ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αποδοτικές λειτουργίες αναζήτησης εύρους σε B+-Δέντρα με χρήση RCU-HTM

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Ι. Μπίλλης

Επιβλέπων: Γεώργιος Ι. Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αποδοτικές λειτουργίες αναζήτησης εύρους σε B+-Δέντρα με χρήση RCU-HTM

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Ι. Μπίλλης

Επιβλέπων: Γεώργιος Ι. Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Φεβρουαρίου 2020.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2020

.....

Παναγιώτης Ι. Μπίλλης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Τα πολυπύρηνα υπολογιστικά συστήματα δε συνιστώνται πλέον μόνο για επιστημονικές εφαρμογές, αλλά αποτελούν συχνή επιλογή για την κάλυψη ποικίλων υπολογιστικών αναγκών. Για την αξιοποίηση των συστημάτων αυτών είναι απαραίτητη η ύπαρξη κατάλληλων παράλληλων αλγορίθμων και δομών δεδομένων, που θα εκμεταλλεύονται στο μέγιστο τους διαθέσιμους πόρους, θα κλιμακώνουν αποδοτικά και θα εγγυόνται τη συνέπεια των αποτελεσμάτων.

Κύριος σκοπός της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη μίας παράλληλης δομής δεδομένων που θα υποστηρίζει αποδοτικές λειτουργίες αναζήτησης εύρους. Για να φτάσουμε στο σημείο αυτό, ξεκινάμε παραθέτοντας το θεωρητικό υπόβαθρο που θα μας χρειαστεί, αναλύοντας ιδιαίτερα τα στοιχεία που πρόκειται να χρησιμοποιήσουμε. Στη συνέχεια αναφερόμαστε σε ήδη υπάρχουσες υλοποιήσεις, βλέποντας έτσι τι πρέπει να προσέξουμε για τη δημιουργία ενός αποδοτικού αλγορίθμου. Ακολούθως, αναλύουμε την τεχνική RCU-HTM, η οποία συνδυάζει την τεχνική Read-Copy-Update με Transactional Memory. Έπειτα, την εφαρμόζουμε σε ένα B+ Δέντρο, δημιουργώντας έτσι μία αποδοτική παράλληλη δομή δεδομένων που προσφέρει ιδιαίτερα αποδοτικές λειτουργίες αναζήτησης εύρους.

Τέλος, αξιολογούμε πειραματικά τον αλγόριθμο που αναπτύξαμε, συγκρίνοντας την απόδοση του με ορισμένες από τις ήδη υπάρχουσες υλοποιήσεις. Τα αποτελέσματα δείχνουν ότι η τεχνική RCU-HTM εφαρμοσμένη σε ένα B+ Δέντρο δίνει μία πολύ αποδοτική υλοποίηση για αρκετά διαφορετικά σενάρια μετρήσεων, στην πλειονότητα των οποίων υπερτερεί των ανταγωνιστών της. Σε ορισμένα σενάρια μάλιστα, η απόδοση της είναι έως και τέσσερις φορές μεγαλύτερη αυτής των υπόλοιπων υλοποιήσεων. Κλείνοντας, οδηγούμαστε σε ενδιαφέροντα συμπεράσματα και προτείνουμε ορισμένες μελλοντικές επεκτάσεις που προκύπτουν από την παρούσα διπλωματική εργασία.

Λέξεις-Κλειδιά: παράλληλος προγραμματισμός, παράλληλες δομές δεδομένων, κλιμακωσιμότητα, B+ Δέντρα, παράλληλες λειτουργίες αναζήτησης εύρους, RCU-HTM, Read-Copy-Update, Hardware Transactional Memory

Abstract

Nowadays, multicore computing systems are not only used for scientific applications, but also serve as a rather common solution for many computing needs. In order to exploit to the maximum these systems, we have to design concurrent algorithms and data structures, which will take full advantage of the available resources, scale efficiently and guarantee consistency.

The major goal of this thesis is the deployment of a concurrent data structure, which supports efficient range queries. To get at this point, we start by studying the corresponding theoretical background and analyzing especially those elements, which will help us design our algorithm. Afterwards we refer to already existent implementations, which let us understand what should be taken into consideration, while designing our algorithm. In the next part we analyze the RCU-HTM technique, which combines Read-Copy-Update with Transactional Memory, and apply it to B+ Tree. This results in an efficient concurrent data structure, which supports particularly efficient range queries.

Subsequently, we evaluate our algorithm by comparing it with some of the already existent implementations. Results show that RCU-HTM, when applied to B+ Tree, offers an implementation, which copes efficiently in many different evaluation scenarios. In the great majority of those scenarios RCU-HTM outperforms its competitors. Finally, we are driven to interesting conclusions and make proposals for future work, which arise from this thesis.

Keywords: parallel programming, concurrent data structures, scalability, B+ Tree, concurrent range queries, RCU-HTM, Read-Copy-Update, Hardware Transactional Memory

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του επίκουρου καθηγητή Γεώργιου Ι. Γκούμα.

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Γκούμα τόσο για τις γνώσεις και την έμπνευση που μου προσέφερε μέσα από τη διδασκαλία του, όσο και για τη δυνατότητα που μου έδωσε να ασχοληθώ με την παρούσα διπλωματική εργασία.

Ιδιαζόντως θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα Δημήτριο Σιακαβάρα για τη βοήθειά που απλόχερα μου παρείχε σε όλα τα στάδια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και για το χρόνο που αφιέρωσε. Χωρίς την πολύτιμη συμβολή του η ολοκλήρωση της παρούσης δε θα ήταν εφικτή.

Επιπλέον, οφείλω να ευχαριστήσω τους φίλους μου, οι οποίοι ήταν δίπλα μου όποτε χρειάστηκε, προσφέροντας μου την απαραίτητη ψυχολογική υποστήριξη.

Τέλος, δεν μπορώ παρά να ευχαριστήσω θερμά την οικογένειά μου για την υποστήριξη και την εμπιστοσύνη που μου έχει δείξει όλα αυτά τα χρόνια.

Παναγιώτης Μπίλλης,
Αθήνα, 4 Φεβρουαρίου 2020

Πίνακας περιεχομένων

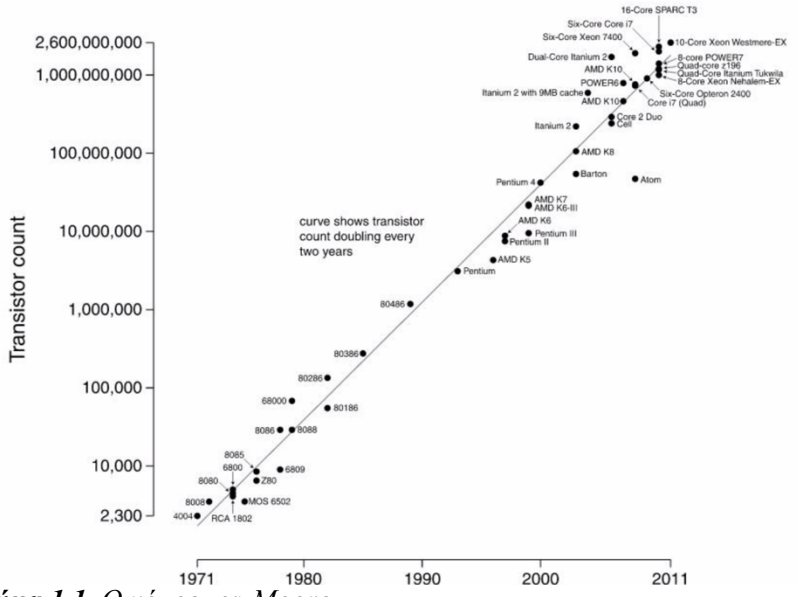
Πίνακας σχημάτων

Κεφάλαιο 1
Εισαγωγή

1.1 Γενικά

Ο Gordon Moore προέβλεψε το 1965 πως ο αριθμός των τρανζίστορς ενός πυκνού ολοκληρωμένου κυκλώματος θα διπλασιάζεται ανά χρόνο[1]. Μία δεκαετία αργότερα, το 1975, επανεξετάζοντας τα δεδομένα, ο Moore αναθεώρησε τη πρόβλεψή του, θέτοντας ως απαιτούμενο διάστημα τα δύο χρόνια. Η διατύπωση αυτή είναι γνωστή ως νόμος του Moore. Ο νόμος του Moore επαληθεύτηκε στην πράξη, καθώς έκτοτε ο αριθμός των τρανζίστορς ενός ολοκληρωμένου κυκλώματος διπλασιάζεται ανά δύο χρόνια.

Συχνά χρονικό διάστημα για την επιβεβαίωση του θεωρούνται οι 18 μήνες, καθώς η απόδοση των μικροεπεξεργαστών, όπως παρατήρησε ο David House, διπλασιάζεται μετά το πέρας 18 μηνών τόσο λόγω της αύξησης του αριθμού των τρανζίστορς από τα οποία αποτελούνται, όσο και λόγω της αύξησης της ταχύτητάς τους.



Σχήμα 1.1. *Ο νόμος του Moore*

Η συνεχής αύξηση όμως του αριθμού των τρανζίστορς ανά επεξεργαστή συναντά φυσικούς περιορισμούς. Η συνύπαρξη όλο και περισσότερων τρανζίστορς σ' έναν μικροεπεξεργαστή οδηγεί στην κατανάλωση περισσότερης ενέργειας και άρα στην παραγωγή περισσότερης θερμότητας. Οι δυσκολίες στην κατασκευή ημιαγωγών κατάλληλου μεγέθους για την επιβεβαίωση του νόμου του Moore ξεκίνησαν το 2012, ενώ το 2015 η Intel επιβεβαίωσε τις προβλέψεις που ήθελαν το ρυθμό επιβεβαίωσης του νόμου του Moore να επιβραδύνεται.

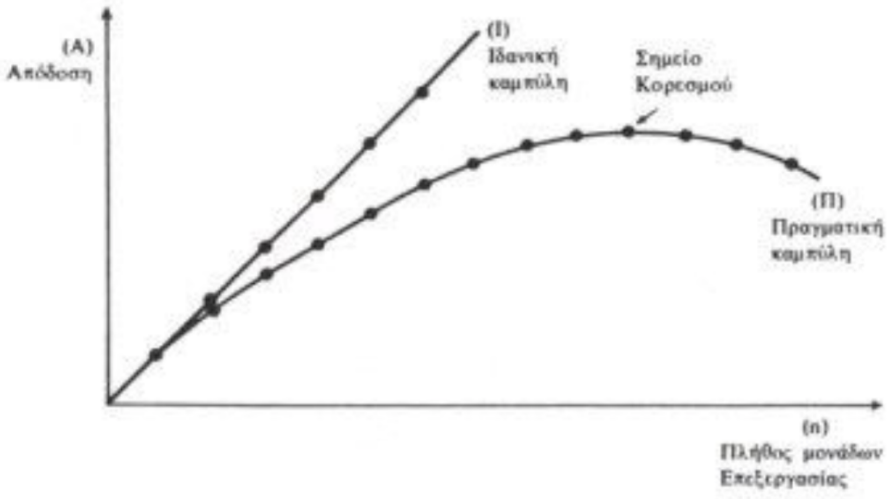
Εκτός από την αύξηση των τρανζίστορς ανά μικροεπεξεργαστή, καλύτερη απόδοση έχουμε και με τη χρήση παραλληλισμού σε επίπεδο εντολών (Instruction-level Parallelism – ILP). Στο pipelining για παράδειγμα μπορούμε να έχουμε πολλές εντολές που

εκτελούνται ταυτόχρονα, εφόσον η κάθε εντολή βρίσκεται σε διαφορετικό στάδιο εκτέλεσης. Έτσι επιτυγχάνεται καλύτερη απόδοση. Είναι όμως σαφές ότι οι τεχνικές αυτές μπορούν να βελτιώσουν την απόδοση μόνο έως ένα βαθμό.

Βλέπουμε άρα ότι στην αναζήτηση μεγαλύτερης απόδοσης πρέπει να στραφούμε σε άλλες λύσεις. Μία τέτοια λύση αποτελούν οι πολυπύρηνοι επεξεργαστές. Μέχρι πρότινος, τα υπολογιστικά συστήματα στήριζαν όλη τους την απόδοση στην ύπαρξη μίας ισχυρής κεντρικής μονάδας επεξεργασίας (ΚΜΕ) που εκτελούσε σειριακά τις εντολές του προγράμματος. Πλέον οι πολυπύρηνοι επεξεργαστές αποτελούν βασικό συστατικό όχι μόνο υπερυπολογιστών, αλλά ακόμα και προσωπικών υπολογιστών. Ιδιαίτερα σε επιστημονικές εφαρμογές, αλλά και εφαρμογές σε βάσεις δεδομένων, η χρήση πολυπύρηνων επεξεργαστών αποδεικνύεται πολύ αποδοτική.

Η απόδοση ενός υπολογιστικού συστήματος μπορεί να μετρηθεί με διάφορους δείκτες. Ο πλέον διαδεδομένος δείκτης απόδοσης είναι το πλήθος των εκτελούμενων πράξεων κινητής υποδιαστολής ανά δευτερόλεπτο, FLOP (Floating – point Operations Per Second). Ένας άλλος δείκτης απόδοσης είναι το πλήθος των εντολών που εκτελούνται ανά μονάδα του χρόνου, όπως η μονάδα MIPS (Million Instructions Per Second), δηλαδή 10^6 εντολές ανά δευτερόλεπτο.

Στο διάγραμμα του σχήματος 1.2 φαίνονται η ιδανική και η πραγματική απόδοση ενός παράλληλου υπολογιστή σε συνάρτηση με τον αριθμό των μονάδων επεξεργασίας.



Σχήμα 1.2. *Η απόδοση ενός παράλληλου υπολογιστή, ως συνάρτηση του πλήθους των μονάδων επεξεργασίας.*

1.2 Παράλληλος προγραμματισμός

Είναι φανερό ότι με την αξιοποίηση των πολυπύρηνων συστημάτων οι κλασικοί σειριακοί αλγόριθμοι δεν μπορούν να προσφέρουν βέλτιστη απόδοση. Για τον σκοπό αυτό οι αλγόριθμοι πρέπει να επανασχεδιαστούν ώστε να εκμεταλλεύονται στο μέγιστο την παραλληλοποίηση που προσφέρουν οι πολυπύρηνες αρχιτεκτονικές. Ο παράλληλος

προγραμματισμός έχει επομένως ως σκοπό την αποτελεσματική χρήση των υπολογιστών παράλληλης επεξεργασίας.

1.2.1 Μέτρηση απόδοσης

Ακόμα είναι σημαντικό να ορίσουμε κάποιες παραμέτρους που μας βοηθούν στη μέτρηση της απόδοσης ενός παράλληλου προγράμματος:

- **Χρόνος:** ως χρόνος ενός παράλληλου αλγορίθμου ορίζεται η μέγιστη χρονική διάρκεια από την εκκίνηση ενός αλγορίθμου στον πρώτο επεξεργαστή που θα αρχίσει, ως τον τερματισμό του στον τελευταίο επεξεργαστή που θα ολοκληρώσει την εκτέλεσή του.
- **Επιτάχυνση (S):** ως επιτάχυνση ενός παράλληλου αλγορίθμου που εκτελείται σε **p** επεξεργαστές ορίζεται ο λόγος του χρόνου του ταχύτερου ακολουθιακού αλγορίθμου προς το χρόνο εκτέλεσης του παράλληλου αλγορίθμου στους **p** επεξεργαστές και έχει ιδανική τιμή **p**.

$$Speedup(S) = \frac{T_s}{T_p}$$

- **Αποτελεσματικότητα (E):** ως αποτελεσματικότητα ενός παράλληλου αλγορίθμου που εκτελείται σε **p** επεξεργαστές ορίζεται ο λόγος της επιτάχυνσης **S** προς το πλήθος των επεξεργαστών **p** και έχει ιδανική τιμή 1.
- **Κλιμακωσιμότητα:** η κλιμακωσιμότητα εκφράζει ποιοτικά την ικανότητα ενός προγράμματος να βελτιώνει την επίδοσή του με την προσθήκη επιπλέον επεξεργαστών.
- **Συνάρτηση επεκτασιμότητας f(σp):** η συνάρτηση επεκτασιμότητας **f(σp)**, (όπου **σ** σταθερά), δίνει το χρόνο εκτέλεσης του αλγορίθμου για διαφορετικά πλήθη στοιχείων εισόδου, καθώς μεταβάλλεται το πλήθος των επεξεργαστών ανάλογα με τη μεταβολή του πλήθους των στοιχείων εισόδου. Ιδανικά λοιπόν η συνάρτηση αυτή μετρά τη δυνατότητα του αλγορίθμου να κάνει χρήση περισσότερων επεξεργαστών εάν αυξηθεί το μέγεθος του προβλήματος.

1.2.2 Παράγοντες απόδοσης – Νόμος του Amdahl

Σε αυτό το σημείο είναι σημαντικό να αναφερθούμε σε παράγοντες που μπορούν να επηρεάσουν την απόδοση ενός παράλληλου αλγορίθμου.

Ένας σημαντικός παράγοντας που επηρεάζει την απόδοση ενός παράλληλου αλγορίθμου προκύπτει από την ανάγκη ανταλλαγής πληροφοριών μεταξύ των επεξεργαστών, καθώς και από την ανάγκη συγχρονισμού μεταξύ τους σε αλγορίθμους που ο συγχρονισμός είναι απαραίτητος ώστε να εκτελείται ορθά ο αλγόριθμος. Έτσι έχουμε το κόστος επικοινωνίας.

Η ανάγκη της επικοινωνίας διαφέρει σε κάθε αλγόριθμο. Γι' αυτό άλλωστε οι πιο αποδοτικοί παράλληλοι αλγόριθμοι προσπαθούν να περιορίσουν την εξάρτηση μεταξύ των διεργασιών. Αυτό γίνεται πολλές φορές κάνοντας επιπλέον υπολογισμούς, το οποίο είναι συχνά αποδοτικότερο από την επικοινωνία μεταξύ διεργασιών. Το κόστος επικοινωνίας εξαρτάται και από το είδος της παράλληλης μηχανής. Για παράδειγμα, η επικοινωνία σε μηχανές κοινής μνήμης είναι ταχύτερη από την επικοινωνία με ανταλλαγή μηνυμάτων.

Τα συστήματα παράλληλης επεξεργασίας συχνά περιλαμβάνουν κοινά αγαθά των οποίων η χρήση δεν μπορεί να γίνεται ταυτόχρονα από περισσότερους του ενός επεξεργαστές. Παράδειγμα αποτελούν η κοινή μνήμη, αλλά και οι συσκευές εισόδου/εξόδου. Σε αυτές τις περιπτώσεις απαιτείται συγχρονισμός μεταξύ των

επεξεργαστών ώστε να διασφαλίζεται ότι μόνο ένας από αυτούς θα μπορεί να χρησιμοποιεί το κοινό αγαθό ενώ οι υπόλοιποι θα πρέπει να περιμένουν. Αυτό ονομάζεται αμοιβαίος αποκλεισμός (mutual exclusion).

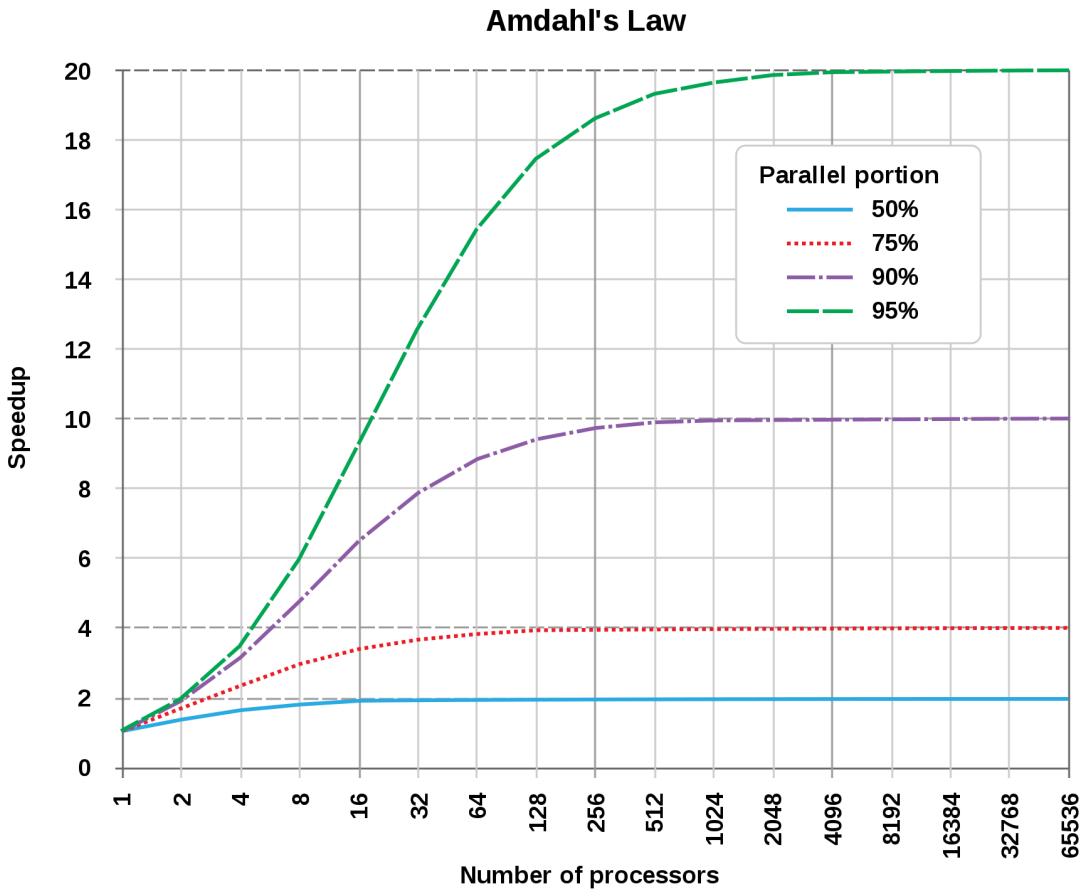
Ένας ακόμα σημαντικός παράγοντας που επηρεάζει την απόδοση ενός παράλληλου αλγορίθμου είναι το ποσοστό αυτού που πρέπει εκ φύσεως να εκτελεστεί ακολουθιακά. Έστω ότι ένας αλγόριθμος χρειάζεται 1 μονάδα χρόνου για να εκτελεστεί σειριακά και ένα ποσοστό του f ($0 \leq f \leq 1$) το οποίο δεν μπορεί να παραλληλοποιηθεί. Στην περίπτωση αυτή ο χρόνος εκτέλεσης του παράλληλου προγράμματος θα είναι:

$$T_p = fT_s + \frac{1-f}{p}T_s$$

και άρα η μέγιστη τιμή της επιτάχυνσης θα είναι:

$$S = \frac{1}{f + \frac{1-f}{p}}$$

Η έκφραση αυτή είναι γνωστή ως νόμος του Amdahl[2] και αποτελεί έναν από τους βασικότερους περιορισμούς της παράλληλης επεξεργασίας. Ο νόμος του Amdahl χρησιμοποιείται συχνά για να προβλέψει τη θεωρητική επιτάχυνση που μπορεί να επιτευχθεί όταν χρησιμοποιούνται περισσότεροι του ενός επεξεργαστές. Για παράδειγμα, αν το 10% ενός αλγορίθμου δεν μπορεί να παραλληλοποιηθεί, τότε η μέγιστη τιμή της επιτάχυνσης δεν μπορεί να είναι μεγαλύτερη από 10. Στο διάγραμμα του σχήματος 1.3 βλέπουμε τη θεωρητική επιτάχυνση για διαφορετικά ποσοστά παραλληλοποιήσιμου κώδικα, καθώς αυξάνεται ο αριθμός των επεξεργαστών που συνεργάζονται για την εκτέλεση του αλγορίθμου.

