ότι όλες οι UPDATE έχουν σειριοποιηθεί ακριβώς πριν την $gv(S)$ επιτελούνται σε διαφορετικές συνιστώσες). Αυτό είναι άτοπο, καθώς η $U$ έχει σειριοποιηθεί μεταξύ της $U_j$ και της $S$.

2) Υποθέτουμε ότι η $w_j$ προηγείται της $r_1$. Υποθέτουμε αρχικά ότι η write της $U$ έπεται της $w_j$. Αφού η $gv(S)$ επιστρέφει την τιμή $v_j$ για τη συνιστώσα $A_j$, η τελευταία εντολή write στον καταχωρητή $R_j$ που προηγείται της $r_j$ είναι η $w_j$. Αφού η $w$ έπεται της $w_j$, τότε η $w$ θα έπεται της $r_j$. Αφού η $gv(S)$ έχει σειριοποιηθεί ακριβώς πριν την $r_1$, το σημείο σειριοποίησης της $gv(S)$ προηγείται της $w$. Αφού η $U$ σειριοποιείται ακριβώς πριν την $gv(S)$, προκύπτει ότι η $U$ δε σειριοποιείται στην $w$ εντολή της. Από τους ισχυρισμούς 1 και 2 του Λήμματος 9.6, υπάρχει μία SCAN $S'$, τέτοια ώστε η $S'$ να επέστρεψε $v$ και το σημείο σειριοποίησης της $U$ να βρίσκεται εντός του διαστήματος εκτέλεσης της $S'$. Επιπρόσθετα από τον ισχυρισμό 2 του Λήμματος 9.6 προκύπτει ότι η $w$ προηγείται του τέλους του διαστήματος εκτέλεσης της $S'$. Αφού η $S$ επιστρέφει $v_j$ και όχι $v$ για τη συνιστώσα $A_j$, ισχύει $S' \neq S$. Αν η $S'$ έπεται της $S$, τότε το σημείο σειριοποίησης της $U$ έπεται του τέλους του διαστήματος εκτέλεσης της $S$, το οποίο και είναι άτοπο. Αν η $S'$ προηγείται της $S$, η $w$ (που προηγείται του τέλους του διαστήματος εκτέλεσης της $S'$) προηγείται της $r_j$, το οποίο και είναι άτοπο.

3) Τέλος υποθέτουμε ότι η $w$ προηγείται της $w_j$. Από το Λήμμα 9.5 προκύπτει ότι η $U$ σειριοποιείται εντός του διαστήματος εκτέλεσής της. Έτσι το τελευταίο σημείο στο οποίο η $U$ δύναται να σειριοποιηθεί είναι στην εντολή write της $w$. Αφού η $w$ προηγείται της $w_j$ και η $U$ σειριοποιείται έπειτα από την $U_j$, προκύπτει ότι η $U_j$ δε σειριοποιείται στην write της. Το Λήμμα 9.6 συνεπάγεται ότι υπάρχει μία SCAN $S''$, τέτοια ώστε η $U_j$ να σειριοποιείται εντός του διαστήματος εκτέλεσης της $S''$. Αν ισχύει $S = S''$, από τον τρόπο ανάθεσης των σημείων σειριοποίησης, το σημείο σειριοποίησης της $U_j$ τοποθετείται ακριβώς πριν το σημείο σειριοποίησης της $S$, ενώ καμία άλλη UPDATE στη συνιστώσα $A_j$ δε δύναται να σειριοποιηθεί μεταξύ αυτών. Αφού η $U$ σειριοποιείται μεταξύ της $U_j$ και της $S$, τότε θα ισχύει $S \neq S''$.

Υποθέτουμε ότι η $S''$ έπεται της $S$. Τότε από τον ισχυρισμό 4 του Λήμματος 9.6 συνεπάγεται ότι υπάρχει μία UPDATE $U'$, της οποίας η εντολή write $w'$ έπεται της $w_j$ και προηγείται του τέλους του διαστήματος εκτέλεσης της $S''$. Προκύπτει ότι η $S$ δεν ανέγνωσε στον καταχωρητή $R_j$ την τιμή που έγγραψε η $U_j$. Έτσι η $S$ δεν επιστρέφει τιμή ίση με $v_j$ για τη συνιστώσα $A_j$, το οποίο και είναι άτοπο.