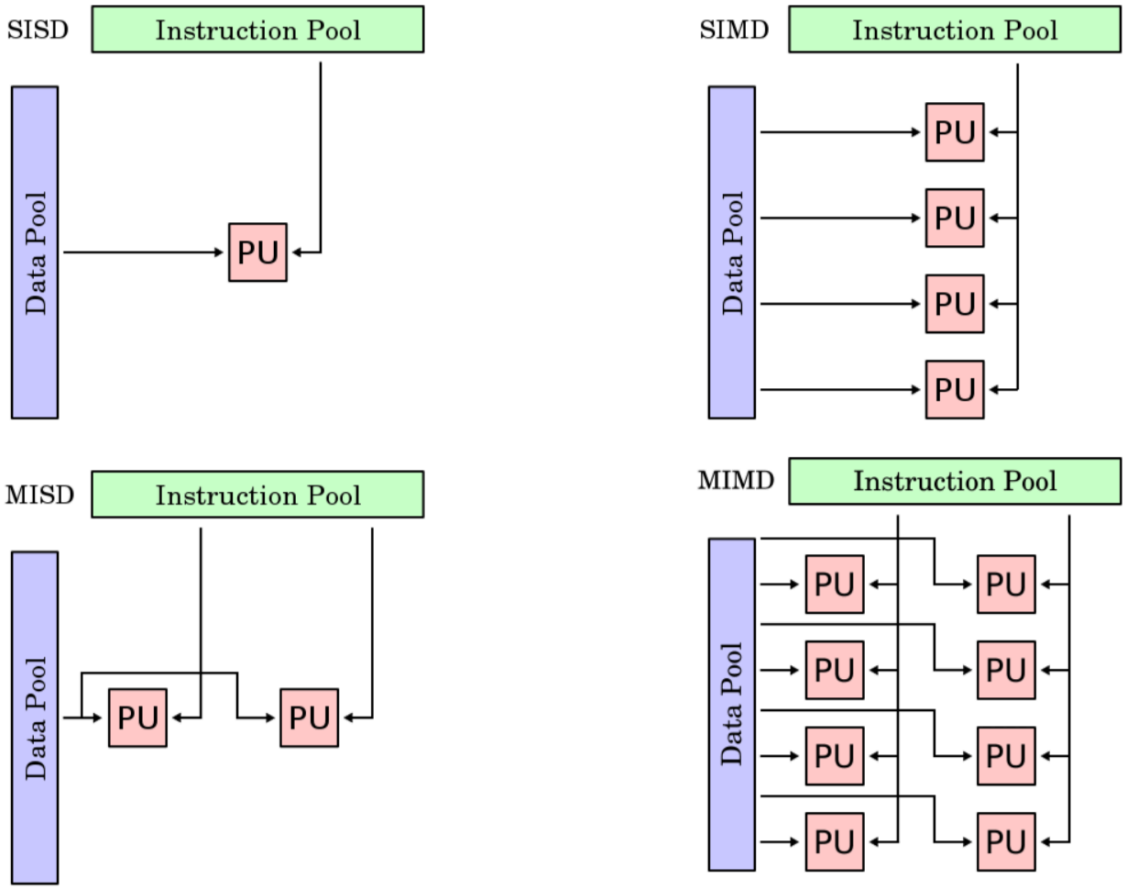


Σχήμα 1.3. Η θεωρητική επιτάχυνση ενός προγράμματος συναρτήσει του ποσοστού του παραλληλοποιήσιμου κώδικα και του αριθμού των επεξεργαστών βάσει του νόμου του Amdahl

1.3 Παράλληλες αρχιτεκτονικές

Σύμφωνα με την ταξινόμηση του Flynn[3] (σχ. 1.4) οι αρχιτεκτονικές υπολογιστών διακρίνονται βάσει του παραλληλισμού που χρησιμοποιούν για επεξεργασία εντολών και δεδομένων ως εξής:

- **SISD:** Single Instruction Single Data
  Ένας σειριακός υπολογιστής που δε χρησιμοποιεί παραλληλισμό ούτε στην επεξεργασία των εντολών, ούτε των ροών δεδομένων.
- **SIMD:** Single Instruction Multiple Data
  Ένας παράλληλος υπολογιστής, όπου μία εντολή μπορεί να εκτελείται σε πολλαπλά δεδομένα.
- **MISD:** Multiple Instruction Single Data
  Στην περίπτωση αυτή έχουμε πολλαπλές ροές εντολών που εκτελούνται στα ίδια δεδομένα.
- **MIMD:** Multiple Instruction Multiple Data
  Ένας παράλληλος υπολογιστής στον οποίο πολλαπλοί αυτόνομοι επεξεργαστές εκτελούν ταυτόχρονα διαφορετικές εντολές σε διαφορετικά δεδομένα. Σε αυτήν την αρχιτεκτονική ανήκουν τα clusters και τα συστήματα πολυπύρηνων αρχιτεκτονικών.



Σχήμα 1.4. *Η ταξινόμηση του Flynn.*

Οι παράλληλες αρχιτεκτονικές μπορούν ακόμα να κατηγοριοποιηθούν, σύμφωνα με την οργάνωση της μνήμης τους σε τρεις κατηγορίες:

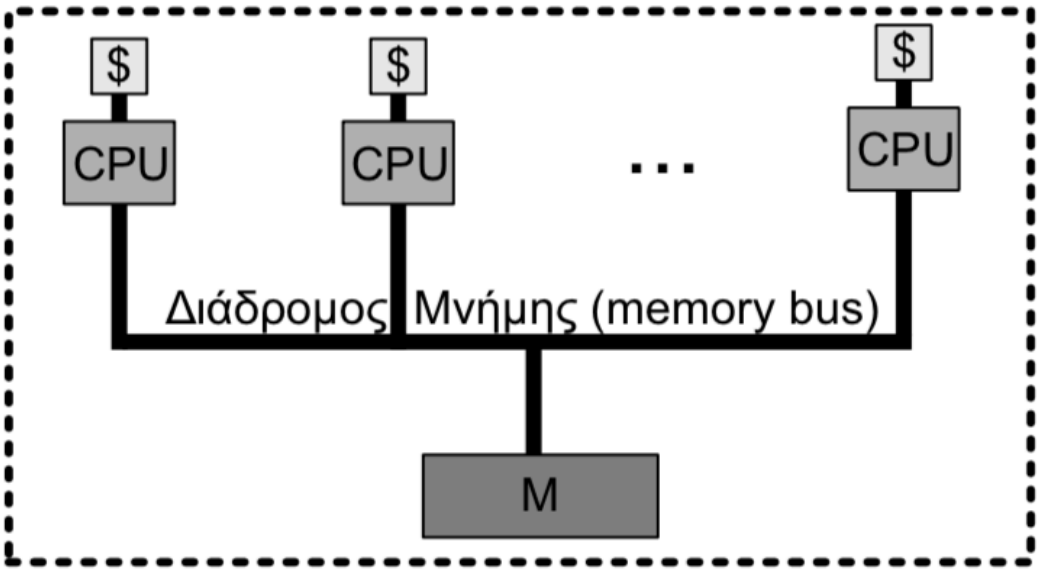
- Κοινής μνήμης (shared memory)
    - UMA (Uniform memory access): ο χρόνος προσπέλασης είναι ανεξάρτητος του επεξεργαστή και της θέσης μνήμης.
    - NUMA (Non-uniform memory access): ο χρόνος προσπέλασης εξαρτάται από τον επεξεργαστή και τη θέση μνήμης.
    - cc-NUMA (cache-coherent NUMA): NUMA με συνάφεια κρυφής μνήμης
- Κατανεμημένης μνήμης (distributed memory)
- Υβριδική

1.3.1 Αρχιτεκτονική κοινής μνήμης

Στις αρχιτεκτονικές κοινής μνήμης οι επεξεργαστές έχουν κοινή μνήμη, ενώ κάθε επεξεργαστής διαθέτει τοπική ιεραρχία κρυφών μνημών και έτσι απαιτείται η υλοποίηση πρωτοκόλλου συνάφειας μνήμης ώστε να διατηρηθεί η συνάφεια των δεδομένων στην κρυφή μνήμη. Η πρόσβαση σε όλα τα δεδομένα γίνεται με εντολές ανάγνωσης και εγγραφής στη μνήμη, ενώ συνήθως τα συστήματα διαθέτουν ατομικές εντολές που διευκολύνουν το συγχρονισμό. Συνήθως η διασύνδεση γίνεται μέσω διαδρόμου μνήμης (memory bus), αλλά και πιο εξελιγμένων δικτύων διασύνδεσης. Στην εικόνα του σχήματος

1.5 φαίνεται η οργάνωση μνήμης σε μία αρχιτεκτονική κοινής μνήμης με συμμετρική οργάνωση μνήμης.

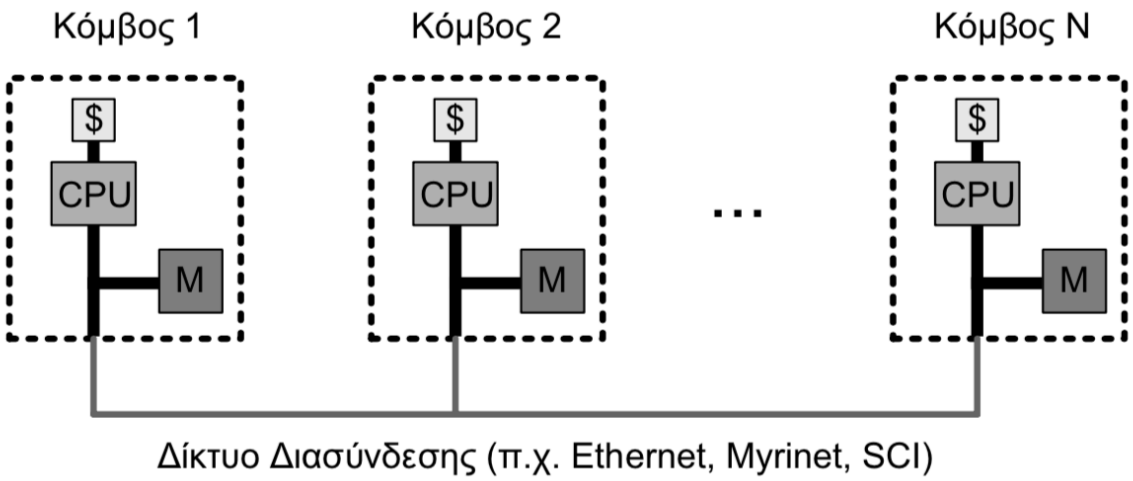
Στην αρχιτεκτονική κοινής μνήμης η εύκολη πρόσβαση στα κοινά δεδομένα διευκολύνει τον παράλληλο προγραμματισμό. Παρ' όλα αυτά, καθώς οι επεξεργαστές μπορούν να χρησιμοποιήσουν κοινά δεδομένα παράλληλα, υπάρχει το ενδεχόμενο καταστάσεων ανταγωνισμού και έτσι είναι απαραίτητος ο συγχρονισμός μεταξύ των επεξεργαστών ώστε να αποφευχθούν παράλληλες προσβάσεις σε κοινά δεδομένα. Ακόμα είναι σημαντικό να τονίσουμε πως πρόκειται για μία δύσκολα κλιμακώσιμη αρχιτεκτονική για πάνω από λίγες δεκάδες κόμβους λόγω του κοινού διαύλου επικοινωνίας.



Σχήμα 1.5. *Η αρχιτεκτονική κοινής μνήμης.*

1.3.2 Αρχιτεκτονική κατανεμημένης μνήμης

Στην αρχιτεκτονική κατανεμημένης μνήμης κάθε επεξεργαστής έχει δική του τοπική μνήμη και ιεραρχία τοπικών μνημών και λέγεται κόμβος. Κάθε κόμβος διασυνδέεται με τους υπόλοιπους μέσω ενός δικτύου διασύνδεσης (π.χ. Ethernet, Myrinet, SCI). Η πρόσβαση σε δεδομένα που βρίσκονται σε απομακρυσμένους κόμβους γίνεται ρητά μέσω κλήσεων επικοινωνίας, ανταλλαγής μηνυμάτων (send / receive) ή μέσω συνεννόησης των δύο πλευρών για πρόσβαση στην απομακρυσμένη μνήμη. Στην εικόνα του σχήματος 1.6 φαίνεται η οργάνωση μίας τυπικής αρχιτεκτονικής κατανεμημένης μνήμης.

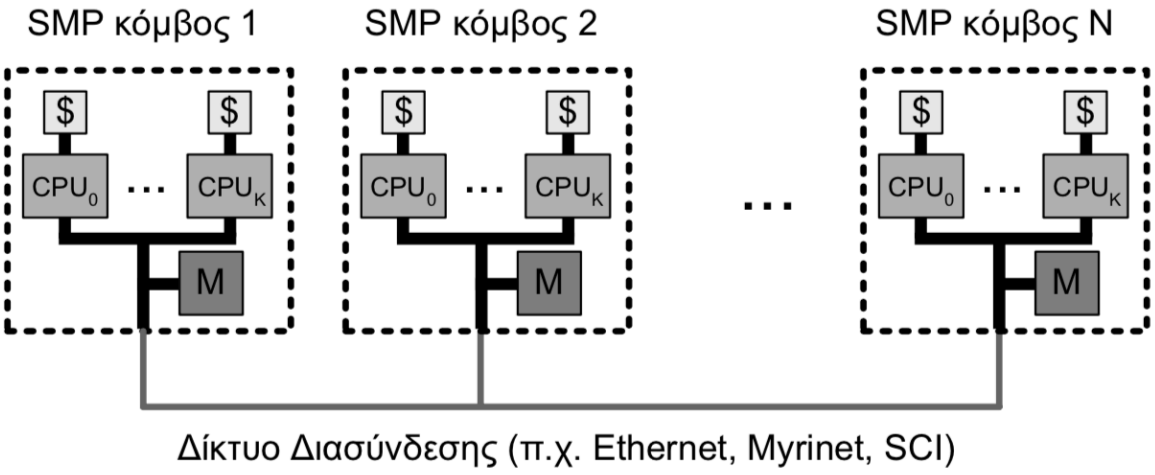


Σχήμα 1.6. *Η αρχιτεκτονική κατανεμημένης μνήμης.*

Η κατανεμημένη μνήμη καθιστά δύσκολο τον προγραμματισμό, καθώς ο προγραμματιστής απαιτείται να σχεδιάσει και να υλοποιήσει την πρόσβαση σε διακριτές μνήμες (κατακερματισμένος προγραμματισμός). Παρ' όλη τη δυσκολία όμως, η αρχιτεκτονική κατανεμημένης μνήμης προσφέρει μεγάλη κλιμακωσιμότητα, καθώς κλιμακώνει σε χιλιάδες υπολογιστικούς κόμβους.

1.3.3 Υβριδική αρχιτεκτονική

Η υβριδική αρχιτεκτονική συνδυάζει τις δύο προηγούμενες αρχιτεκτονικές. Όπως φαίνεται και στο σχήμα 1.7 πρόκειται για κόμβους με αρχιτεκτονική κοινής μνήμης που διασυνδέονται μέσω ενός δικτύου διασύνδεσης σε αρχιτεκτονική κατανεμημένης μνήμης. Πρόκειται για την τυπική αρχιτεκτονική των σύγχρονων συστοιχιών-υπερυπολογιστών, των data centers και των υποδομών cloud.



Σχήμα 1.7. *Η αρχιτεκτονική κατανεμημένης μνήμης.*

1.4 Συγχρονισμός

Όπως έχουμε δει μέχρι στιγμής, στον παράλληλο προγραμματισμό, πολλές διεργασίες μπορούν να έχουν πρόσβαση σε κοινά δεδομένα και μεταβλητές. Είναι λοιπόν απαραίτητο να υπάρχει ένας μηχανισμός συγχρονισμού μεταξύ των διεργασιών ώστε να εξασφαλιστεί η σωστή λειτουργία του προγράμματος και η συνέπεια του αποτελέσματος. Σε αντίθετη περίπτωση το αποτέλεσμα θα είναι απροσδιόριστο και όχι το επιθυμητό.

Εφαρμόζοντας συγχρονισμό αποτρέπουμε περισσότερες από μία εργασία να εκτελούν ταυτόχρονα ένα τμήμα του προγράμματος, το οποίο καλείται κρίσιμο τμήμα. Όταν μία διεργασία εκτελεί το κρίσιμο τμήμα, τότε οι υπόλοιπες διεργασίες οφείλουν να περιμένουν μέχρι η διεργασία αυτή να ολοκληρώσει την εκτέλεση του κρίσιμου τμήματος.

Μερικοί μηχανισμοί συγχρονισμού:

- **Κλειδώματα (locks):** το κλείδωμα προστατεύει ένα τμήμα κώδικα από ταυτόχρονη πρόσβαση πολλών διεργασιών, καθώς μόνο η διεργασία που έχει λάβει το κλείδωμα μπορεί να προχωρήσει.
- **Σημαφόροι (semaphores):** ο σημαφόρος είναι ένα ακέραιος αριθμός που επιτρέπει μόνο τρεις ενέργειες:
    - Αρχικοποίηση
    - Αύξηση τιμής κατά 1
    - Μείωση τιμής κατά 1 και μπλοκάρισμα εάν η νέα τιμή είναι 0 (σε κάποιες υλοποιήσεις αρνητική)
- **Παρακολουθητές (monitors) και μεταβλητές συνθήκης (condition variables):** πρόκειται για ένα ζεύγος κλειδώματος και condition variable (m, c). Μία διεργασία αναστέλλει την εκτέλεσή της μέχρι να ισχύσει η κατάλληλη συνθήκη, ενώ μία διεργασία ξυπνάει κάποια από (ή όλες) τις διεργασίες που περιμένουν.

Στους παραπάνω μηχανισμούς χρησιμοποιείται η blocking τεχνική του αμοιβαίου αποκλεισμού. Η υλοποίηση της τεχνικής αυτής δεν είναι τόσο εύκολη καθώς συχνά είναι απαραίτητη η χρήση περισσοτέρων του ενός κλειδώματος. Στην περίπτωση αυτή μία προβληματική κατάσταση που μπορεί να προκύψει είναι αυτή του αδιεξόδου (deadlock). Όταν έχουμε deadlock, δύο διεργασίες περιμένουν η μία την άλλη να τελειώσει ώστε να ολοκληρωθούν. Είναι σαφές λοιπόν ότι ο προγραμματιστής πρέπει να δίνει ιδιαίτερη βάση στη σωστή χρήση των κλειδωμάτων.

Μία άλλη τεχνική συγχρονισμού είναι η χρήση ατομικών εντολών (atomic operations). Η ατομική εντολή επιτρέπει σε έναν επεξεργαστή να διαβάσει μία θέση μνήμης και να γράψει σε αυτή, χωρίς να μπορούν άλλοι επεξεργαστές να διαβάσουν τη θέση μνήμης ή να γράψουν σε αυτή την ίδια στιγμή. Το βασικό πλεονέκτημα των ατομικών εντολών είναι ότι πρόκειται για non-blocking τρόπο συγχρονισμού. Σε αντίθεση δηλαδή με τα κλειδώματα, όταν μία διεργασία κατέχει ένα κλείδωμα δεν εμποδίζονται οι υπόλοιπες διεργασίες από το να εκτελέσουν τον κώδικά τους. Το βασικό μειονέκτημα των ατομικών εντολών ότι μπορούν να εκτελέσουν ένα περιορισμένο σύνολο εντολών.

Μία ακόμα non-blocking τεχνική συγχρονισμού είναι η transactional memory. Πρόκειται για προγραμματιστικό μοντέλο που επιτρέπει ατομικές εκτελέσεις τμημάτων κώδικα. Ως βασικό κομμάτι της εκπόνησης της παρούσας διπλωματικής εργασίας, η transactional memory θα αναλυθεί περαιτέρω σε επόμενο κεφάλαιο.

Κεφάλαιο 2

Παράλληλες δομές δεδομένων και B+ Δέντρα

2.1 Παράλληλες δομές δεδομένων

Ο όρος δομή δεδομένων αναφέρεται σε διαφορετικούς τρόπους αποθήκευσης και οργάνωση δεδομένων σε έναν υπολογιστή, ώστε αυτά να μπορούν να χρησιμοποιηθούν εύκολα και αποδοτικά. Η επιλογή μίας συγκεκριμένης δομής δεδομένων έχει να κάνει με την εφαρμογή που μας ενδιαφέρει κάθε φορά.

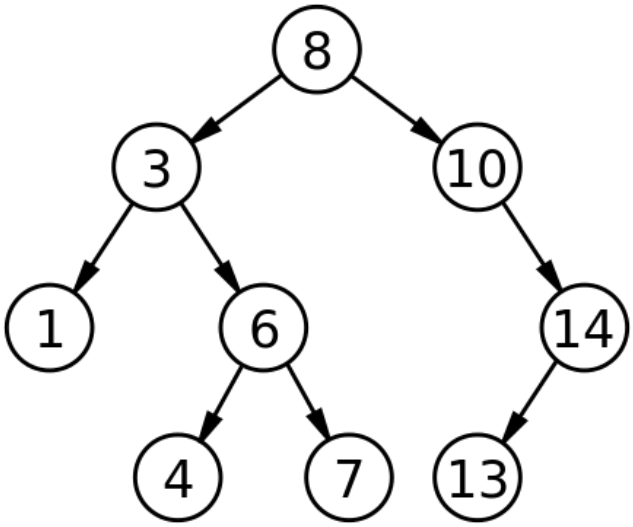
Για να χρησιμοποιηθεί μία δομή δεδομένων από έναν παράλληλο αλγόριθμο θα πρέπει αυτή να επανασχεδιαστεί κατάλληλα. Αυτό συμβαίνει καθώς σε έναν παράλληλο αλγόριθμο πολλές διεργασίες μπορούν ταυτόχρονα να έχουν πρόσβαση σε μία δομή δεδομένων. Θέλουμε λοιπόν αυτές οι προσβάσεις να γίνονται συγχρονισμένα, ώστε η δομή που χρησιμοποιούμε να παραμένει συνεπής. Με στόχο άρα τη δημιουργία μίας αποδοτικής παράλληλης δομής δεδομένων, είναι σημαντικό κατά την παραλληλοποίηση αυτής να ελαχιστοποιήσουμε το σειριακό κομμάτι του αλγορίθμου και το κόστος συγχρονισμού.

2.2 Δέντρα αναζήτησης

Στην επιστήμη των υπολογιστών, δέντρο ονομάζεται η δομή δεδομένων που προσομοιάζει την ιεραρχική δομή ενός δέντρου. Πρόκειται για ένα πεπερασμένο μη κενό σύνολο στοιχείων, από τα οποία ένα είναι η ρίζα, ενώ τα υπόλοιπα επιμερίζονται σε δέντρα που ονομάζονται υποδέντρα του δέντρου.

Ως δέντρο αναζήτησης ορίζεται ένα δέντρο του οποίου κάθε στοιχείο αποτελείται από ένα ζεύγος κλειδιού-τιμής εντός ενός συνόλου. Ένα δέντρο για να είναι δέντρο αναζήτησης πρέπει το κλειδί κάθε κόμβου να είναι μεγαλύτερο από όλα τα κλειδιά των αριστερά υποδέντρων και μικρότερο από αυτά των δεξιά υποδέντρων. Το πλεονέκτημα των δέντρων αναζήτησης είναι η αποδοτική αναζήτηση στοιχείων στο δέντρο, εφόσον το δέντρο είναι λογικά ισοζυγισμένο. Δηλαδή θα πρέπει κάθε φύλλο να έχει σχετικά το ίδιο βάθος στο δέντρο. Οι βασικές λειτουργίες ενός δέντρου αναζήτησης είναι η αναζήτηση (lookup), η εισαγωγή (insert) και η διαγραφή (delete) ενός στοιχείου. Στο σχήμα 2.1 φαίνεται ένα απλό δέντρο αναζήτησης.

Σε ένα δέντρο οι κόμβοι που έχουν παιδιά ονομάζονται εσωτερικοί, ενώ οι κόμβοι που δεν έχουν παιδιά ονομάζονται εξωτερικοί (φύλλα). Εάν σε ένα δέντρο οι τιμές μπορούν να αποθηκευτούν σε όλους τους κόμβους, τότε τα δέντρα αυτά ονομάζονται internal, ενώ, αν οι τιμές μπορούν να αποθηκευτούν μόνο σε εξωτερικούς κόμβους, τότε τα δέντρα ονομάζονται external. Στην δεύτερη περίπτωση οι εσωτερικοί κόμβοι ονομάζονται κόμβοι δρομολόγησης, καθώς δεν έχουν τιμές, αλλά χρησιμεύουν στο να μπορούμε να οδηγηθούμε στην τιμή που θέλουμε, η οποία είναι αποθηκευμένη στα φύλλα του δέντρου.



***Σχήμα 2.1**. Ένα απλό δέντρο αναζήτησης.*

2.3 Δυαδικά δέντρα αναζήτησης (BST)

Ένα δυαδικό δέντρο αναζήτησης είναι ένα internal δέντρο αναζήτησης, στο οποίο κάθε κόμβος έχει το πολύ δύο παιδιά. Το δυαδικό δέντρο αναζήτησης δεν είναι ισοζυγισμένο και κάθε παιδί είναι είτε φύλλο είτε ρίζα ενός άλλου δέντρου αναζήτησης. Κάθε εσωτερικός κόμβος έχει ένα κλειδί και τουλάχιστον ένα υποδέντρο. Επίσης τα δυαδικά δέντρα αναζήτησης ικανοποιούν την ιδιότητα της δυαδικής αναζήτησης, καθώς το κλειδί κάθε κόμβου είναι μεγαλύτερο από τα κλειδιά του αριστερού υποδέντρου και μικρότερο από τα κλειδιά του δεξιού υποδέντρου. Το δέντρο του σχήματος 2.1 είναι δυαδικό δέντρο αναζήτησης.

Υπάρχουν διάφορες παραλλαγές του δυαδικού δέντρου αναζήτησης όπως τα avl trees και τα red-black trees.

2.4 Β-Δέντρα

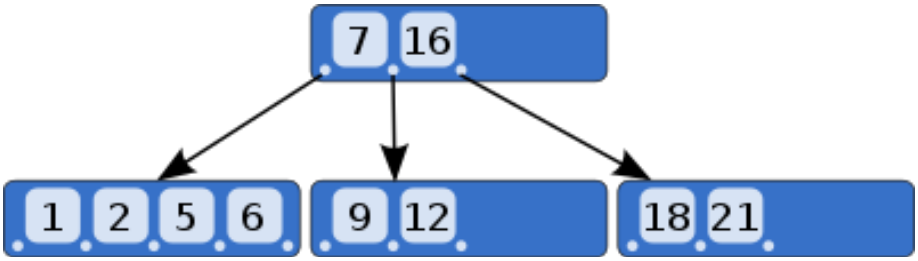
Ένα Β-Δέντρο (B-Tree) είναι ένα ισορροπημένο εσωτερικό δέντρο αναζήτησης που αναπτύχθηκε από του Bayer και McCreight το 1972[4]. Πρόκειται για μία γενίκευση του δυαδικού δέντρου αναζήτησης, όπου κάθε κόμβος μπορεί να έχει περισσότερα του ενός παιδιά. Στα B-Trees κάθε εσωτερικός κόμβος περιέχει έναν αριθμό κλειδιών και παιδιών, ο οποίος πρέπει να βρίσκεται εντός ενός συγκεκριμένου εύρους. Για το λόγο αυτό μετά από μία λειτουργία εισαγωγής ή διαγραφής μπορεί το δέντρο να χρειάζεται να ισορροπήσει ξανά. Αυτό γίνεται είτε με μετακίνηση κλειδιού μεταξύ αδελφικών κόμβων μέσω του πατρικού κόμβου, είτε με το σπάσιμο ενός κόμβου σε νέους, είτε με τη συνένωση κόμβων. Λόγω του ότι ο αριθμός των κλειδιών και των παιδιών ενός κόμβου επιτρέπεται να βρίσκεται σε ένα εύρος τιμών, στα B-Trees χρειάζεται σπανιότερα εξισορρόπηση σε σχέση με άλλα ισορροπημένα δέντρα αναζήτησης.

Βασικό χαρακτηριστικό ενός B-Tree είναι η τάξη του m. Ένα B-Tree με τάξη m έχει τις εξής ιδιότητες:

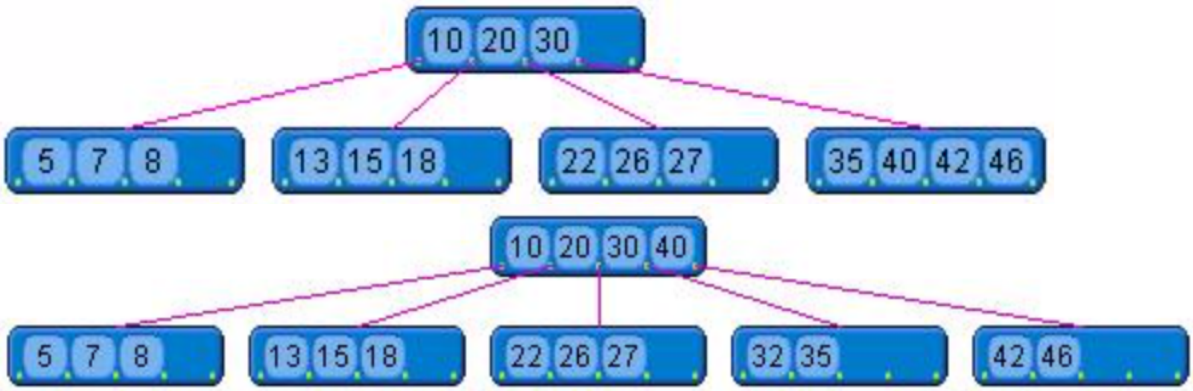
- Όλα τα φύλλα του δέντρου πρέπει να έχουν το ίδιο βάθος στο δέντρο.
- Όλοι οι κόμβοι εκτός της ρίζας πρέπει να έχουν τουλάχιστον $\lceil m/2 \rceil - 1$ κλειδιά και το πολύ m-1 κλειδιά.

- Όλοι οι εσωτερικοί κόμβοι εκτός της ρίζας πρέπει να έχουν τουλάχιστον ⌈m/2⌉ παιδιά.
- Εάν η ρίζα του δέντρου δεν είναι φύλλο πρέπει να έχει τουλάχιστον δύο παιδιά.
- Ένας εσωτερικός κόμβος με n-1 κλειδιά θα έχει n παιδιά.
- Τα κλειδιά σε έναν κόμβο πρέπει να είναι ταξινομημένα.

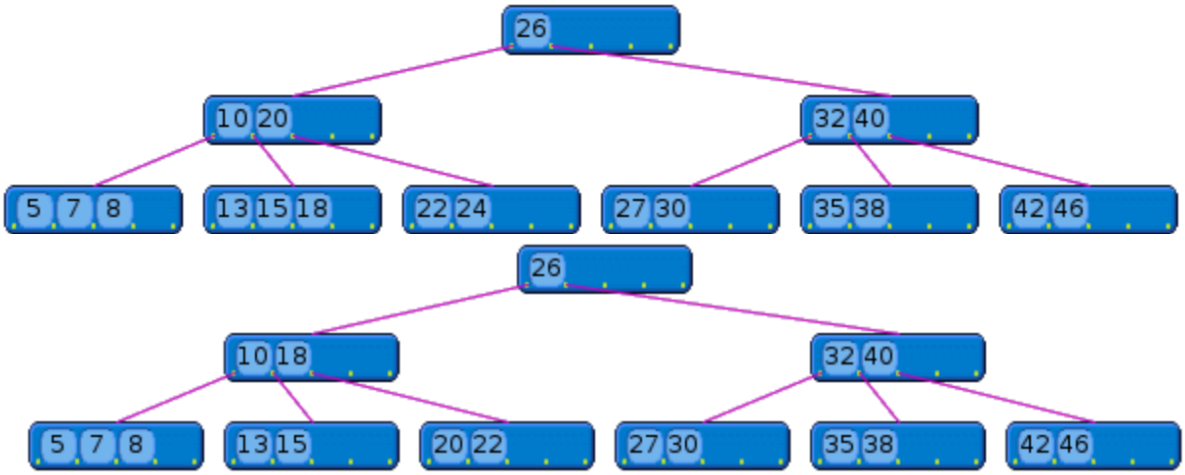
Στο σχήμα 2.2 απεικονίζεται ένα B-Tree τάξης 5, ενώ στα σχήματα 2.3 και 2.4 φαίνονται οι λειτουργίες εισαγωγής και διαγραφής αντίστοιχα.



Σχήμα 2.2. *Ένα B-Tree τάξης 5.*



Σχήμα 2.3. *Ένα B-Tree πριν και μετά την εισαγωγή του 32. Βλέπουμε ότι έχουμε σπάσιμο του τελευταίου κόμβου σε δύο νέους και μεταφορά της τιμής 40 στον πατρικό κόμβο.*



Σχήμα 2.4. *Ένα B-Tree πριν και μετά τη διαγραφή του 24. Βλέπουμε ότι καθώς το δεύτερο φύλλο πρόκειται να έχει λιγότερα από τα απαιτούμενα στοιχεία, γίνεται μεταφορά του 18 από το προηγούμενο φύλλο στον πατρικό κόμβο και μετακίνηση από τον πατρικό κόμβο στο φύλλο του 20.*

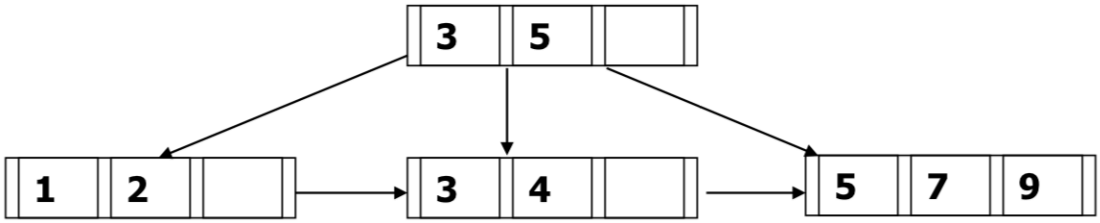
2.5 B+ Δέντρα

2.5.1 Γενικά

Ένα B+ Δέντρο (B+ Tree) μπορεί να περιγραφεί ως ένα B-Tree, το οποίο όμως είναι external. Οι εσωτερικοί κόμβοι χρησιμοποιούνται δηλαδή μόνο για δρομολόγηση, ενώ τα ζεύγη κλειδιών-τιμών αποθηκεύονται στα φύλλα. Επίσης, τα φύλλα του δέντρου ενώνονται μεταξύ τους. Κάθε φύλλο δηλαδή έχει δείκτη (pointer) που δείχνει στο επόμενο φύλλο. Ως τάξη b ενός B+ Tree ορίζεται ο μέγιστος αριθμός παιδιών που μπορεί να έχει ένας κόμβος. Τα B+ Trees χρησιμοποιούνται ιδιαίτερα σε βάσεις δεδομένων (data bases) και συστήματα αρχείων (file systems). Στο σχήμα 2.5 φαίνονται τα χαρακτηριστικά των κόμβων ενός B+ Tree ανάλογα με το είδος τους, ενώ στο σχήμα 2.6 φαίνεται ένα τυπικό B+ Tree τάξης 4.

Τύπος Κόμβου	**Τύπος Παιδιών**	**Ελάχιστος αριθμός παιδιών**	**Μέγιστος αριθμός παιδιών**	**Αριθμός παιδιών για b=16**
Ρίζα (μόνος κόμβος του δέντρου)	Τιμές	1	b-1	1-15
Ρίζα	Εσωτερικοί κόμβοι ή φύλλα	2	b	2-16
Εσωτερικός κόμβος	Εσωτερικοί κόμβοι ή φύλλα	$\lceil b/2 \rceil$	b	8-16
Φύλλο	Τιμές	$\lceil (b-1)/2 \rceil$	b-1	8-15

Σχήμα 2.5. *Χαρακτηριστικά ενός B+ Tree.*

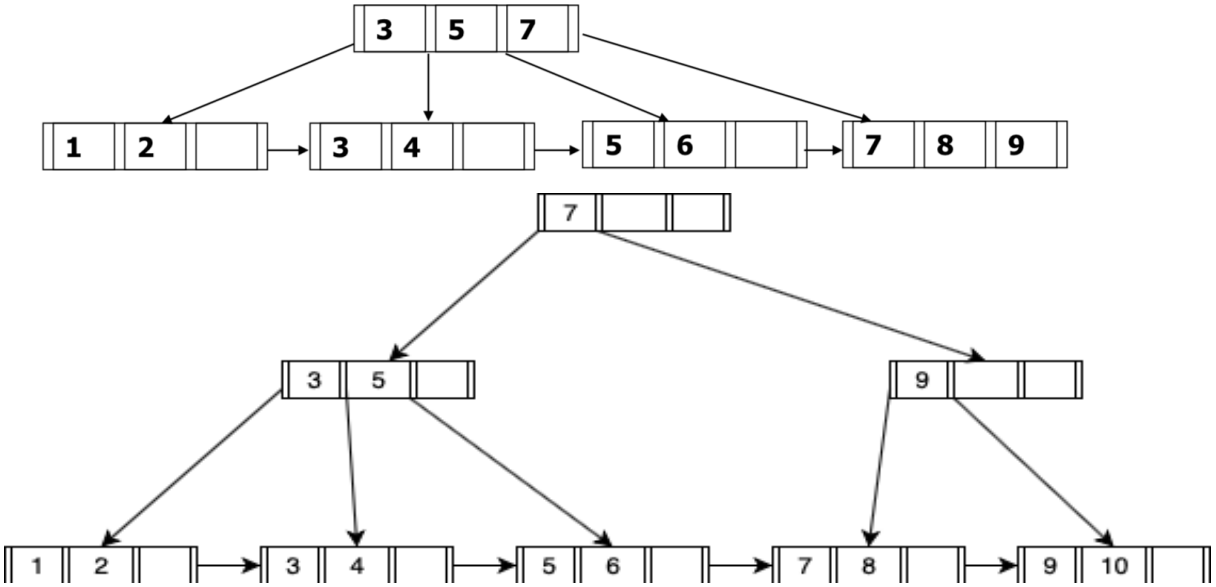


Σχήμα 2.6. *Ένα B+ Tree.*

2.5.2 Λειτουργίες σε ένα B+ Δέντρο

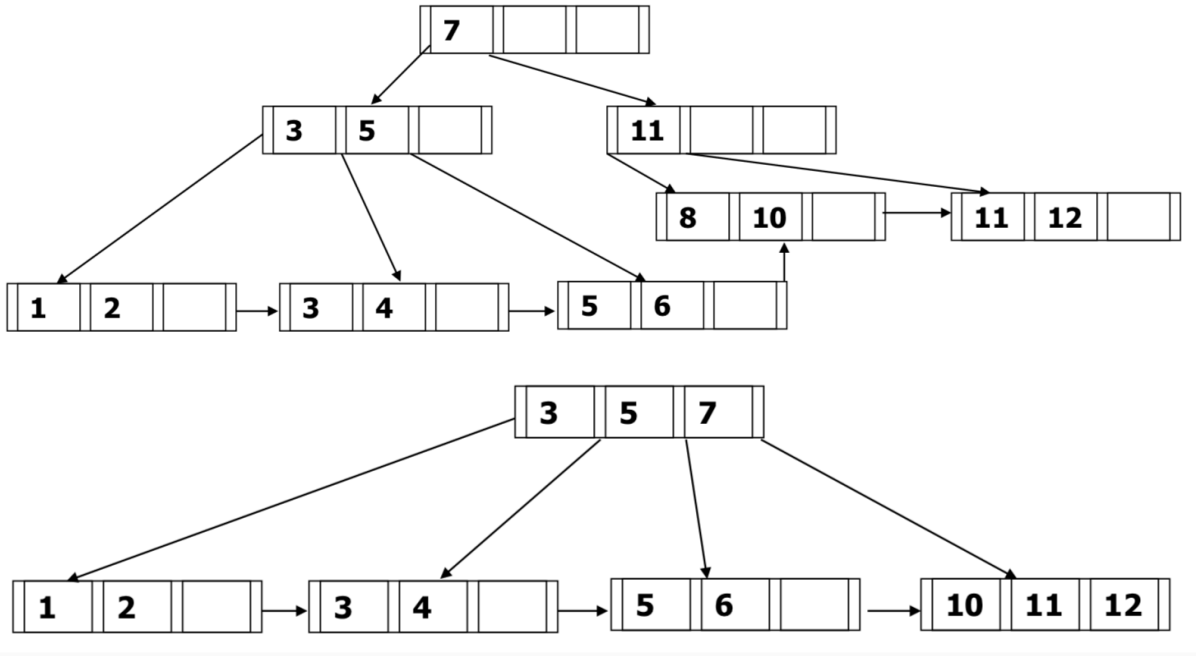
Σε ένα B+ Tree έχουμε τις εξής βασικές λειτουργίες:

- **Αναζήτηση (search/lookup)**
  Για να ερευνήσουμε εάν ένα κλειδί βρίσκεται ή όχι στο δέντρο χρησιμοποιούμε τη λειτουργία της αναζήτησης. Ξεκινώντας από τη ρίζα και με τη βοήθεια των εσωτερικών κόμβων και των τιμών δρομολόγησης που περιέχουν, καταλήγουμε στο φύλλο που θα βρίσκεται το κλειδί που αναζητάμε, εάν αυτό υπάρχει στο δέντρο. Έπειτα κοιτάμε εάν υπάρχει το κλειδί στις τιμές του φύλλου και έτσι αποφασίζουμε εάν αυτό βρίσκεται στο δέντρο.
- **Εισαγωγή (insert)**
  Για να εισάγουμε ένα κλειδί στο δέντρο πρώτα εκτελούμε τη διαδικασία της αναζήτησης. Εάν το κλειδί υπάρχει ήδη στο δέντρο, τότε δεν κάνουμε τίποτα. Σε αντίθετη περίπτωση, εάν το φύλλο που πρέπει να γίνει η εισαγωγή δεν είναι γεμάτο, εισάγουμε το κλειδί. Εάν το φύλλο είναι γεμάτο, προχωράμε σε σπάσιμο του φύλλου σε δύο φύλλα και εισάγουμε το κλειδί, όπως είδαμε και στα B-Trees. Το σπάσιμο των κόμβων γίνεται μέχρι να φτάσουμε σε έναν κόμβο του δέντρου που να χωράει τους δύο νέους κόμβους. Έτσι μπορεί να οδηγηθούμε το πολύ μέχρι και σε σπάσιμο της ρίζας. Σημαντικό είναι κατά το σπάσιμο ενός φύλλου να προσέχουμε ώστε τα φύλλα να παραμένουν σωστά συνδεδεμένα μεταξύ τους. Στο σχήμα 2.7 φαίνεται η εισαγωγή ενός στοιχείου σε ένα B+ Tree με σπάσιμο φύλλου.