Υποθέτουμε ότι η $S''$ έπεται της $S$. Από τον τρόπο ανάθεσης των σημείων σειριοποίησης, η $U_j$ σειριοποιείται ακριβώς πριν την $gv(S'')$. Αφού η $S''$ σειριοποιείται στο ίδιο σημείο με αυτό της $gv(S'')$, η $U_j$ έχει σειριοποιηθεί ακριβώς πριν την $S''$. Από το Λήμμα 9.5 προκύπτει ότι η $S''$ σειριοποιείται εντός του διαστήματος εκτέλεσής της, έτσι η $U_j$ σειριοποιείται εντός του διαστήματος εκτέλεσης της $S''$. Αφού η $S$ επιστρέφει τιμή ίση με $v_j$, η $U_j$ επιτελεί την write της πριν το τέλος της $S$, και έτσι η $U_j$ τερματίζει πριν το τέλος της $S$. Έτσι το σημείο σειριοποίησης της $U_j$ δε δύναται να έπεται του διαστήματος εκτέλεσης της $S$, το οποίο και είναι άτοπο. ■

Έστω $S$ μία οποιοδήποτε SCAN. Η $S$ σειριοποιείται στο ίδιο σημείο με αυτό της $gv(S)$ και επιστρέφει το ίδιο διάνυσμα τιμών. Το Λήμμα 9.7 συνεπάγεται ότι η $gv(S)$ επιστρέφει ένα συνεπές διάνυσμα τιμών. Έτσι συμπεραίνουμε ότι η $S$ επιστρέφει ένα συνεπές διάνυσμα τιμών.

***Θεώρημα 9.8*** *Ο αλγόριθμος SweepLine είναι μία Single-Scanner υλοποίηση ατομικών στιγμιοτύπων πολλαπλής εγγραφής που επιτυγχάνει χρονική πολυπλοκότητα* $O(m^2)$ *για τη SCAN και την UPDATE χρησιμοποιώντας* $m + 1$ *κοινούς καταχωρητές μεγέθους* $O(m \cdot log(|T|) + log\, k + log\, n)$ *bit.*

# ΚΕΦΑΛΑΙΟ 10. ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ

Στην παρούσα εργασία παρουσιάστηκε μια μεγάλη συλλογή αποτελεσματικών υλοποιήσεων ατομικών στιγμιοτύπων. Στη συλλογή αυτή ανήκουν οι πρώτοι Single-Scanner αλγόριθμοι που επιτυγχάνουν χρονική πολυπλοκότητα που εξαρτάται μόνο από τον αριθμό των συνιστωσών του ατομικού στιγμιοτύπου $m$ και όχι από το πλήθος των διεργασιών $n$ (που στις περισσότερες εφαρμογές είναι πολύ μεγαλύτερο). Επίσης στη συλλογή αυτή ανήκουν και οι πρώτοι χρονικά βέλτιστοι Single-Scanner αλγόριθμοι.

Μια ενδιαφέρουσα μελλοντική ερευνητική κατεύθυνση είναι η μελέτη της ύπαρξης ενός αλγορίθμου που θα επιτυγχάνει την βέλτιστη χρονική πολυπλοκότητα του RT-Opt, ενώ παράλληλα θα μειώνει το πλήθος των καταχωρητών που ο RT-Opt χρησιμοποιεί από $O(mn)$ σε $O(n)$. Είναι επίσης ενδιαφέρον να μελετηθεί αν είναι εφικτό να σχεδιαστεί χρονικά βέλτιστος Single-Scanner αλγόριθμος που χρησιμοποιεί ακόμη λιγότερους από $O(n)$ καταχωρητές.