Σχήμα 2.7. *Ένα B+Tree πριν και μετά της εισαγωγή του κλειδιού 10. Βλέπουμε ότι στην περίπτωση αυτή χρειάστηκε σπάσιμο τόσο του φύλλου, όσο και του πατρικού του κόμβου, που στην περίπτωση αυτή είναι η ρίζα.*

- **Διαγραφή (delete)**
  Η διαγραφή ακολουθεί τη λογική της εισαγωγής μέχρι τον εντοπισμό του ζητούμενου φύλλου. Έπειτα, εφόσον το κλειδί που θέλουμε να διαγράψουμε περιέχεται στο φύλλο, διαγράφουμε το κλειδί, αλλιώς η διαγραφή δεν έχει κάποιο αποτέλεσμα. Αφού διαγράψουμε το κλειδί, ελέγχουμε εάν το φύλλο είναι τουλάχιστον μισογεμάτο, όπως πρέπει να είναι. Εάν δεν είναι θα πρέπει είτε να μεταφέρουμε ένα κλειδί από γειτονικό φύλλο το οποίο είναι περισσότερο από μισογεμάτο, ώστε και τα δύο φύλλα να έχουν τον απαραίτητο αριθμό κλειδιών, είτε να συνενώσουμε το φύλλο με ένα γειτονικό του φύλλο το οποίο είναι ακριβώς μισογεμάτο. Στη δεύτερη περίπτωση θα πρέπει να ελέγξουμε στη συνέχεια και τον πατρικό κόμβο ώστε να είναι και αυτός τουλάχιστον μισογεμάτος. Στο σχήμα 2.8 φαίνεται η λειτουργία της διαγραφής σε ένα B+ Tree με συνένωση φύλλων.



Σχήμα 2.8. *Ένα B+ Tree πριν και μετά τη διαγραφή του κλειδιού 8.*

- **Αναζήτηση Εύρους (range query)**
  Εκτός από τις πιο συνηθισμένες λειτουργίες αναζήτησης, εισαγωγής και διαγραφής, στα B+ Trees είναι συνηθισμένες και οι λειτουργίες αναζήτησης εύρους. Ζητούμενο ενός range query είναι η εύρεση των κλειδιών που ανήκουν σε ένα εύρος μεταξύ δυο δοσμένων κλειδιών. Τα B+ Trees είναι κατάλληλα για τις λειτουργίες αυτές, καθώς τα φύλλα του είναι συνδεδεμένα μεταξύ τους. Έτσι για την εκτέλεση ενός range query αρκεί να αναζητήσουμε την ελάχιστη τιμή του ζητούμενου εύρους και αφού εντοπίσουμε το φύλλο που αυτή μπορεί να βρίσκεται αρκεί να προσπελάσουμε τα φύλλα ένα-ένα μέχρι να βρούμε κάποιο στο οποίο υπάρχει κλειδί που είναι εκτός εύρους. Τα B+ Trees μας επιτρέπουν δηλαδή να πηγαίνουμε από το ένα φύλλο στο επόμενο κατευθείαν, χωρίς να χρειάζεται να ψάξουμε για όλες τις τιμές εντός εύρους ξεκινώντας από την ρίζα.

2.6 Τακτικές συγχρονισμού για παράλληλες δομές δεδομένων

Στις παράλληλες δομές δεδομένων, όπως και στα παράλληλα B+ Trees, μπορούμε να έχουμε ταυτόχρονη εκτέλεση λειτουργιών που αλλάζουν τη δομή. Είναι σαφές λοιπόν, πως για να έχουμε μία συνεπή δομή, θα πρέπει στην υλοποίηση της δομής να συμπεριλάβουμε ένα μηχανισμό συγχρονισμού, ο οποίος θα εξασφαλίζει πως οι λειτουργίες θα μπορούν να εκτελούνται παράλληλα με ασφάλεια. Στη συνέχεια θα δούμε τις πιο διαδεδομένες τακτικές συγχρονισμού.

2.6.1 Coarse-grain synchronization

Ο συγχρονισμός coarse-grain είναι μία τεχνική δημιουργίας παράλληλων δομών δεδομένων στην οποία χρησιμοποιείται ένα κλείδωμα για όλη τη δομή. Έτσι, όταν μία διεργασία θέλει να εκτελέσει μία λειτουργία στη δομή, τότε απαιτείται πρώτα να αποκτήσει το κλείδωμα, να εκτελέσει τη λειτουργία και έπειτα να το απελευθερώσει. Πρόκειται για

μία εύκολη στην υλοποίηση και στη χρήση τεχνική συγχρονισμού, η οποία όμως περιορίζει σημαντικά την παράλληλη πρόσβαση νημάτων στη δομή και κατά συνέπεια την απόδοση του παράλληλου προγράμματος, καθώς μονό ένα νήμα μπορεί να έχει πρόσβαση στη δομή κάθε στιγμή. Λειτουργίες δηλαδή όπως οι lookup, insert και delete εκτελούνται εν τέλει σειριακά και η απόδοση του παράλληλου προγράμματος δε διαφέρει από αυτή του σειριακού.

2.6.2 Fine-grain synchronization

Στην τεχνική αυτή χρησιμοποιούνται περισσότερα κλειδώματα σε τμήματα της δομής. Σε ένα B+ Tree για παράδειγμα θα μπορούσαμε να έχουμε ένα κλείδωμα για κάθε κόμβο του δέντρου. Η τεχνική αυτή προσφέρει μεγαλύτερο παραλληλισμό σε σχέση με την coarse-grain, καθώς πολλά νήματα μπορούν να έχουν παράλληλη πρόσβαση στη δομή, εφόσον ενδιαφέρονται για διαφορετικά τμήματα αυτής. Η χρησιμοποίηση πολλών κλειδωμάτων όμως δημιουργεί και μεγάλο κόστος κτήσης και απελευθέρωσης αυτών, ενώ πρόκειται γενικά για μία δύσκολα υλοποιήσιμη τεχνική. Για να υπάρχει πρόοδος και να αποφεύγονται τα αδιέξοδα κάθε λειτουργία θα πρέπει να αποκτά τα κλειδώματα με την ίδια σειρά.

2.6.3 Optimistic synchronization

Παρόλο που αποτελεί βελτίωση σε σχέση με τον coarse-grain συγχρονισμό, ο fine-grain συγχρονισμός δημιουργεί όπως είδαμε καθυστερήσεις με τις κτήσεις και απελευθερώσεις κλειδωμάτων, ενώ υπάρχει και η πιθανότητα ένα νήμα να μπλοκάρει κάποιο άλλο από το να εργαστεί σε ένα άλλο τμήμα της δομής. Μία περεταίρω βελτίωση αποτελεί ο optimistic συγχρονισμός. Σύμφωνα με την τεχνική αυτή δε χρησιμοποιούμε κλειδώματα όταν αναζητούμε ένα στοιχείο κατά τις λειτουργίες της εισαγωγής και της διαγραφής. Αφού εκτελέσουμε την αναζήτηση χωρίς κλειδώματα, κλειδώνουμε τους κατάλληλους κόμβους και ελέγχουμε αν η δομή είναι συνεπής. Για παράδειγμα μπορεί να ελεγχθεί εάν δύο κόμβοι είναι ακόμα προσβάσιμοι από τη ρίζα και εάν η μεταξύ τους σχέση συνεχίζει να ισχύει. Εάν κάτι έχει αλλάξει απελευθερώνουμε τα κλειδώματα και προσπαθούμε από την αρχή, ενώ εάν η δομή παραμένει συνεπής εκτελούμε τη λειτουργία. Για την μέθοδο της αναζήτησης η τεχνική του optimistic συγχρονισμού εξακολουθεί να χρησιμοποιεί κλειδώματα. Η τεχνική επιτρέπει την παράλληλη πρόσβαση νημάτων στη δομή και είναι αποτελεσματική, καθώς το κόστος να διατρέξουμε τη δομή μία φορά με κλειδώματα είναι μεγαλύτερο απ' ότι περισσότερες φορές χωρίς κλειδώματα. Επίσης το να χρειαστεί να επαναληφθεί μία λειτουργία δε συμβαίνει ιδιαίτερα συχνά.

2.6.4 Lazy Synchronization

Ένα βασικό μειονέκτημα του optimistic συγχρονισμού είναι το γεγονός ότι η λειτουργία της αναζήτησης χρησιμοποιεί κλειδώματα. Αυτό είναι ιδιαίτερα αποθαρρυντικό, καθώς οι λειτουργίες της αναζήτησης είναι πολύ συχνότερες από τις υπόλοιπες. Τη λύση στο πρόβλημα αυτό έρχεται να δώσει ο lazy συγχρονισμός.

Σύμφωνα με την τεχνική αυτή προσθέτουμε σε κάθε κόμβο (του B+ Tree για παράδειγμα) μία boolean μεταβλητή, την οποία ονομάζουμε marked και μας δείχνει εάν ο κόμβος βρίσκεται στη δομή ή έχει διαγραφεί. Έτσι η λειτουργίες αναζήτησης μπορούν τώρα να διατρέχουν τη λίστα χωρίς κλειδώματα, ελέγχοντας επιπλέον την boolean μεταβλητή. Η insert, όπως και πριν, διατρέχει τη δομή, κλειδώνει τους απαραίτητους

κόμβους, ελέγχει τη συνέπεια και ανάλογα συνεχίζει. Για τον έλεγχο της συνέπειας δε χρειάζεται πλέον να διατρέξουμε τη δομή από την αρχή, αλλά αρκούν τοπικοί έλεγχοι στους κόμβους. Η delete είναι lazy, δηλαδή εκτελείται σε δύο βήματα. Κατά το πρώτο βήμα ο κόμβος αφαιρείται λογικά από τη δομή με το πεδίο marked να γίνεται true, ενώ κατά το δεύτερο βήμα γίνεται η φυσική αφαίρεση του κόμβου και η επανατοποθέτηση των δεικτών.

2.6.5 Non-blocking synchronization

Τελευταίο βήμα είναι να απαλείψουμε πλήρως τα κλειδώματα και από τις τρεις βασικές λειτουργίες των δομών δεδομένων (lookup, insert και delete). Αυτό είναι σημαντικό, καθώς καθιστά τη δομή ταχύτερη, ενώ αποφεύγουμε καταστάσεις όπου η αποτυχία του νήματος που κρατάει ένα κλείδωμα οδηγεί στην αποτυχία όλης της εφαρμογής. Η γενική ιδέα του non-blocking συγχρονισμού είναι να χειριστούμε τα πεδία marked και next του κόμβου της δομής (π.χ. μίας λίστας) σαν μία ατομική μονάδα, που ο επεξεργαστής μπορεί να χειριστεί ατομικά, ώστε κάθε προσπάθεια αλλαγής του next, όταν το marked είναι true, δηλαδή ο κόμβος έχει διαγραφεί από τη δομή, να αποτυγχάνει.

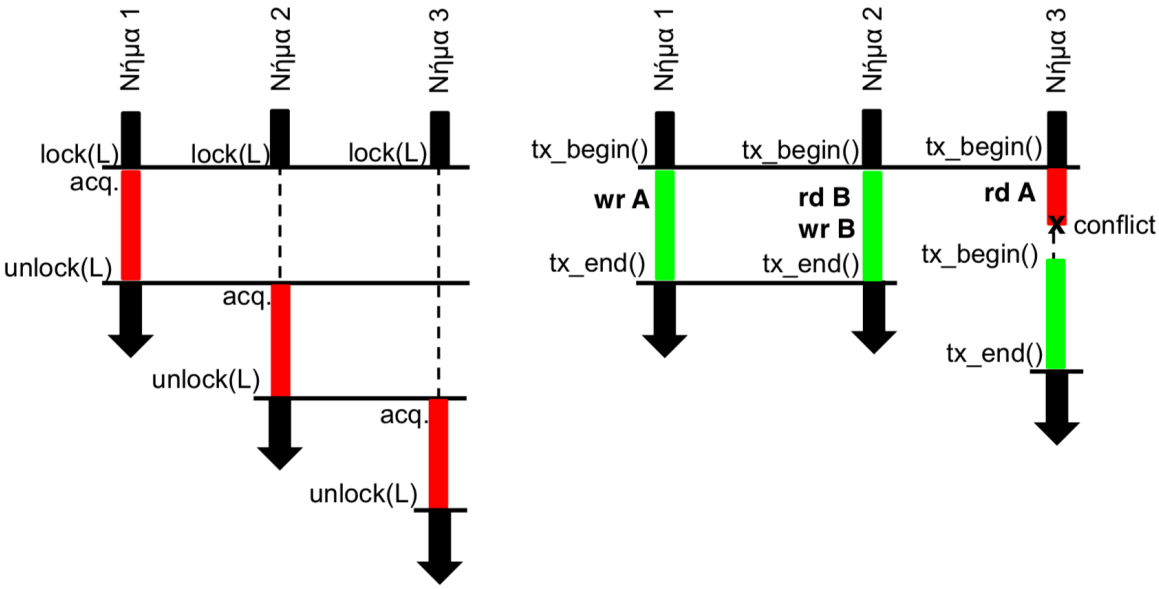
Πιο συγκεκριμένα, η αφαίρεση ενός κλειδιού περιλαμβάνει την ενημέρωση του πεδίου marked, καθώς και μία απόπειρα να ενημερωθούν οι διευθύνσεις. Κάθε νήμα που διατρέχει τη δομή εκτελώντας την insert ή την delete αφαιρεί φυσικά τους κόμβους που συναντά και έχουν σβηστεί λογικά. Η αποτυχία για ατομική ενημέρωση θα οδηγεί στην επαναπροσπάθεια διατρέχοντας από την αρχή. Επίσης έχουμε τη βοηθητική κλάση Window και τη μέθοδο find(), η οποία επιστρέφει ένα δείκτη στο ακριβώς μικρότερο στοιχείο από αυτό που πρόκειται να προστεθεί ή να διαγραφεί και έναν στο ακριβώς μεγαλύτερο στοιχείο. Επίσης, εάν συναντήσει στοιχεία που έχουν σβηστεί λογικά, τα σβήνει και φυσικά.

Κεφάλαιο 3
Transactional Memory

3.1 Γενικά

Όπως είδαμε στο προηγούμενο κεφάλαιο είναι εύκολο να υλοποιήσουμε coarse-grain αλγορίθμους, αλλά για να έχουμε καλή απόδοση είναι απαραίτητο να υλοποιήσουμε fine-grain αλγορίθμους, οι οποίοι είναι σαφώς πιο δύσκολοι. Επίσης είδαμε ότι τα κλειδώματα σε πολλές περιπτώσεις δεν είναι βοηθητικά και ρίχνουν την απόδοση. Ιδανικά θέλουμε να συνδυάσουμε την προγραμματιστική ευκολία ενός coarse-grain συγχρονισμού και την κλιμακωσιμότητα της fine-grain τεχνικής. Τη λύση έρχεται να δώσει η Transactional Memory (TM).

Στην Transactional Memory ο προγραμματιστής σημειώνει κομμάτια κώδικα τα οποία πρέπει να εκτελεστούν ατομικά, τα οποία ονομάζονται transaction, και το TM σύστημα είναι υπεύθυνο για τη σωστή εκτέλεση. Ο χρήστης δηλώνει δηλαδή το τι πρέπει να γίνει, αλλά όχι το πως, μεταφέροντας έτσι την πολυπλοκότητα του fine-grain συγχρονισμού στο TM σύστημα. Η απόδοση σε αυτήν την περίπτωση εξαρτάται από το κρίσιμο τμήμα, καθώς και την υλοποίηση του TM συστήματος, το οποίο πρέπει να εξασφαλίζει **ατομικότητα** (atomicity), δηλαδή πως είτε όλες οι εγγραφές ενός transaction θα αποθηκευτούν στη μνήμη (commit) είτε καμία (abort), **απομόνωση** (isolation), δηλαδή πως οι εγγραφές ενός transaction είναι ορατές στον υπόλοιπο κώδικα μόνο μετά το commit και **σειριοποίηση** (serializability), δηλαδή πως τα αποτελέσματα των transaction είναι ίδια με αυτά της σειριακής εκτέλεσης. Στο σχήμα 3.1 βλέπουμε σχηματικά παράδειγμα εκτέλεσης ενός κρίσιμου τμήματος από 3 νήματα όταν έχουμε κλειδώματα και όταν χρησιμοποιούμε TM.



Σχήμα 3.1. *Παράδειγμα εκτέλεσης ενός κρίσιμου τμήματος με κλειδώματα και Transactional Memory.*

Συνοψίζοντας τα πλεονεκτήματα της Transactional Memory αξίζει να αναφερθούμε στα εξής:

- Ευκολία προγραμματισμού παραπλήσια με αυτή των coarse-grain υλοποιήσεων.
- Επίδοση παραπλήσια με αυτή των fine-grain υλοποιήσεων.
- Όταν ένα νήμα αποτυγχάνει δε «χάνονται» κλειδώματα. Σε περίπτωση αποτυχίας γίνεται abort και ξαναπροσπαθούμε.
- Μπορούμε να συνθέσουμε επιμέρους ατομικές λειτουργίες σε μία ενιαία ατομική λειτουργία, εξασφαλίζοντας μία ασφαλή εκτέλεση, αλλά και κλιμακωσιμότητα.

3.2 Υλοποίηση ενός TM συστήματος

Η Transactional Memory βασίζεται στο γεγονός ότι δε χρειάζεται κάποιος μηχανισμός συγχρονισμού εφόσον το σύστημα μπορεί να ανιχνεύει conflicts μεταξύ λειτουργιών που εκτελούνται από διαφορετικές διεργασίες. Τα conflicts μπορούν να προκύψουν είτε από εγγραφή ενός transaction σε μία θέση μνήμης την οποία διαβάζει ή γράφει ένα άλλο transaction, είτε από ανάγνωση μίας θέσης μνήμης από ένα transactions, στην οποία ένα άλλο transaction πραγματοποιεί εγγραφή. Όταν δεν ανιχνευθούν conflicts κατά την εκτέλεση ενός transaction τότε μπορεί να γίνει commit, δηλαδή να γίνουν ορατές οι αλλαγές του transaction για όλους του επεξεργαστές. Σε αντίθετη περίπτωση έχουμε abort, δηλαδή όλες οι αλλαγές απορρίπτονται και το transaction είναι σα να μην έγινε ποτέ.

Ένα TM σύστημα μπορεί να υλοποιηθεί είτε ως λογισμικό (Software Transactional Memory – STM), είτε στο υλικό (Hardware Transactional Memory – HTM), είτε ως συνδυασμός των δύο αυτών τεχνικών (Hybrid Transactional Memory). Δύο βασικά ζητήματα κατά την υλοποίηση ενός TM συστήματος είναι το data versioning, δηλαδή η διαχείριση των παλιών και των νέων εκδόσεων δεδομένων για την περίπτωση του abort, και η εύρεση και επίλυση συγκρούσεων (conflict detection and resolution), ώστε να ανιχνεύουμε πότε πρέπει να γίνει abort.

3.2.1 Software Transactional Memory (STM)

Πρόκειται για μία υλοποίηση TM που βασίζεται αποκλειστικά στο λογισμικό και μπορεί να υλοποιηθεί είτε ως lock-free αλγόριθμος είτε με τη χρήση κλειδωμάτων. Βασικό της πλεονέκτημα είναι το ότι, καθώς βασίζεται αποκλειστικά στο λογισμικό, μπορεί να χρησιμοποιηθεί σε οποιοδήποτε σύστημα. Από την άλλη, για τον ίδιο λόγο, δεν αποτελεί ιδιαίτερα αποδοτική επιλογή. Μεταξύ άλλων STM υλοποιήσεις είναι και οι: TinySTM και Lightweight Transactional Library σε C/C++, Shielded και STMNet σε C#, Deuce και JVSTM σε Java και ScalaSTM σε Scala. Οι STM υλοποιήσεις έρχονται σε μορφή βιβλιοθήκης.

3.2.2 Hardware Transactional Memory (HTM)

Πρόκειται για μία υλοποίηση TM που βασίζεται αποκλειστικά στο υλικό, όπου γίνονται η ανίχνευση των conflicts και η εκτέλεση των commit και abort. Η ιδέα είναι να βασιστούμε στα ήδη υπάρχοντα πρωτόκολλα για cache-coherence, τα οποία παρέχουν σχεδόν ότι χρειαζόμαστε για να υλοποιήσουμε τα transactions. Μπορούν δηλαδή να εντοπίσουν και να επιλύσουν conflicts μεταξύ εγγραφών, καθώς και μεταξύ εγγραφών και

αναγνώσεων, ενώ χρησιμοποιούν buffer για αβέβαιες αλλαγές αντί να αλλάζουν κατευθείαν τη μνήμη. Αρκεί λοιπόν μόνο η αλλαγή ορισμένων πραγμάτων.

Παρά του ότι η HTM είναι λιγότερο χρονοβόρα της STM, συνεχίζει να προσθέτει κόστος κατά την εκτέλεση του transaction ειδικά στην περίπτωση συνεχόμενων abort. Βασικό μειονέκτημα όμως του HTM είναι το ότι δεν μπορεί ένα πρόγραμμα να εκτελεστεί σε διαφορετικά συστήματα. Ο κώδικας θα πρέπει να διαφοροποιείται για να εκτελεστεί σε διαφορετικούς επεξεργαστές που υποστηρίζουν διαφορετικές υλοποιήσεις HTM, ενώ δε θα μπορεί να εκτελεστεί σε μηχάνημα που δεν υποστηρίζει HTM.

3.2.3 Hybrid Transactional Memory

Πρόκειται για υλοποίηση TM που συνδυάζει τις δύο παραπάνω υλοποιήσεις. Υπάρχουν υλοποιήσεις που εξαρτώνται περισσότερο από το υλικό, αλλά και υλοποιήσεις που εξαρτώνται περισσότερο από το λογισμικό. Για παράδειγμα υπάρχουν υλοποιήσεις που χρησιμοποιούν το υλικό για να επιταχύνουν STM υλοποιήσεις. Αυτές ονομάζονται hardware accelerated STMs.

3.2.4 Data versioning

Όπως είπαμε ήδη, ένα βασικό ζήτημα υλοποίησης της TM είναι το data versioning, δηλαδή η διαχείριση παλιών (committed) και νέων (uncommitted) δεδομένων για ταυτόχρονα transactions. Δύο κύριες μέθοδοι είναι το lazy versioning και το eager versioning.

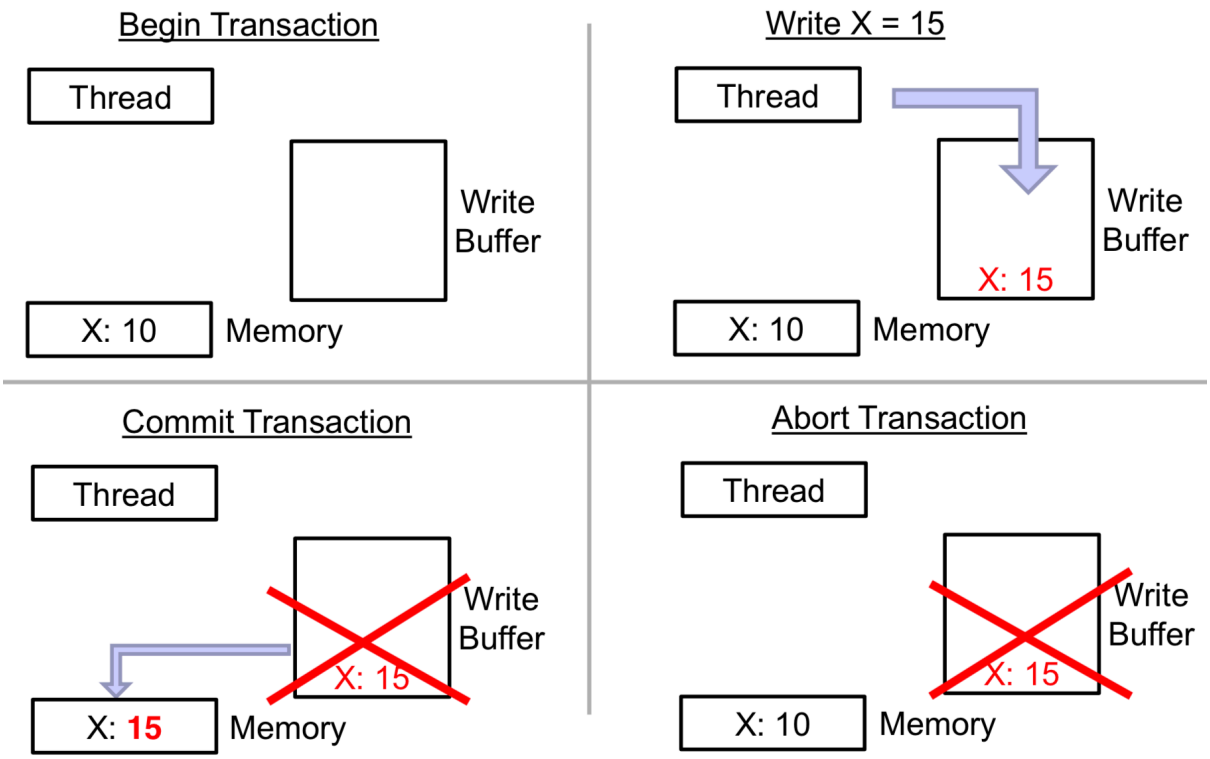
Σύμφωνα με το lazy versioning χρησιμοποιούμε έναν buffer για καταγραφή των αλλαγών και η πραγματική ενημέρωση της μνήμης γίνεται κατά το commit. Αυτό έχει ως αποτέλεσμα να έχουμε γρήγορα aborts, καθώς αρκεί απλά να «πετάξουμε» τον buffer. Από την άλλη πλευρά όμως έχουμε αργά commits, καθώς τότε πρέπει να γραφτούν τα δεδομένα στη μνήμη. Αντίθετα στο eager versioning έχουμε άμεση ενημέρωση της μνήμης, ενώ κρατάμε μία εγγραφή (log) με τα παλιά δεδομένα. Στην περίπτωση αυτή έχουμε γρήγορο commit καθώς τα δεδομένα είναι ήδη στη μνήμη, αλλά αργά aborts. Στις εικόνες των σχημάτων 3.2 και 3.3 φαίνονται παραδείγματα lazy και eager versioning.

3.2.5 Conflict detection and Resolution

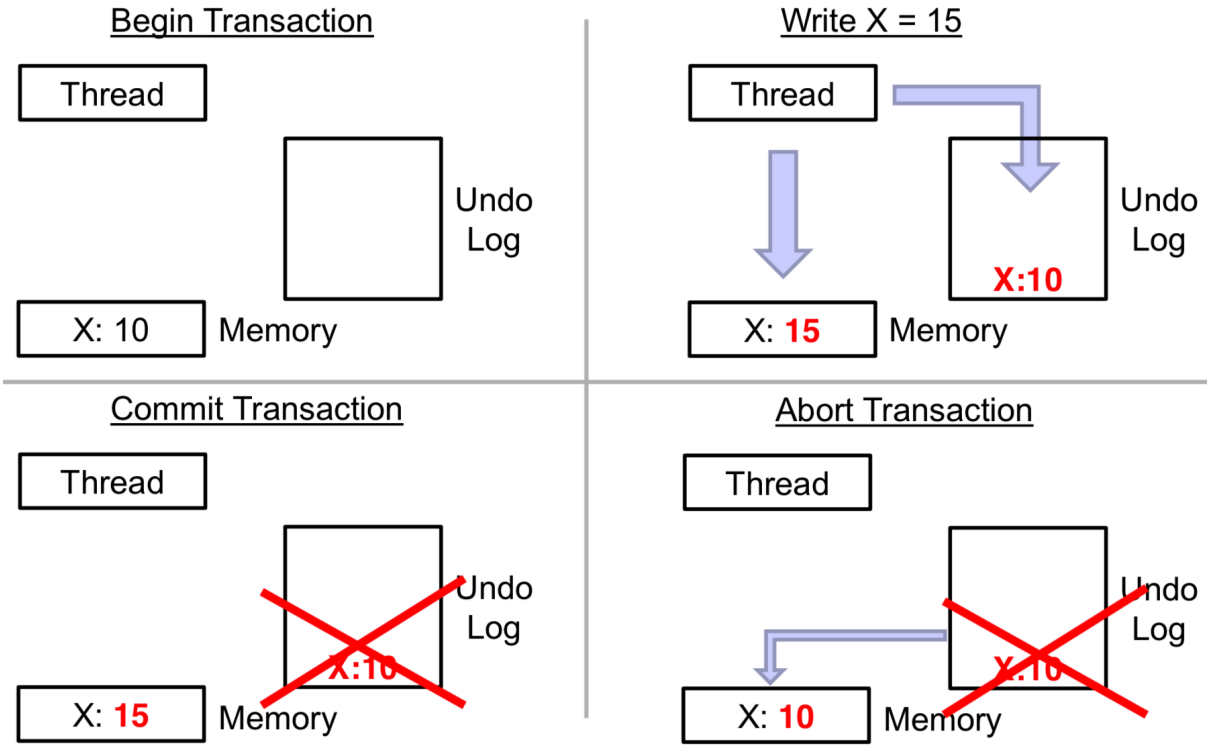
Το δεύτερο βασικό ζήτημα υλοποίησης της TM είναι η ανίχνευση και η επίλυση των συγκρούσεων. Τα δύο είδη conflict που μπορεί να συμβούν είναι το read-write conflict, όπου ένα transaction διαβάζει διεύθυνση που έχει γραφτεί από τρέχων transaction, καθώς και το write-write conflict όπου δύο transactions γράφουν στην ίδια διεύθυνση μνήμης. Για την επίλυση ενός conflict υπάρχουν διάφορες πολιτικές όπως οι stall, writer wins και committer wins, ενώ δύο κύριες μέθοδοι υλοποίησης του conflict detection είναι το pessimistic (eager) detection και το optimistic (lazy) detection.

Σύμφωνα με το pessimistic detection γίνεται έλεγχος για conflicts σε κάθε load και store, καθώς και χρήση contention manager για να αποφασίσει να κάνει stall ή abort. Η «απαισιοδοξία» έγκειται στο γεγονός ότι θεωρούμε ότι θα έχουμε conflict και έτσι ελέγχουμε μετά από κάθε πρόσβαση στη μνήμη. Σε περίπτωση που όντως έχουμε conflict γλιτώνουμε άσκοπη δουλειά. Σύμφωνα με το optimistic detection από την άλλη γίνεται έλεγχος για conflicts κατά το commit, ενώ σε περίπτωση conflict δίνεται προτεραιότητα στο commit transaction. Η «αισιοδοξία» έγκειται στο γεγονός ότι πιστεύουμε ότι δε θα

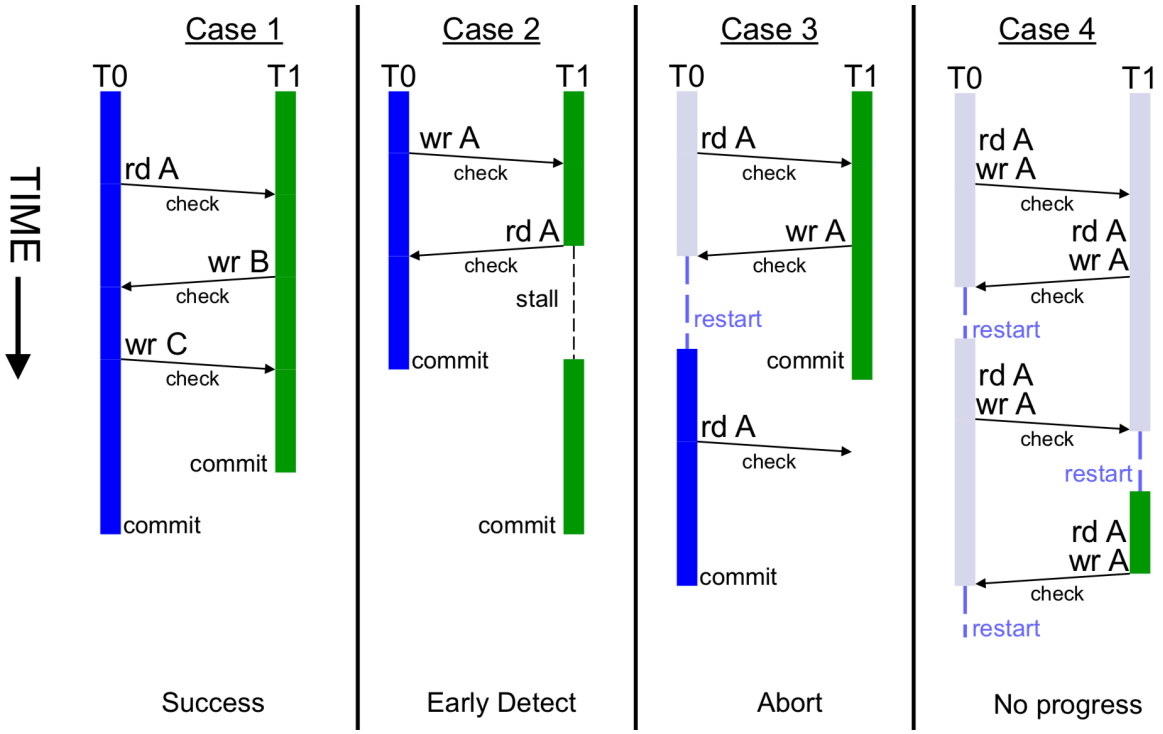
έχουμε conflict και άρα ελέγχουμε μόνο κατά το commit, αποφεύγοντας τον έλεγχο σε κάθε πρόσβαση στη μνήμη. Στις εικόνες των σχημάτων 3.4 και 3.5 φαίνονται παραδείγματα pessimistic και optimistic detection.



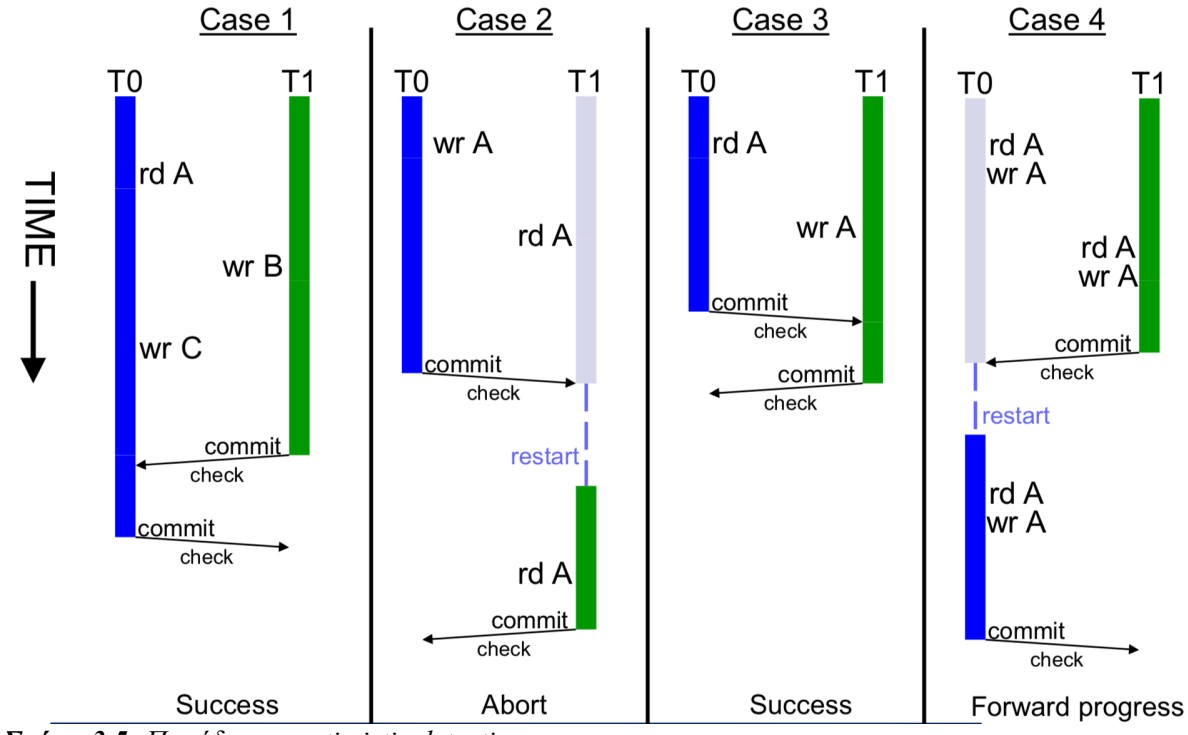
Σχήμα 3.2. *Παράδειγμα lazy versioning.*



Σχήμα 3.3. *Παράδειγμα eager versioning.*



Σχήμα 3.4. *Παράδειγμα pessimistic detection.*



Σχήμα 3.5. *Παράδειγμα optimistic detection.*

3.3 Intel® Transactional Synchronization Extensions (TSX)

Η Intel[5], ξεκινώντας το 2013, άρχισε να προσφέρει μία υλοποίηση HTM μέσω των επεξεργαστών Haswell. Πρόκειται για την πρώτη διαθέσιμη υλοποίηση σε εμπορικούς επεξεργαστές.

Ο προγραμματιστής αρκεί να σημειώσει το κομμάτι κώδικα που θα εκτελεστεί το transaction χρησιμοποιώντας τις εντολές και συναρτήσεις (extensions) που παρέχονται από την υλοποίηση. Όταν το κομμάτι αυτό εκτελείται, το σύστημα βρίσκεται σε transactional mode. Τότε, όλες οι θέσεις μνήμης που αφορούν το transaction μεταφέρονται στην μνήμη cache και κατηγοριοποιούνται σε δύο σετ, το read και το write, τα οποία αποτελούνται από τις θέσεις μνήμης που διαβάζει και γράφει το transaction αντίστοιχα. Αν ανιχνευθεί κάποιο conflict κατά τη διάρκεια του transaction, τότε γίνεται abort και όλες οι αλλαγές αναιρούνται. Σε αντίθετη περίπτωση το transaction γίνεται commit και οι τροποποιημένες cache lines μεταφέρονται στην κύρια μνήμη.

Το Intel TSX παρέχει στον χρήστη δύο interfaces, το Restricted Transactional Memory (RMT) και το Hardware Lock Elision (HLE). Η βασική διαφορά τους είναι πως ο κώδικας που προκύπτει με το HLE μπορεί να εκτελεστεί και σε συστήματα που δεν υποστηρίζουν TSX. Στην περίπτωση αυτή τα προθέματα αντιμετωπίζονται ως nops, δηλαδή εντολές που δεν κάνουν κάτι. Αντίθετα, το RTM προσφέρει μεγαλύτερη ευελιξία, καθώς ο προγραμματιστής επιλέγει τι θα γίνει σε περίπτωση abort.

3.3.1 Hardware Lock Elision (HLE)

Πρόκειται για έναν εύκολο τρόπο χρήσης HTM σε ήδη υπάρχοντα κώδικα. Το HLE αφαιρεί ουσιαστικά το ήδη υπάρχον κλείδωμα και προσπαθεί να εκτελέσει τον κώδικα ως transaction. Αν υπάρξει conflict, γίνεται abort, αποκτάται το κλείδωμα και το κρίσιμο τμήμα επανεκτελείται, ενώ σε αντίθετη περίπτωση γίνεται commit και οι αλλαγές καθίστανται μόνιμες. Το HLE είναι δηλαδή ένας απλός αλλά και περιοριστικός, λόγω των κλειδωμάτων και της μη ευελιξίας, τρόπος αξιοποίησης του HTM. Με το HLE εισάγονται δύο νέα προθέματα εντολών assembly, τα XACQUIRE και XRELEASE. Στο σχήμα 3.6 φαίνεται ένα παράδειγμα HLE σε γλώσσα C.

```
/* acquire lock */
while (__sync_lock_test_and_set(&lock_var) == 0)
    /* do nothing */;

... Critical section with lock acquired ...

/* release lock */
__sync_lock_release(&lock_var);
```

```
/* elide lock */
while (__hle_acquire_test_and_set(&lock_var) == 0)
    /* do nothing */;

... Critical section with lock acquired ...

/* release lock */
__hle_release_clear(&lock_var);
```

Σχήμα 3.6. *Παράδειγμα χρήσης του HLE. Στο πρώτο κομμάτι κώδικα βλέπουμε την υλοποίηση ενός κλασικού TAS lock, ενώ στο δεύτερο βλέπουμε πως κάνουμε elide το lock.*

3.3.2 Restricted Transactional Memory (RTM)

Πρόκειται για μία εναλλακτική υλοποίηση του HLE που δίνει περισσότερη ευελιξία στον προγραμματιστή, ο οποίος μπορεί να καθορίσει τι θα γίνει στην περίπτωση που το transaction δεν μπορέσει να εκτελεστεί επιτυχώς. Το RTM προσθέτει τέσσερεις νέες εντολές assembly για τη διαχείριση των transactions, τις XBEGIN, XEND, XTEST και XABORT. Ο προγραμματιστής σημειώνει το κρίσιμο τμήμα χρησιμοποιώντας τις εντολές XBEGIN και XEND για την αρχή και το τέλος του τμήματος αντίστοιχα. Η εντολή XTEST δείχνει εάν εκτελείται transaction η όχι, ενώ η εντολή XABORT(status) επιτρέπει στον προγραμματιστή να αναγκάσει το τρέχων transaction να κάνει abort, έχοντας τη δυνατότητα να καθορίσει μέσω της μεταβλητής status το λόγο για τον οποίο έγινε το abort.