Θα ήταν ακόμη ενδιαφέρον να μελετηθεί αν μπορεί να αναπτυχθεί ένας γενικός αλγόριθμος ατομικών στιγμιοτύπων από CAS καταχωρητές που θα επιτυγχάνει τη χρονική πολυπλοκότητα του C-Snap και θα χρησιμοποιεί μόνο $m$ καταχωρητές, οι οποίοι ωστόσο θα είναι μικρότερου μεγέθους από αυτούς που χρησιμοποιεί ο C-Snap. Η ανάπτυξη ενός αλγορίθμου που θα επιτυγχάνει χρονική πολυπλοκότητα για τη SCAN μικρότερη από αυτή του C-Snap χωρίς να επηρεάζεται η χρονική πολυπλοκότητα της UPDATE είναι ένα ακόμη σημαντικό ανοικτό ερευνητικό πρόβλημα.

Τέλος, θα ήταν ενδιαφέρουσα η ανάπτυξη ενός Single-Scanner αλγορίθμου που θα είναι βέλτιστος ως προς τη χωρική και χρονική του πολυπλοκότητα. Ένας τέτοιος αλγόριθμος θα επιτύγχανε χρονική πολυπλοκότητα $O(m^2)$ για τη SCAN χρησιμοποιώντας μόνο $m$ καταχωρητές. Ο SweepLine επιτυγχάνει αυτή τη χρονική πολυπλοκότη-

τα για τη SCAN αλλά δεν είναι χωρικά βέλτιστος αφού χρησιμοποιεί έναν παραπάνω καταχωρητή.

# ΑΝΑΦΟΡΕΣ

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. Atomic snapshots of shared memory. *In Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing.* pp. 1-14, August 1990.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp and D. Touitou. Long-lived renaming made adaptive. *In procceedings of the 18th ACM Symposium on Principles of Distributed Computing.* pp. 91-103, 1999.

[3] Marcos K. Aguilera, Susan Spence and Alistair Veitch. Olive: distributed point-in-time branching storage for real systems. *In proceedings of 3rd Symposium on Networked Systems Design & Implementation.* pp. 367-380, 2006.

[4] J. H. Anderson. Composite registers. *In Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing.* pp. 15-29, August 1990.

[5] J. H. Anderson. Multi-writer composite registers. *Distributed Computing.* Vol. 7(4), pp. 175-195, April 1993.

[6] J. Aspnes. Time and space-efficient randomized consensus. *Journal of Algorithms.* Vol. 14(3), pp. 414-431, May 1993.

[7] J. Aspnes and M. Herlinhy. Wait-free data structures in the asynchronous PRAM model. *In Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures.* pp. 340-349, 1990.

[8] H. Attiya, F. Ellen and P. Fatourou. The Complexity of Updating Multi-Writer Snapshot Object. *In Proceedings of 8th International Conference on Distributed Computing and Networking.* pp. 319-349, December 2006.

[9] H. Attiya and A. Fouren. Adaptive and Efficient Algorithms for lattice agreement and renaming. *SICOMP.* Vol. 31(2), pp. 642-664, December 2001.

[10] H. Attiya, M. Herlihy and O. Rachman. Atomic snapshots using lattice agreement. *Distributed Computing.* Vol. 8, pp. 121-132, 1995.

[11] H. Attiya, N. Lynch and N. Shavit. Are wait-free algorithms fast? *Journal of the ACM.* Vol. 41(4), pp. 724-763, 1994.

[12] H. Attiya and O. Rachman. Atomic snapshots in O(nlogn) operations. *SIAM Journal of Computing.* Vol. 27(2), pp. 319-340, 1998.

[13] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edition.* Wiley, 2004.

[14] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems.* February 1985.

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

[16] P. Fatourou, F. Fich and E. Ruppert. Space-Optimal multi-writer snapshot objects are slow! *In Proceedings of the 21th ACM Symposium on Principles of Distributed Computing.* pp. 13-20, 2002.

[17] P. Fatourou, F. Fich and E. Ruppert. Time-space tradeoffs for implementations of snapshots. *In Proceedings of the 38th ACM Symposium on Theory of Computing.* 2006.