Σε περίπτωση abort του transaction εκτελείται ο fallback κώδικας. Ο προγραμματιστής ορίζει τη θέση μνήμης στον κώδικα που θα εκτελεστεί σε περίπτωση abort (fallback address). Αυτή η διεύθυνση είναι η ακριβώς επόμενη εντολή μετά την XBEGIN. Η XBEGIN επιστρέφει μία τιμή που δείχνει αν μία διεργασία είναι σε transactional mode ή έχει γίνει abort. Έτσι ο EAX καταχωρητής ενημερώνεται σύμφωνα με την κατάσταση του transaction, όπως φαίνεται στον πίνακα του σχήματος 3.7. Ο προγραμματιστής έχει λοιπόν τη δυνατότητα να ελέγξει την κατάσταση του transaction εκτελώντας μία λογική πράξη μεταξύ της τιμής που επιστρέφεται και των παρακάτω σταθερών:

- _XBEGIN_STARTED: Το transaction έχει ξεκινήσει επιτυχώς.
- _XABORT_CONFLICT: Το transaction έχει γίνει abort λόγω conflict στη μνήμη.
- _XABORT_CAPACITY: Το transaction έχει γίνει abort λόγω υπερχείλισης μνήμης.
- _XABORT_EXPLICIT: Το transaction διεκόπη ρητά από τον προγραμματιστή.
- _XABORT_RETRY: Το transaction διεκόπη, αλλά αν ξαναπροσπαθήσει μπορεί να επιτύχει.
- _XABORT_DEBUG: Το transaction διεκόπη λόγω debug.
- _XABORT_NESTED: Έγινε abort σε ένα εσωτερικό εμφωλευμένο transaction.

Θέση bit στον EAX καταχωρητή	Σημασία
0	Στο λογικό 1 εάν είχαμε abort με εντολή XABORT.
1	Στο λογικό 1, εάν το transaction μπορεί να πετύχει με επαναπροσπάθεια.
2	Στο λογικό 1 εάν έχουμε conflict μνήμης.
3	Στο λογικό 1 εάν κάποιος εσωτερικός buffer υπερχείλισε.
4	Στο λογικό 1 εάν έγινε διακοπή λόγω debug.
5	Στο λογικό 1 εάν έγινε abort σε εμφωλευμένο transaction.
23:6	Reserved.
31:24	Το όρισμα που δόθηκε στην εντολή XABORT. Είναι έγκυρο όταν το bit 0 είναι στο λογικό 1.

Σχήμα 3.7. *Η κατάσταση του transaction όπως αποτυπώνεται στα bits του EAX καταχωρητή.*

Είναι πιθανό κάποια transactions να γίνονται συνέχεια abort και κατά συνέπεια να μην υπάρχει πρόοδος. Για την περίπτωση αυτή ο προγραμματιστής πρέπει να υλοποιήσει ένα εναλλακτικό non-transactional μονοπάτι (fallback path). Πρόκειται για μία εναλλακτική υλοποίηση που δεν χρησιμοποιεί το RTM και είναι συνήθως ένα κομμάτι κώδικα με coarse-grained locking. Η υλοποίηση αυτή είναι απαραίτητη, καθώς εγγυάται την πρόοδο στην εκτέλεση του προγράμματος. Περιλαμβάνει ένα καθολικό κλείδωμα, το οποίο πρέπει να εισάγεται από τον προγραμματιστή στο read set των transactions, για να γίνονται abort εάν αυτό κλειδωθεί από μία άλλη διεργασία και να μην εμφανιστούν προβλήματα συνάφειας μνήμης. Στο σχήμα 3.8 φαίνεται ένα παράδειγμα χρήσης του RTM σε γλώσσα προγραμματισμού C.

```
int aborts = MAX_TX_RETRIES;
lock_t = fallback_global_lock;

start_tx:
int status = TX_BEGIN();
if (status == TX_BEGIN_STARTED) {
    if (fallback_global_lock is locked)
        TX_ABORT();

    ... Critical Section ...

    TX_END();
} else { /* status != TX_BEGIN_STARTED */
    if (--aborts > 0)
        /* retry transaction */
        goto start_tx;

    acquire_lock(fallback_global_lock);
    ... Critical Section ...
    release_lock(fallback_global_lock);
}
```

Σχήμα 3.8. *Παράδειγμα χρήσης RTM.*

Κεφάλαιο 4

Παράλληλες λειτουργίες αναζήτησης εύρους (Concurrent range queries)

4.1 Γενικά

Όπως είδαμε και σε προηγούμενο κεφάλαιο ζητούμενο μίας λειτουργίας αναζήτησης εύρους είναι η εύρεση των κλειδιών, των οποίων οι τιμές βρίσκονται μεταξύ δύο δοσμένων τιμών. Η υλοποίηση των λειτουργιών αναζήτησης εύρους είναι αρκετά σαφής, ιδιαίτερα σε δομές δεδομένων όπως τα B+-Δέντρα, στα οποία αρκεί απλώς να βρούμε το φύλλο στο οποίο βρίσκεται (ή θα βρισκόταν εάν υπήρχε) το κλειδί με την ελάχιστη τιμή του εύρους (min) και να προσπελάσουμε (traverse) τα φύλλα του δέντρου μέχρι να βρούμε κάποιο, το οποίο περιέχει κλειδί με τιμή μεγαλύτερη της μέγιστης τιμής του ζητούμενου εύρους (max).

Η υλοποίηση αυτή είναι αρκετά εύκολη όταν προγραμματίζουμε σειριακά, όπου μία μόνο διεργασία/νήμα αναλαμβάνει να εκτελέσει όλες τις λειτουργίες, οι οποίες εκτελούνται σειριακά η μία μετά την άλλη. Όταν όμως προγραμματίζουμε για παράλληλα συστήματα απαιτείται κάποιος μηχανισμός συγχρονισμού που θα καθιστά την υλοποίηση συνεπής, αλλά ταυτόχρονα και αποδοτική. Στην περίπτωση αυτή τα πράγματα είναι σαφώς πιο δύσκολα, καθώς μία υλοποίηση δεν αρκεί να εκτελεί τις λειτουργίες που θέλουμε σωστά, αλλά να είναι και γρήγορη.

Μια αρχική απλοϊκή προσέγγιση είναι να χρησιμοποιήσουμε ένα κλείδωμα για όλη τη δομή του B+-Tree (coarse-grain locking). Στην περίπτωση αυτή κάθε λειτουργία της δομής, άρα και τα range queries, θα πρέπει να εξασφαλίσουν το κλείδωμα και ύστερα θα μπορούν να κάνουν το οτιδήποτε όσον αφορά τη δομή, είτε πρόκειται για ανάγνωση της δομής, είτε για εγγραφή σε αυτή. Η υλοποίηση αυτή είναι αρκετά εύκολη για τον προγραμματιστή, αλλά πρακτικά δεν εκμεταλλεύεται καθόλου ένα πολυπύρηνο σύστημα, καθώς κάθε λειτουργία θα πρέπει να περιμένει την ολοκλήρωση της προηγούμενης για να ξεκινήσει. Κατά συνέπεια μπορεί εύκολα να κριθεί ως μη αποδοτική. Μία fine-grain υλοποίηση από την άλλη πλευρά θα είχε καλύτερη απόδοση, όμως θα είχε σοβαρό μειονέκτημα όσον αφορά την προγραμματιστική δυσκολία. Ακόμα, τα πολλά κλειδώματα προσθέτουν επιπλέον κόστος επικοινωνίας.

4.2 Lock-free Linearizable 1-Dimensional Range Queries

Μία τεχνική υλοποίησης range queries που αξίζει να σημειωθεί είναι αυτή που προτάθηκε από τον Chatterjee το 2017[14]. Πιο συγκεκριμένα ο Chatterjee παρουσιάζει μία γενική μέθοδο υλοποίησης λειτουργιών αναζήτησης εύρους σε lock-free μονοδιάστατες ταξινομημένες δομές δεδομένων. Για να το υλοποιήσει αυτό χρησιμοποιεί την ιδέα που παρουσιάζεται στο [18], χρησιμοποιώντας το σχεδιασμό των αντικειμένων snap-collector[17]. Για την ακρίβεια φτιάχνει ένα ενεργό σύνολο διαδικασιών που εκτελούν λειτουργίες αναζήτησης εύρους και τις εφοδιάζει με ένα αντικείμενο range-collector, το οποίο είναι ουσιαστικά ένα αντικείμενο snap-collector, όπως αυτό περιγράφεται στο [17], το οποίο έχει προκαθορισμένο associated range.

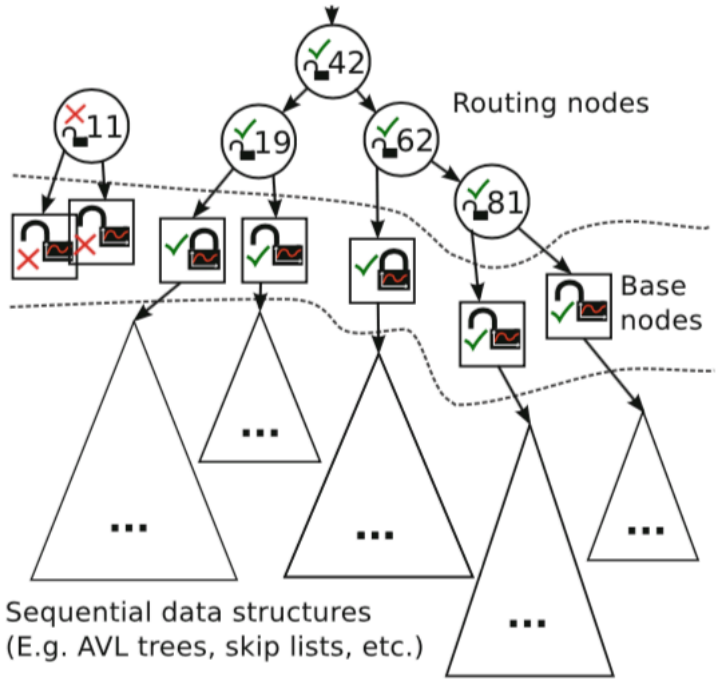
Για να πραγματοποιήσει ένα ενεργό σύνολο χρησιμοποιεί μία lock-free συνδεδεμένη λίστα από range-collectors, όπου ένας καινούργιος range-collector μπορεί να

τοποθετηθεί μόνο στην ουρά της. Ξεκινώντας ένα range query (εδώ ονομάζεται Range Search) διασχίζει τη λίστα ψάχνοντας έναν active range-collector με συμπίπτων εύρος. Εάν αυτός βρεθεί, τότε χρησιμοποιείται για μία παράλληλη συντονισμένη σάρωση εύρους, ενώ μόλις αυτή ολοκληρωθεί ο range-collector διαγράφεται από τη λίστα.

4.3 Range Queries Using Contention Adapting Search Trees

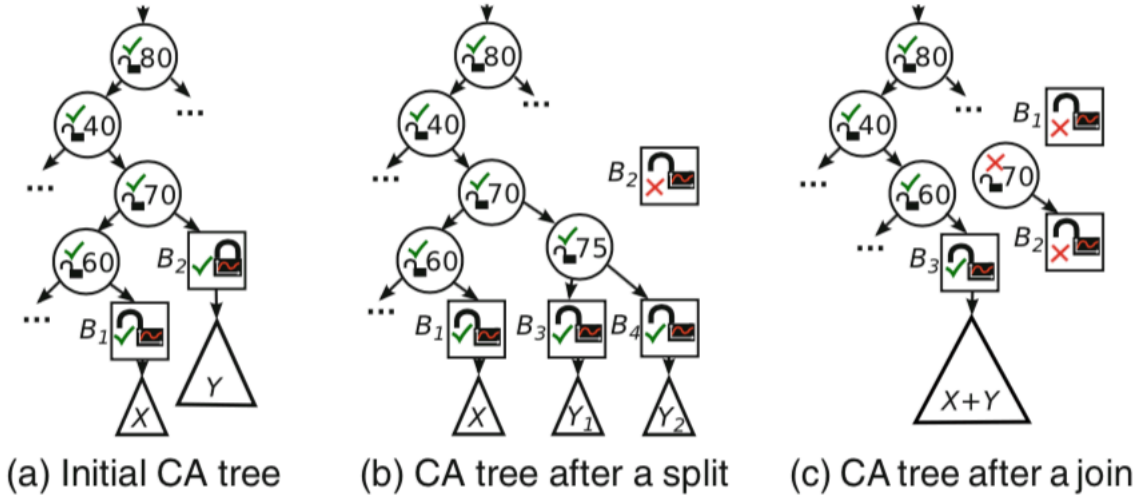
Το 2015 οι Sagonas και Winblad[6] πρότειναν μία νέα υλοποίηση για παράλληλες λειτουργίες αναζήτησης εύρους, χρησιμοποιώντας Contention Adapting Search Trees (CA trees), τα οποία αναπτύχθηκαν από τους ίδιους[7]. Τα CA trees είναι μία οικογένεια δομών δεδομένων που συλλέγουν στατιστικά δεδομένα για τον ανταγωνισμό που υπάρχει από τις διεργασίες/νήματα για κάποια στοιχεία και μπορούν να προσαρμόζονται δυναμικά ανάλογα με τα δεδομένα αυτά και τα μοτίβα αναζήτησης ακόμα και όταν αυτά αλλάζουν δυναμικά. Εκτός από τα range queries, οι Sagonas και Winblad χρησιμοποιούν τα CA trees και για range updates, κάτι όμως που δε θα μας απασχολήσει στα πλαίσια της παρούσας διπλωματικής εργασίας.

Σε ένα CA tree τα στοιχεία (κλειδιά και τιμές) αποθηκεύονται σε μία ακολουθιακή δομή δεδομένων (π.χ. δέντρα AVL, skip-lists κλπ.), η οποία έχει ως ρίζα έναν κόμβο-βάση, ο οποίος περιλαμβάνει ένα κλείδωμα καθώς και τα προαναφερθέντα στατιστικά στοιχεία. Ο συγχρονισμός των προσβάσεων είναι ανεξάρτητος για κάθε κόμβο-βάση, ενώ οι κόμβοι-βάσεις είναι συνδεδεμένοι μεταξύ τους με κόμβους δρομολόγησης, όπως φαίνεται και στο σχήμα 4.1.



Σχήμα 4.1. *Η δομή ενός CA tree. Το σχήμα προέρχεται από το [7].*

Όταν εντοπίζεται ότι ο ανταγωνισμός για έναν κόμβο βάση είναι μεγάλος, τότε το υποδέντρο της βάσης αυτής σπάει στα δύο, ώστε να μειωθεί ο ανταγωνισμός. Αντίστοιχα, εάν ο ανταγωνισμός για έναν κόμβο-βάση είναι χαμηλός, τότε ο κόμβος-βάση ενώνεται με έναν γειτονικό του κόμβο-βάση. Οι λειτουργίες αυτές φαίνονται σχηματικά στο σχήμα 4.2.



Σχήμα 4.2. *Το σπάσιμο και η συνένωση κόμβων-βάσεων σε ένα CA tree. Το σχήμα προέρχεται από το [7].*

Ο ανταγωνισμός υπολογίζεται ως εξής. Η κάθε διεργασία/νήμα ελέγχει εάν χρειάστηκε ή όχι να περιμένει για να αποκτήσει το κλείδωμα και αντίστοιχα αυξομειώνει τον μετρητή στατιστικών που βρίσκεται στον κόμβο-βάση. Η αναζήτηση στους κόμβους δρομολόγησης γίνεται χωρίς τη χρήση κλειδωμάτων, παρόλο που αυτοί περιέχουν κλείδωμα και ένα flag (✓ ή ✗), που δείχνει εάν ο κόμβος είναι έγκυρος. Τα τελευταία χρησιμοποιούνται για συγχρονισμό στην περίπτωση συνένωσης κόμβων-βάσεων, ενώ ένας κόμβος γίνεται μη έγκυρος όταν αντικαθίσταται κατά τη συνένωση ή το διαχωρισμό κόμβων.

Για λειτουργίες όπως η εισαγωγή και η διαγραφή στοιχείου σε ένα CA tree, αρχικά γίνεται αναζήτηση για τον κόμβο-βάση, κάτω από τον οποίο θα βρίσκεται, αν υπάρχει, το ζητούμενο κλειδί. Στη συνέχεια ο κόμβος-βάση κλειδώνεται και ελέγχεται εάν είναι έγκυρος. Στην περίπτωση που δεν είναι έγκυρος η αναζήτηση επαναλαμβάνεται μέχρι να καταλήξει σε κάποιον έγκυρο κόμβο. Έπειτα η λειτουργία προωθείται στη δομή που βρίσκεται κάτω από τον κόμβο-βάση, όπου και εκτελείται κανονικά. Πριν ο κόμβος-βάση ξεκλειδωθεί γίνεται έλεγχος για τον ανταγωνισμό, ώστε να διαπιστωθεί εάν χρειάζεται κάποια προσαρμογή. Εάν διαπιστωθεί ότι χρειάζεται, τότε αυτή πραγματοποιείται.

Οι λειτουργίες αναζήτησης μπορούν να υλοποιηθούν παρόμοια με τις λειτουργίες της εισαγωγής και της διαγραφής. Χρησιμοποιώντας όμως ένα seqlock (sequential lock) στους κόμβους-βάσεις οι λειτουργίες αναζήτησης μπορούν να υλοποιηθούν αισιόδοξα (optimistically) χωρίς την ανάγκη δηλαδή απόκτησης του κλειδώματος. Σε περίπτωση που η αισιόδοξη αναζήτηση αποτύχει, τότε πρέπει να αποκτηθεί το κλείδωμα και έπειτα να εκτελεστεί η αναζήτηση.

Για τις λειτουργίες αναζήτησης εύρους χρειάζεται να κλειδωθούν όλοι οι κόμβοι-βάσεις που μπορεί να περιέχουν κλειδιά εντός τους εύρους. Για να αποφευχθούν τα αδιέξοδα τα κλειδώματα αποκτούνται πάντα κατά αύξουσα σειρά. Αρχικά εντοπίζεται και κλειδώνεται ο πρώτος κόμβος-βάση που περιέχει κλειδιά εντός του εύρους. Στη συνέχεια πρέπει να βρεθούν οι υπόλοιποι κόμβοι-βάσεις, κάτι που δεν είναι εύκολο, καθώς κόμβοι δρομολόγησης μπορεί να σπάσουν και να αποκοπούν από το δέντρο. Όπως και στις λειτουργίες αναζήτησης, προτείνεται μία αισιόδοξη βελτιστοποίηση. Χρησιμοποιώντας,

όπως και προηγουμένως τα seqlocks, τα range queries μπορούν να εκτελεστούν χωρίς την ανάγκη για απόκτηση κλειδωμάτων. Στην περίπτωση που η αισιόδοξη εκτέλεση αποτύχει, τότε το range query θα πρέπει να πραγματοποιηθεί κανονικά χρησιμοποιώντας τα κλειδώματα.

4.4 Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data

Το 2017 προτάθηκε από τον Winblad[15] μία νέα τεχνική υλοποίησης παράλληλων range queries, η οποία συνδυάζει τα CA trees, όπως και η τεχνική που είδαμε στην προηγούμενη ενότητα, με δομές δεδομένων που χρησιμοποιούν αμετάβλητα δεδομένα (immutable data). Η διαφορά μεταξύ δομών δεδομένων που χρησιμοποιούν immutable data σε σχέση με τις αντίστοιχες που χρησιμοποιούν μεταβλητά δεδομένα (mutable data) είναι το γεγονός ότι οι πρώτες δεν τροποποιούν τη δομή in-place, αλλά επιστρέφουν μία νέα έκδοση αυτής, αφήνοντας έτσι την αρχική δομή ανέπαφη.

Η χρήση immutable data επιτρέπει βελτιστοποίηση των CA δέντρων. Καθώς το CA tree υλοποιείται πλέον χρησιμοποιώντας mutable αναφορές/δείκτες σε immutable δεδομένα, οι λειτουργίες αναζήτησης και οι λειτουργίες αναζήτησης εύρους αρκεί να αποκτήσουν αντίγραφο των δεικτών που χρειάζονται κατά τη διάσχιση του δέντρου. Τα immutable δεδομένα μπορεί η λειτουργία να τα διασχίσει στη συνέχεια, αφού πλέον τα κλειδώματα των κόμβων βάσεων έχουν ξεκλειδωθεί. Η βελτιστοποίηση αυτή επιτρέπει, ιδιαίτερα στα range queries, σημαντική μείωση του χρόνου που χρειαζόταν για την ανάγνωση των mutable δεδομένων στην υλοποίηση της προηγούμενης ενότητας.

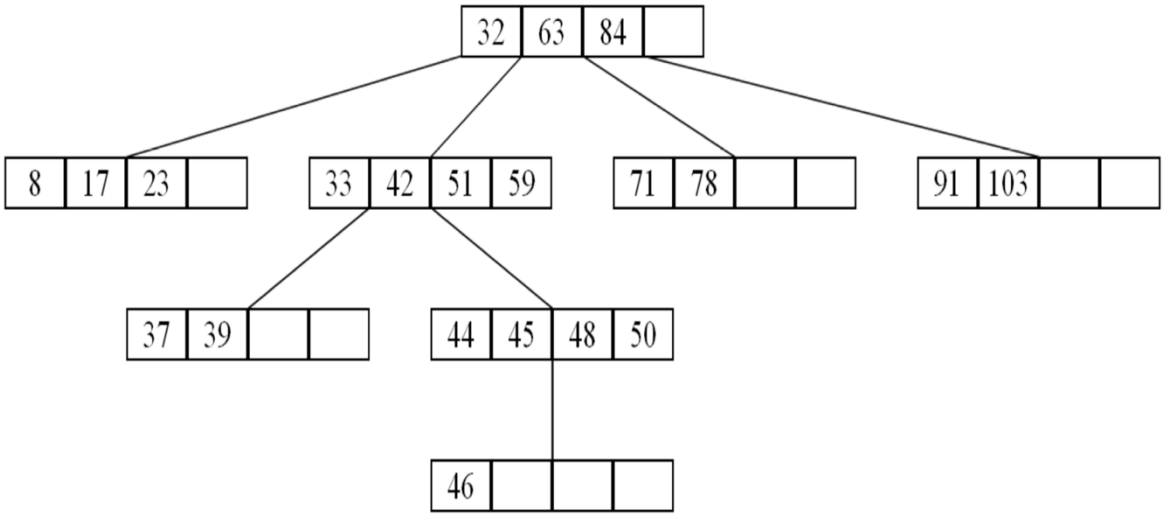
Για την υλοποίηση της βελτιστοποιημένης αυτής εκδοχής των CA trees, το CA tree χρησιμοποιεί μία mutable αναφορά σε ένα immutable treap[16]. Ένα treap είναι ένα self-balancing δυαδικό δέντρο αναζήτησης με αποτελεσματικές λειτουργίες split και join, γεγονός ιδιαίτερα σημαντικό για τις λειτουργίες των συνένωσης και σπασίματος στο CA tree.

4.5 Range Queries in Non-blocking k-ary Search Trees

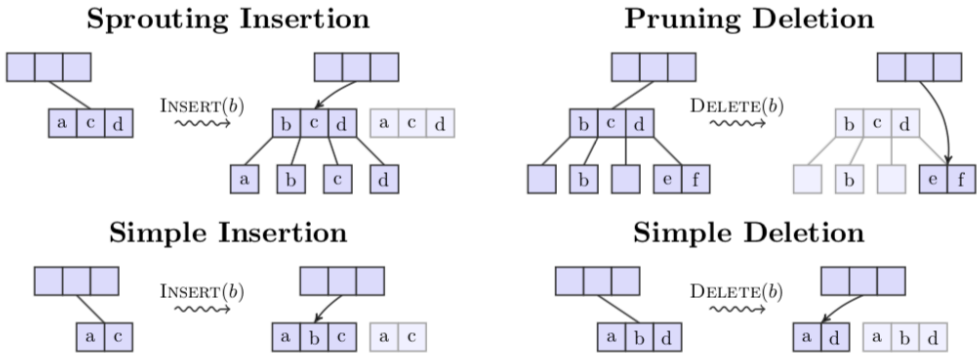
Μία ακόμα ενδιαφέρουσα υλοποίηση παράλληλων λειτουργιών αναζήτησης εύρους προτάθηκε από τους Brown και Anvi[8], οι οποίοι χρησιμοποίησαν ένα non-blocking k-ary search tree (k-ST), που προτάθηκε από τους Brown και Helga[9]. Το k-ary search tree είναι ένα δέντρο αναζήτησης, κάθε κόμβος του οποίου έχει το πολύ k παιδιά. Παράδειγμα του k-ary search tree για k=5 φαίνεται στο σχήμα 4.3. Αξίζει να σημειωθεί πως το δυαδικό δέντρο αναζήτησης (BST) είναι μία ειδική περίπτωση του k-ary search tree για k=2.

Στην υλοποίηση που χρησιμοποιήθηκε από τους Brown και Helga, το k-ST είναι external, κάτι που σημαίνει πως τα κλειδιά αποθηκεύονται μόνο στα φύλλα του δέντρου, ενώ οι εσωτερικοί κόμβοι είναι κόμβοι δρομολόγησης. Έτσι κάθε φύλλο έχει το πολύ k-1 κλειδιά, ενώ μπορεί και να μην έχουν κάποιο κλειδί, στην οποία περίπτωση ονομάζονται άδεια φύλλα. Οι εσωτερικοί κόμβοι θα πρέπει να έχουν ακριβώς k παιδιά και k-1 κλειδιά. Για την εισαγωγή ενός στοιχείου στο δέντρο, ένα φύλλο αντικαθίσταται είτε από ένα μεγαλύτερο φύλλο είτε από ένα μικρό υποδέντρο εάν το φύλλο ήταν ήδη γεμάτο. Η διαγραφή ενός στοιχείου από το δέντρο είτε αντικαθιστά ένα φύλλο με ένα μικρότερό του, είτε κλαδεύει τόσο το ίδιο το φύλλο όσο και το γονικό του κόμβο και στη θέση τους μένει

το μοναδικό μη άδειο φύλλο μετά τη διαγραφή του κλειδιού, όπως φαίνεται στο σχήμα 4.4, όπου φαίνονται παραδείγματα εισαγωγής και διαγραφής στοιχείων σε ένα k-ST.



Σχήμα 4.3. *Ένα k-ary Search Tree για k=5.*



Σχήμα 4.4. *Οι λειτουργίες της εισαγωγής και της διαγραφής στο k-ST. Το σχήμα προέρχεται από το [8].*

Για το συγχρονισμό των λειτουργιών της εισαγωγής και της διαγραφής σε κάθε εσωτερικό κόμβο προστίθεται ένα αντικείμενο UpdateStep, που δείχνει εάν μία λειτουργία έχει αποκλειστική πρόσβαση στους δείκτες στα παιδιά ενός κόμβου. Αυτή η τεχνική συγχρονισμού επεκτείνει τη δουλειά των Ellen et al.[10]. Όταν μία λειτουργία θέλει να αλλάξει ένα δείκτη σε έναν εσωτερικό κόμβο, τότε αποθηκεύει ατομικά ένα αντικείμενο UpdateStep στον κόμβο αυτό, χρησιμοποιώντας την ατομική εντολή compare and swap (CAS). Μία λειτουργία δεν μπορεί να αποθηκεύσει ένα αντικείμενο UpdateStep σε έναν κόμβο, εάν μία άλλη το έχει ήδη κάνει, μέχρι αυτή να το αποδεσμεύσει. Ένα αντικείμενο UpdateStep χωρίζεται σε flag και mark. Το flag χρησιμοποιείται για να δείξει ότι κάποιος δείκτης πρόκειται να αλλάξει, ενώ το mark αποτρέπει τους δείκτες ενός κόμβου να αλλάξουν όταν αυτός έχει αφαιρεθεί από το δέντρο. Για να γίνει, λοιπόν, μία εισαγωγή ή διαγραφή στοιχείου πρώτα δημιουργείται το αντίγραφο που θα τοποθετηθεί στο δέντρο και στη συνέχεια με χρήση CAS τροποποιείται το flag του κόμβου που θα γίνει η αλλαγή

δείκτη, ενώ αν χρειάζεται τροποποιείται και το mark (με χρήση CAS) του κόμβου που θα αφαιρεθεί. Ύστερα, αλλάζει ο δείκτης και το flag επανατροποποιείται πάλι με χρήση CAS.

Ένα σημαντικό χαρακτηριστικό της υλοποίησης αυτής είναι ο μηχανισμός helping που χρησιμοποιείται. Έστω ότι μία διεργασία έχει χρησιμοποιήσει το flag ή το mark σε κάποιον κόμβο με σκοπό να ολοκληρώσει κάποια αλλαγή στο δέντρο. Τα αντικείμενα flag ή mark τροποποιούνται ώστε να περιέχουν αρκετές πληροφορίες ώστε κάποια άλλη διεργασία να μπορεί να τα διαβάσει και να κάνει τις τροποποιήσεις στο δέντρο που ήθελε να ολοκληρώσει η πρώτη διεργασία. Έτσι, ο μηχανισμός helping εγγυάται την non-blocking πρόοδο, καθώς μία διεργασία που δεν μπορεί να συνεχίσει βοηθάει μία άλλη να ολοκληρωθεί και στη συνέχεια επαναλαμβάνεται.

Οι Brown και Anvi προσέθεσαν μία λειτουργία validate, η οποία χρησιμοποιείται για τα range queries, δέχεται ως όρισμα μία ακολουθία από δείκτες σε φύλλα και δείχνει αν τα φύλλα αυτά βρίσκονται ακόμα στο δέντρο ή αν κάποιο από αυτά έχει ή πρόκειται να αντικατασταθεί.

Για την υλοποίηση των λειτουργιών αναζήτησης εύρους, αρχικά εντοπίζονται όλα τα φύλλα που μπορεί να περιέχουν κλειδιά εντός του εύρους αναζήτησης. Αυτό γίνεται με αναζήτηση κατά βάθος, η οποία υλοποιείται με στοίβα αντί αναδρομής, ενώ για τα φύλλα που εντοπιστούν αποθηκεύεται μία λίστα από δείκτες που δείχνουν σε αυτά. Στη συνέχεια καλείται η validate και, εάν αυτή επιτύχει, τότε το range query ολοκληρώνεται επιστρέφοντας δείκτες στα φύλλα που περιλαμβάνουν κλειδιά εντός του εύρους. Σε περίπτωση αποτυχίας της validate το range query ξεκινάει από την αρχή.

Κεφάλαιο 5

Παράλληλες λειτουργίες αναζήτησης εύρους σε B+ Δέντρα με χρήση RCU-HTM

5.1 Εισαγωγή

Στο κεφάλαιο αυτό προτείνεται μία νέα μέθοδος για την υλοποίηση παράλληλων λειτουργιών αναζήτησης εύρους με χρήση B+ Δέντρων και της τεχνικής RCU-HTM[11],[12] που αναπτύχθηκε στο εργαστήριο υπολογιστικών συστημάτων της σχολής ηλεκτρολόγων μηχανικών και μηχανικών ηλεκτρονικών υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου από τους Siakavaras, Nikas, Goumas και Koziris και έχει εφαρμοστεί σε δέντρα δυαδικής αναζήτησης (BST).

Η τεχνική αυτή συνδυάζει την HTM, η οποία αναλύθηκε σε προηγούμενο κεφάλαιο, με την τεχνική Read-Copy-Update (RCU), η οποία θα αναλυθεί στη συνέχεια, και έχει εφαρμοστεί σε δέντρα δυαδικής αναζήτησης. Το RCU-HTM δίνει πολύ αποδοτικά παράλληλα δέντρα τα οποία επιτρέπουν στις διεργασίες/νήματα να τα διασχίζουν χωρίς κάποιο συγχρονισμό, καθώς οι αλλαγές στο δέντρο γίνονται σε αντίγραφα αυτού και όχι in-place.

5.2 Read-Copy-Update (RCU)

Η RCU[24] είναι μία τεχνική συγχρονισμού που βασίζεται στον αμοιβαίο αποκλεισμό. Σύμφωνα με την τεχνική αυτή οι λειτουργίες αναζήτησης μπορούν να εκτελούνται χωρίς την απαίτηση ύπαρξης συγχρονισμού, ο οποίος όμως είναι απαραίτητος μεταξύ των λειτουργιών που αλλάζουν τη δομή δεδομένων, όπως είναι η εισαγωγή και η διαγραφή κάποιου στοιχείου.

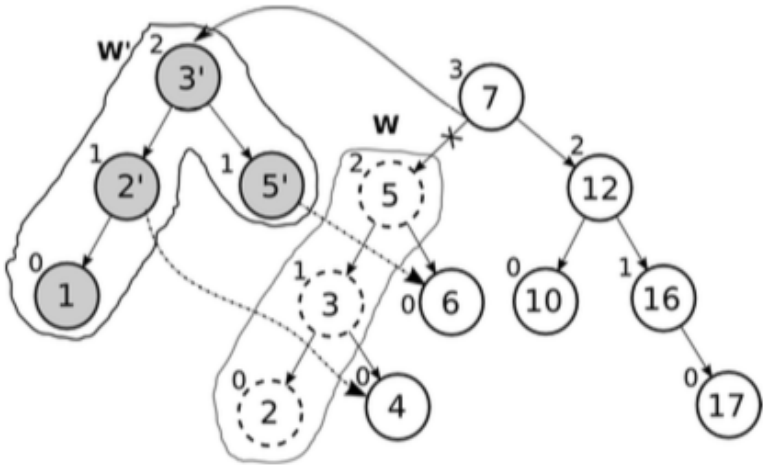
Πιο συγκεκριμένα, οι λειτουργίες της εισαγωγής και της διαγραφής στοιχείων διαβάζουν τη δομή δεδομένων και αντιγράφουν τα μέρη της που πρόκειται να τροποποιήσουν. Στη συνέχεια εκτελούν τις αλλαγές στα ιδιωτικά αντίγραφα που έχουν αποκτήσει και έπειτα αντικαθιστούν με αυτά τα παλιά δεδομένα στη δομή. Οι λειτουργίες αυτές πρέπει να συγχρονίζονται μεταξύ τους με κάποιο μηχανισμό συγχρονισμού, όπως τα κλειδώματα, ώστε να εξασφαλίζεται η συνέπεια της δομής.

Το πλεονέκτημα της συγκεκριμένης τεχνικής είναι ότι οι λειτουργίες αναζήτησης δε χρειάζονται συγχρονισμό, καθώς οι αλλαγές γίνονται σε ιδιωτικά αντίγραφα των δεδομένων και η ενημέρωση της δομής λαμβάνει χώρα σε ένα ατομικό στάδιο. Έτσι, οι υπόλοιπες διεργασίες/νήματα μπορούν να συνεχίσουν διασχίζοντας είτε τα παλιά είτε τα νέα δεδομένα. Είναι σαφές λοιπόν πως η RCU θα είναι πολύ αποτελεσματική σε περιπτώσεις που έχουμε μία εφαρμογή με μεγάλο όγκο αναζητήσεων. Από την άλλη πλευρά, η RCU δεν είναι ιδιαίτερα αποτελεσματική σε εφαρμογές όπου υπάρχει μεγάλος όγκος εισαγωγών και διαγραφών στοιχείων, ενώ επίσης απαιτεί και περισσότερη μνήμη από άλλες τεχνικές συγχρονισμού, λόγω των αντιγράφων που δημιουργούνται.

5.3 RCU-HTM

Ένα μειονέκτημα της RCU είναι η ύπαρξη αμοιβαίου αποκλεισμού μεταξύ διεργασιών που αλλάζουν τη δομή. Η τεχνική RCU-HTM έχει ως στόχο την εκμετάλλευση του ότι οι λειτουργίες αναζήτησης δε χρειάζονται συγχρονισμό στην RCU και το να επιτρέψει στις λειτουργίες της εισαγωγής και της διαγραφής να εκτελούν παράλληλα αλλαγές σε διαφορετικά τμήματα μίας δομής, όπως για παράδειγμα ενός δέντρου. Για να το πετύχει αυτό συνδυάζει την τεχνική RCU με την HTM. Στην ενότητα αυτή θα μιλήσουμε για την τεχνική RCU-HTM γενικά έχοντας υπόψη ένα δέντρο δυαδική αναζήτησης (BST), όπως είναι το AVL δέντρο.

Χρησιμοποιώντας την HTM λοιπόν παρέχεται η δυνατότητα στις λειτουργίες της εισαγωγής και της διαγραφής να τρέχουν παράλληλα χωρίς την ανάγκη συγχρονισμού και να αλλάζουν διαφορετικά τμήματα ενός δέντρου. Αυτό όμως οδηγεί σε πιο πολύπλοκες λειτουργίες, καθώς θα υπάρχουν περιπτώσεις όπου μία διεργασία/νήμα θα έχει ήδη αντικαταστήσει κόμβους που επρόκειτο να αντικαταστήσει κάποια άλλη διεργασία. Κατά συνέπεια κάποιες αλλαγές θα πρέπει να απορρίπτονται. Για να αποφευχθούν λανθασμένες καταστάσεις και να έχουμε συνέπεια αποτελεσμάτων εισάγεται ένα στάδιο επαλήθευσης (validation step) πριν αντικατασταθούν οι κόμβοι στο δέντρο. Πιο συγκεκριμένα, όταν οι λειτουργίες παίρνουν τα αντίγραφα των κόμβων που πρόκειται να αλλάξουν, σημειώνουν και την κατάστασή τους. Έτσι, πριν την αντικατάσταση των κόμβων και κατά το validation, ελέγχεται ότι η κατάσταση των κόμβων που πρόκειται να αντικατασταθούν δεν έχει αλλάξει και με αυτόν τον τρόπο εξασφαλίζεται η συνέπεια των δεδομένων. Σε περίπτωση που το validation δεν είναι επιτυχές, έχει αλλάξει δηλαδή κάτι σε έναν ή περισσότερους κόμβους, τότε η λειτουργία ξεκινάει πάλι από την αρχή και τα αντίγραφα απορρίπτονται. Ακόμα επειδή τα στάδια της επικύρωσης και της αντικατάστασης πρέπει να εκτελεστούν ατομικά περικλείονται μαζί από ένα HTM transaction. Στον κώδικα χρησιμοποιούνται οι μακροεντολές TX_BEGIN, TX_END και TX_ABORT για τη διαχείριση των HTM transactions, οι οποίες χρησιμοποιούν τις αντίστοιχες TSX assembly εντολές για την έναρξη, τον τερματισμό και τη διακοπή ενός transaction. Στο σχήμα 5.1 φαίνεται πως γίνεται η εισαγωγή ενός στοιχείου με τη χρήση αντιγράφων.



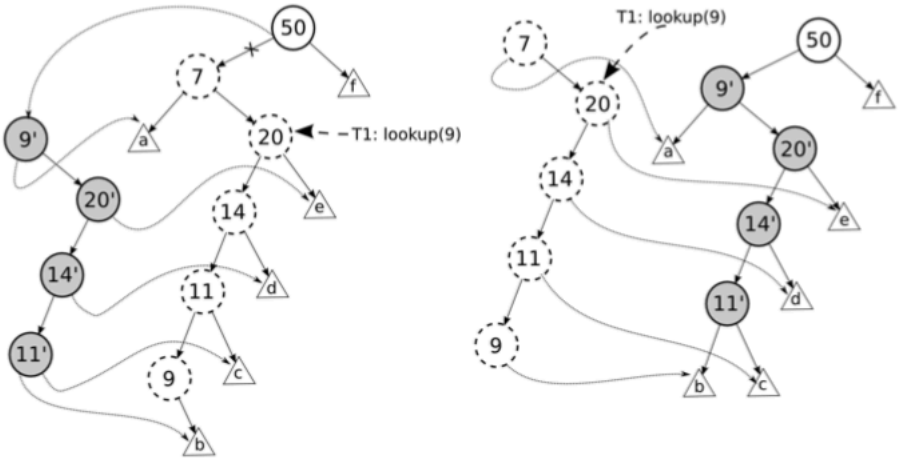
Σχήμα 5.1. *Οι αλλαγές για την εισαγωγή του στοιχείου 1 πραγματοποιούνται σε αντίγραφα των κόμβων του δέντρου και όχι in-place. Τα αντίγραφα αντικαθιστούν στη συνέχεια τους κόμβους στο δέντρο.*

Πιο συγκεκριμένα, για την αναζήτηση ενός στοιχείου χρησιμοποιείται αυτούσιος ο σειριακός κώδικας, καθώς δεν υπάρχει ανάγκη για συγχρονισμό και επανεκκίνηση λειτουργιών. Αυτό οφείλεται στο γεγονός ότι οι λειτουργίες της εισαγωγής και της διαγραφής εργάζονται σε αντίγραφα των κόμβων και έτσι οι λειτουργίες αναζήτησης μπορούν να διασχίζουν με ασφάλεια και συνέπεια το δέντρο.