[18] P. Fatourou and N. D. Kallimanis. Single-Scanner Multi-Writer Snapshot Implementations are Fast! *In Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* pp. 228-237, July 2006.

[19] P. Fatourou and N. D. Kallimanis. Time Optimal, Space-efficient Single-Scanner Snapshots & Efficient Multi-Scanner Snapshots using CAS. DCS 2007-02, Department of Computer Science, University of Ioannina, February 2007.

[20] P. Fatourou and N. D. Kallimanis. Time Optimal, Space-efficient Single-Scanner Snapshots & Multi-Scanner Snapshots using CAS. *In Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* pp. 33-42, August 2007.

[21] R. Gawlick, N. Lynch and N Shavit. Concurrent timestamping made simple. *In Proceedings of the Israel Symposium on the Theory of Computing and Systems.* pp. 171-183, July 1992.

[22] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems.* Vol. 12(3), pp. 463-492, July 1990.

[23] M. Inoue, W. Chen, T. Masuzawa and N. Tokura. Linear Time snapshots using multi-writer multi-reader registers. *In 8th International workshop on Distributed Algorithms, LLNCS # 857.* pp. 130-140, 1994.

[24] A. Israeli, A. Shaham and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory.* Vol. 28(5), pp. 469-468, September/October 1995.

[25] P. Jayanti. An Optimal Multi-Writer Snapshot Algorithm. *In Proceedings of the 37th ACM Symposium on Theory of Computing.* pp. 723-732, May 2005.

[26] P. Jayanti. f-arrays: implementation and applications. *In Proceedings of the 21th ACM Symposium on Principles of Distributed Computing.* pp. 270-279, 2002.

[27] P. Jayanti and S. Petrovic. Effcient wait-free implementation of multiword ll/sc variables. *In Proceedings of the 25th International Conference on Distributed Computing Systems.* pp. 59-68, 2005.

[28] P. Jayanti, K. Tan and S. Toueg. Time and Space Lower Bounds for non-blocking implementations. *SICOMP.* Vol. 30(2), pp. 438-456, June 2000.

[29] L. M. Kirousis, P. Spirakis and P Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. *IEEE Transactions Parallel and Distributed Systems.* Vol. 5(7), pp. 688-696, 1994.

[30] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[31] S. Mullender. *Distributed Systems.* Addison-Wesley, 1994.

[32] Y. Riany, N. Shavit and D. Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science.* Vols. 269(1-2), pp. 163-201, 2001.

# ΔΗΜΟΣΙΕΥΣΕΙΣ ΚΑΙ ΤΕΧΝΙΚΕΣ ΑΝΑΦΟΡΕΣ ΣΥΓΓΡΑΦΕΑ

1) P. Fatourou and N. D. Kallimanis. Time Optimal, Space-efficient Single-Scanner Snapshots & Multi-Scanner Snapshots using CAS. *In Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. pp. 33-42, August 2007.
2) P. Fatourou and N. D. Kallimanis. Single-Scanner Multi-Writer Snapshot Implementations are Fast! *In Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. pp. 228-237, July 2006.
3) P. Fatourou and N. D. Kallimanis. Time Optimal, Space-efficient Single-Scanner Snapshots & Efficient Multi-Scanner Snapshots using CAS. DCS 2007-02, Department of Computer Science, University of Ioannina, February 2007.

# ΒΙΟΓΡΑΦΙΚΟ

Ο Νικόλαος Καλλιμάνης γεννήθηκε το 1983 στην Αμαλιάδα Ηλείας, όπου και ολοκλήρωσε τις λυκειακές του σπουδές το 2001. Το 2001 ξεκίνησε τις σπουδές του στο Τμήμα Πληροφορικής του Πανεπιστημίου Ιωαννίνων, τις οποίες και ολοκλήρωσε το 2005. Από τον Σεπτέμβριο του 2005 είναι μεταπτυχιακός φοιτητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων. Τα ερευνητικά του ενδιαφέροντα εστιάζονται στον χώρο του κατανεμημένου υπολογισμού και ειδικότερα σε προβλήματα που αφορούν συστήματα κοινής μνήμης.