Η λειτουργία της εισαγωγής ενός στοιχείου στο δέντρο ξεκινάει με την αναζήτηση του στοιχείου, που γίνεται όπως και στη λειτουργία της αναζήτησης χωρίς κάποιο συγχρονισμό. Η διάσχιση του δέντρου στην περίπτωση αυτή γίνεται με τη χρήση στοίβας, όπου αποθηκεύονται δείκτες που δείχνουν στους κόμβους που έχουμε περάσει. Αυτό γίνεται για να μπορούμε στη συνέχεια να διασχίσουμε το δέντρο προς την αντίθετη κατεύθυνση για την περίπτωση που χρειάζεται κάποια εξισορρόπηση (rebalance) στο δέντρο. Εάν το κλειδί που πρόκειται να εισαχθεί βρίσκεται ήδη στο δέντρο η λειτουργία επιστρέφει χωρίς να κάνει κάτι άλλο. Σε αντίθετη περίπτωση περνάμε στα επόμενα στάδια της λειτουργίας εισαγωγής που είναι η εισαγωγή του στοιχείου και η εξισορρόπηση του δέντρου. Εδώ, αυτό που γίνεται διαφορετικά απ' ότι στο σειριακό κώδικα είναι το ότι η εισαγωγή του στοιχείου γίνεται σε αντίγραφο του τμήματος του δέντρο που πρόκειται να τροποποιηθεί. Πριν γίνει δηλαδή οποιαδήποτε αλλαγή, η λειτουργία δημιουργεί ιδιωτικά αντίγραφα των κόμβων που πρόκειται να τροποποιήσει. Στο τέλος των δύο αυτών σταδίων έχουμε δύο δείκτες. Έναν δείκτη που δείχνει στο σημείου του δέντρου που θα εισαχθεί το αντίγραφο, το οποίο ονομάζεται σημείο σύνδεσης (connection point), και έναν δείκτη στη ρίζα του αντιγράφου που πρόκειται να συνδεθεί στο δέντρο. Στη συνέχεια απομένουν τα στάδια της επαλήθευσης και της αντικατάστασης, τα οποία, όπως σημειώσαμε και προηγουμένως, λαμβάνουν χώρα εντός ενός HTM transaction. Πριν αρχίσει το transaction ελέγχεται πόσες φορές έχει γίνει προσπάθεια επαλήθευσης και εάν έχει ξεπεραστεί ένας αριθμός προσπαθειών τότε η λειτουργία επανεκκινείται. Στο σημείο αυτό πρέπει να σημειώσουμε ότι η δομή έχει ένα κλείδωμα το οποίο όμως χρησιμοποιείται μόνο εφόσον το transaction αποτύχει περισσότερες από ένα συγκεκριμένο αριθμό φορών. Πριν ξεκινήσει το transaction ελέγχεται το κλείδωμα αυτό ώστε αν κάποια άλλη διεργασία/νήμα το έχει αποκτήσει η εκτέλεση να περιμένει μέχρι αυτό να απελευθερωθεί, αποφεύγοντας έτσι να γίνει abort το transaction. Στη συνέχεια ξεκινάει το transaction. Εάν αυτό ξεκινήσει επιτυχώς ελέγχεται πάλι το κλείδωμα. Εάν αυτό έχει κατοχυρωθεί από άλλη διεργασία/νήμα τότε το transaction γίνεται abort, ενώ σε αντίθετη περίπτωση γίνεται η επαλήθευση του προς αντικατάσταση τμήματος. Αν η επαλήθευση αποτύχει, τότε γίνεται abort. Στην περίπτωση που η επαλήθευση είναι επιτυχημένη, τότε γίνεται η αντικατάσταση του τμήματος στο δέντρο και το transaction γίνεται commit. Για τη διαχείριση των aborts ελέγχεται το είδος τους. Εάν το abort έχει προκύψει από αποτυχημένη επαλήθευση τότε η εισαγωγή επανεκκινείται. Σε αντίθετη περίπτωση, δηλαδή αν το abort οφείλεται σε σύγκρουση με άλλη διεργασία, δεν επανεκκινείται όλη η διεργασία, αλλά ξαναγίνεται προσπάθεια επαλήθευσης. Τέλος, αξίζει να τονίσουμε ότι αν η λειτουργία επαναληφθεί περισσότερες φορές από ένα συγκεκριμένο αριθμό φορών, τότε αυτή εκτελείται με τη χρήση κλειδώματος. Επίσης σημαντικό είναι να σημειωθεί ότι με την ανάγνωση του κλειδώματος αυτό εισέρχεται στο read set του transaction και έτσι εάν μία άλλη διεργασία/νήμα το κατοχυρώσει, τότε το transaction θα γίνει abort.

Η λειτουργία της διαγραφής είναι αντίστοιχη με αυτή της εισαγωγής όταν ο κόμβος που πρόκειται να διαγραφεί από το BST έχει λιγότερα από δύο παιδιά. Στην περίπτωση που ο κόμβος έχει δύο παιδιά η διαδικασία είναι λίγο πιο πολύπλοκη, καθώς τότε πρέπει να βρεθεί ο διάδοχος (successor) κόμβος, ο οποίος θα πάρει τη θέση του προς διαγραφή κόμβου. Για να αποφευχθούν προβληματικές καταστάσεις όταν διαγράφεται ένας κόμβος με δύο παιδιά, απαιτείται να αντιγράφεται ιδιωτικά όλο το μονοπάτι που ενώνει τον κόμβο

και τον διάδοχό του. Στο σχήμα 5.2 φαίνεται η διαγραφή ενός στοιχείου με RCU-HTM, ενώ ένα νήμα πραγματοποιεί τη λειτουργία της αναζήτησης, χωρίς να δημιουργείται κάποιο πρόβλημα.



Σχήμα 5.2. Η διαγραφή του στοιχείου 7 με χρήση RCU-HTM επιτρέπει στο νήμα T1 την αναζήτηση του στοιχείου 9 χωρίς την ανάγκη συγχρονισμού.

5.4 Εφαρμογή RCU-HTM σε B+ Δέντρα

Στην ενότητα αυτή θα εφαρμόσουμε την τεχνική RCU-HTM σε B+ Δέντρα και θα μιλήσουμε για τις λεπτομέρειες της υλοποίησης. Αρχικά πρέπει να τονίσουμε ότι για την υλοποίηση έχει χρησιμοποιηθεί η γλώσσα προγραμματισμού C. Στο σχήμα 5.3 φαίνεται η γενική μορφή ενός κόμβου του B+ Tree. Όπως φαίνεται ένας κόμβος μπορεί να είναι είτε ρίζα, είτε εσωτερικός κόμβος, είτε φύλλο, κάτι το οποίο φαίνεται από τη μεταβλητή type. Ακόμα, ένας κόμβος έχει έναν πίνακα όπου περιέχει τα κλειδιά που βρίσκονται στον κόμβο, έναν πίνακα με δείκτες, οι οποίοι δείχνουν είτε στα παιδιά του κόμβου αν αυτός είναι εσωτερικός είτε στα αποθηκευμένα δεδομένα εάν πρόκειται για φύλλο, καθώς και μία μεταβλητή που δείχνει τον αριθμό των κλειδιών που βρίσκονται αποθηκευμένα στον κόμβο.

```
typedef struct nodet{

    int type; //root=0 (-1 when the only node), internal node=1, leaf=2
    int keys[b-1];
    void *children[b]; //points to children if internal, points to data and next if leaf
    int n; //number of keys

} node;
```

Σχήμα 5.3. *Ένας κόμβος του B+ Tree σε γλώσσα προγραμματισμού C.*

Η λειτουργία της αναζήτησης ενός στοιχείου στο B+ δέντρο με RCU-HTM δε διαφέρει ιδιαίτερα απ' ότι είδαμε στην προηγούμενη ενότητα, όπου μιλήσαμε για την τεχνική RCU-HTM σε δέντρα δυαδικής αναζήτησης. Συγκεκριμένα, ξεκινώντας από τη ρίζα του δέντρου και χρησιμοποιώντας τους εσωτερικούς κόμβους δρομολόγησης ψάχνουμε το φύλλο, στο οποίο θα βρίσκεται αν υπάρχει το κλειδί που ψάχνουμε. Αφού βρούμε το φύλλο ελέγχουμε αν αυτό περιέχει το κλειδί. Η λειτουργία αναζήτησης είναι

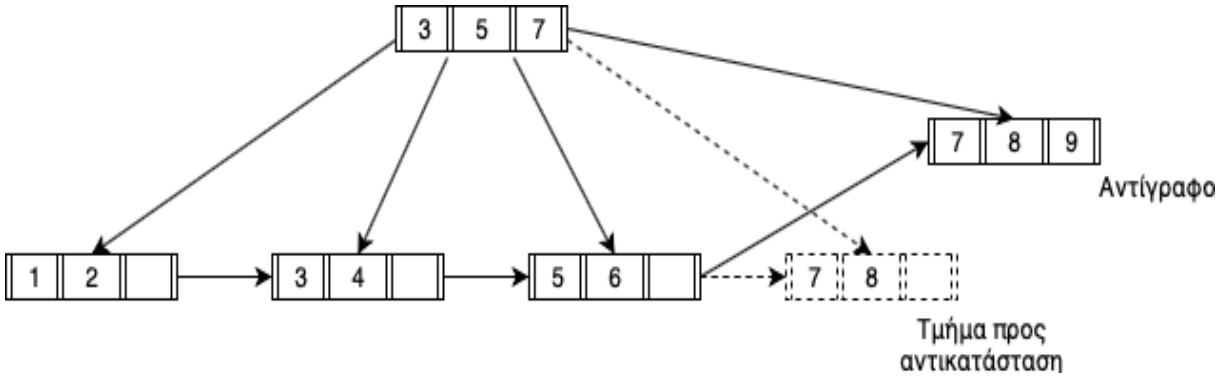
στην ουσία ίδια με αυτή μίας σειριακής υλοποίησης ενός B+ δέντρου, καθώς όπως είπαμε και προηγουμένως στην τεχνική RCU-HTM δεν απαιτείται συγχρονισμός για την αναζήτηση ενός στοιχείου στο δέντρο.

Η λειτουργία της εισαγωγής ενός στοιχείου στο B+ δέντρο έχει μερικά επιπλέον στοιχεία από αυτή σε ένα δέντρο δυαδικής αναζήτησης, λόγω της ιδιαίτερης φύσης του B+ δέντρου. Ξεκινάμε από τη ρίζα του δέντρου ψάχνοντας το φύλλο που πρέπει να εισαχθεί το κλειδί. Κατά την αναζήτηση κρατάμε δείκτες στους κόμβους που διασχίζουμε ώστε να μπορούμε να διασχίσουμε αντίστροφα το συγκεκριμένο μονοπάτι του δέντρου κατά την εισαγωγή, καθώς και αντίγραφα αυτών, τα οποία θα χρησιμεύσουν κατά το στάδιο του validation. Λόγω του ότι στο B+ δέντρο τα φύλλα είναι μεταξύ τους συνδεδεμένα, πρέπει να κρατήσουμε δείκτες τόσο στο προηγούμενο όσο και στο επόμενο φύλλο από αυτό που θα γίνει η εισαγωγή, ώστε να μπορούμε κατά το στάδιο του validation να εξακριβώσουμε ότι αυτά δεν έχουν αλλάξει, αλλά και να συνδέσουμε το φύλλο με αυτά. Τον δείκτη στο επόμενο φύλλο τον παίρνουμε εύκολα από το πεδίο children[b-1] που σε κάθε φύλλο δείχνει στο επόμενό του. Για να πάρουμε τον δείκτη στο προηγούμενο φύλλο χρησιμοποιούμε τη συνάρτηση findPreviousLeaf, η οποία αν χρειαστεί διασχίζει το δέντρο αντίστροφα και βρίσκει το προηγούμενο φύλλο. Ακόμα έχουμε τις βοηθητικές συναρτήσεις findLeftSibling και findRightSibling, οι οποίες βρίσκουν αντίστοιχα τον αριστερά και δεξιά αδελφικό κόμβο ενός οποιοδήποτε κόμβου του δέντρου, εάν αυτός υπάρχει.

Στη συνέχεια προχωράμε με την εισαγωγή του στοιχείου σε αντίγραφο του φύλλου. Αν χρειαστεί να σπάσουμε το φύλλο σε δύο καινούργια, τότε, αφού δημιουργήσουμε τα δύο φύλλα, θα χρειαστεί να πάρουμε αντίγραφο και του πατρικού κόμβου του φύλλου, στον οποίο θα προστεθούν τα δύο νέα φύλλα ως παιδιά αντί του παλιού. Η διαδικασία αυτή μπορεί να επαναληφθεί έως ότου οι δύο κόμβου να χωράνε στον πατρικό τους κόμβο. Αν φτάσουμε στη ρίζα και δε χωράνε σε αυτή και οι δύο κόμβοι, τότε σπάει η ρίζα και δημιουργούνται δύο νέοι εσωτερικοί κόμβοι, οι οποίοι θα έχουν ως πατρικό κόμβο μία νέα ρίζα, της οποίας θα είναι τα μοναδικά παιδιά. Η διαδικασία αυτή είναι ίδια με αυτή του σειριακού αλγορίθμου και έχει δοθεί και σχηματικά στην ενότητα 2.5.2 (σχήμα 2.7).

Πλέον έχουμε ένα δείκτη στη ρίζα του τμήματος που πρόκειται να εισάγουμε στο δέντρο και στο οποίο βρίσκεται το εισαχθέν κλειδί, ένα δείκτη στο φύλλο του τμήματος αυτού, καθώς και δείκτες στα σημεία του δέντρου που θα ενωθούν οι δύο αυτοί κόμβοι. Προτού όμως αντικαταστήσουμε το παλιό τμήμα του δέντρου με το τροποποιημένο αντίγραφο θα πρέπει να περάσουμε από το στάδιο του validation, να ελέγξουμε δηλαδή ότι το τμήμα που πρόκειται να αντικαταστήσουμε δεν έχει μεταβληθεί με οποιοδήποτε τρόπο. Το ίδιο θα πρέπει να ισχύει και για τα σημεία στα οποία πρέπει να συνδέσουμε το τροποποιημένο αντίγραφο. Στο σημείο αυτό ξεκινάμε το transaction για το οποίο ισχύει ό,τι αναλύθηκε στην προηγούμενη ενότητα. Ξεκινώντας το validation διασχίζουμε το δέντρο από τη ρίζα του, εντοπίζοντας πάλι το φύλλο στο οποίο θα βρίσκεται, αν υπάρχει, το προς εισαγωγή κλειδί, καθώς και το προηγούμενο και επόμενο από αυτό φύλλα. Στη συνέχεια ελέγχουμε ότι όλοι οι κόμβοι που πρόκειται να αντικατασταθούν δεν έχουν μεταβληθεί, ενώ το ίδιο κάνουμε και για τους κόμβους που θα συνδέσουμε το αντίγραφο. Εάν βρεθεί κάποια αλλαγή, τότε γίνεται abort, το οποίο χειριζόμαστε όπως είδαμε στην προηγούμενη ενότητα με αποτέλεσμα να επανεκκινηθεί η λειτουργία εισαγωγής. Στην περίπτωση επιτυχούς validation συνδέουμε το αντίγραφο στο δέντρο, αντικαθιστώντας έτσι ένα κομμάτι του δέντρου με τη νέα ενημερωμένη έκδοσή του, η οποία περιλαμβάνει το προς εισαγωγή κλειδί. Τέλος, κάνουμε commit το transaction και η λειτουργία εισαγωγής ολοκληρώνεται επιτυχώς. Στο σχήμα 5.4 βλέπουμε την εισαγωγή του στοιχείου 9 σε ένα B+ δέντρο τάξης 4 με χρήση της τεχνικής RCU-HTM. Συγκεκριμένα με

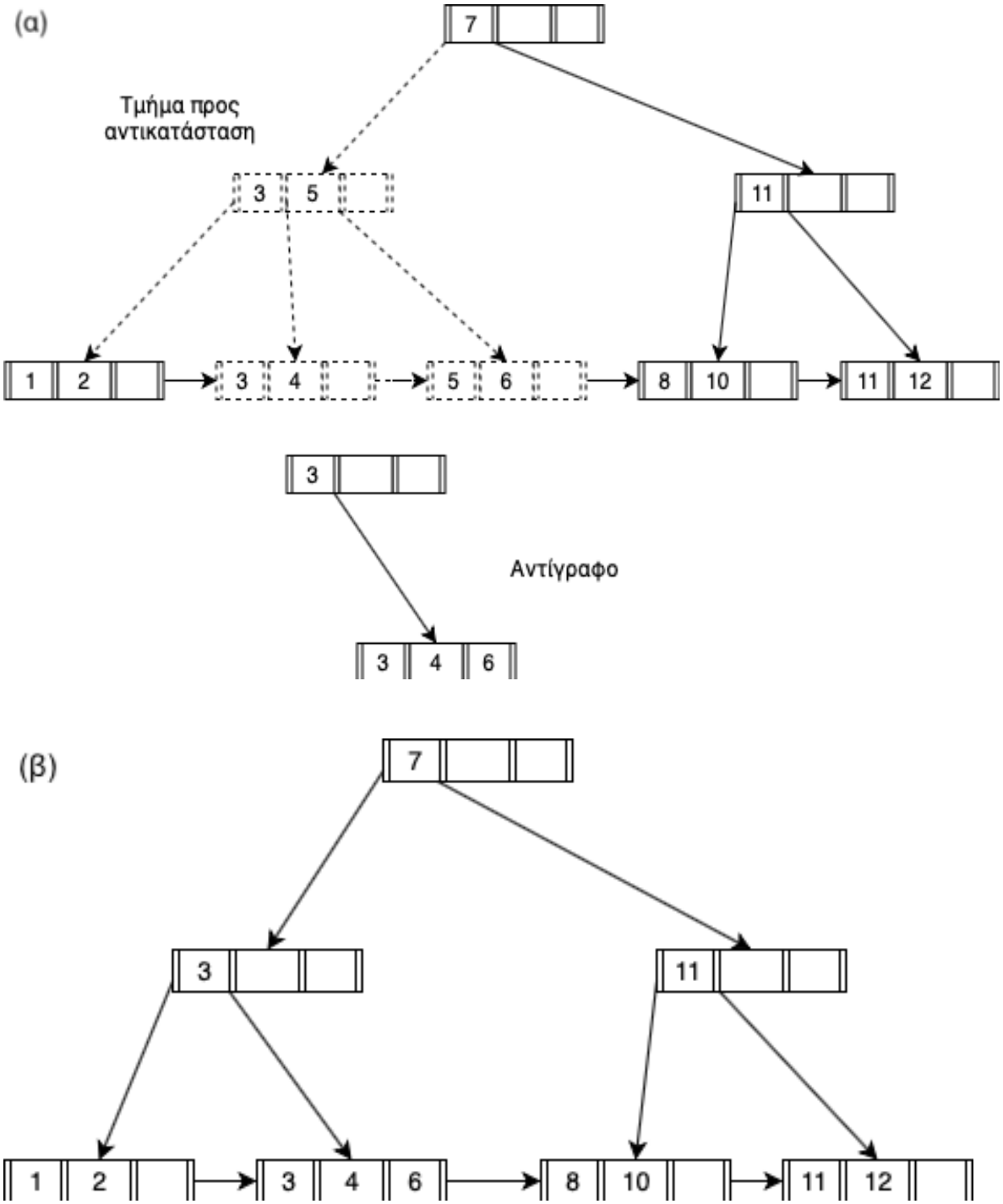
διακεκομμένη γραμμή εμφανίζεται το τμήμα του δέντρου που αντικαθίσταται, ενώ με κανονική γραμμή φαίνεται το δέντρο, όπως αυτό θα είναι μετά την ολοκλήρωση της λειτουργίας της εισαγωγής.



Σχήμα 5.4. *Η εισαγωγή του στοιχείου 9 σε B+ δέντρο 4^{ης} τάξης με χρήση της τεχνικής RCU-HTM.*

Η λειτουργία της διαγραφής είναι παρόμοια με αυτή της εισαγωγής. Η διαφορά έγκειται στο γεγονός ότι κατά τη διαγραφή ενός στοιχείου από ένα B+ δέντρο μπορεί να έχουμε συνενώσεις κόμβων ή μεταφορά στοιχείων από έναν κόμβο σε έναν άλλο. Στις περιπτώσεις αυτές πρέπει εκτός από αντίγραφα των κόμβων αυτών, τα οποία θα τους αντικαταστήσουν στο δέντρο, να κρατάμε και αντίγραφά τους για την επαλήθευσή τους κατά το στάδιο του validation. Ιδιαίτερα όταν αυτά είναι φύλλα θα πρέπει να εντοπίσουμε είτε το προηγούμενο είτε το επόμενο από αυτά φύλλο, ανάλογα αν γίνεται συνένωση με τον αριστερά ή δεξιά αδελφικό κόμβο, ώστε να γίνει σωστά το validation και η αντικατάσταση του κρίσιμου τμήματος του δέντρου.

Στην ενότητα 2.5.2 (σχήμα 2.8) είχαμε δει τη διαγραφή ενός στοιχείου από ένα B+ δέντρο χρησιμοποιώντας τον σειριακό αλγόριθμο. Στο σχήμα 5.5 φαίνεται η διαγραφή του στοιχείου 5 από ένα B+ δέντρο χρησιμοποιώντας την τεχνική RCU-HTM. Όπως έχουμε ήδη δει, όταν διαγράφεται ένα στοιχείο από ένα φύλλο του δέντρου, τότε πρέπει να ελέγξουμε εάν το φύλλο παραμένει τουλάχιστον μισογεμάτο. Στην περίπτωση που το φύλλο έχει τον απαραίτητο αριθμό στοιχείων, τότε δε χρειάζεται να κάνουμε κάτι άλλο και προχωράμε στα στάδια του validation και της αντικατάστασης. Σε αντίθετη περίπτωση όμως πρέπει να προχωρήσουμε σε κάποια εξισορρόπηση. Για να γίνει αυτό μπορούμε να μεταφέρουμε κάποιο κλειδί από αδερφικό φύλλο, εάν αυτό είναι περισσότερο από μισογεμάτο. Εάν όλα (ένα ή δύο) τα αδερφικά φύλλα είναι ακριβώς μισογεμάτα, τότε προχωράμε σε συνένωση δύο φύλλων, ώστε να πληρούνται οι απαραίτητες απαιτήσεις ενός B+ δέντρου. Στο παράδειγμα του σχήματος 5.5 βλέπουμε ότι ως συνέπεια των προηγούμενων και μετά την διαγραφή του στοιχείου 5 από το φύλλο, αυτό συνενώνεται με το αριστερά του. Συγκεκριμένα στο (α) βλέπουμε το δέντρο πριν τη διαγραφή του στοιχείου 5, ενώ με διακεκομμένη γραμμή φαίνεται το τμήμα του δέντρου που πρόκειται εν τέλει να αντικατασταθεί. Επίσης βλέπουμε το αντίγραφο που έχει δημιουργηθεί σύμφωνα με την τεχνική RCU-HTM και το οποίο πρόκειται να εισαχθεί στο δέντρο. Τέλος, στο (β) βλέπουμε το αποτέλεσμα της διαγραφής του στοιχείου 5.

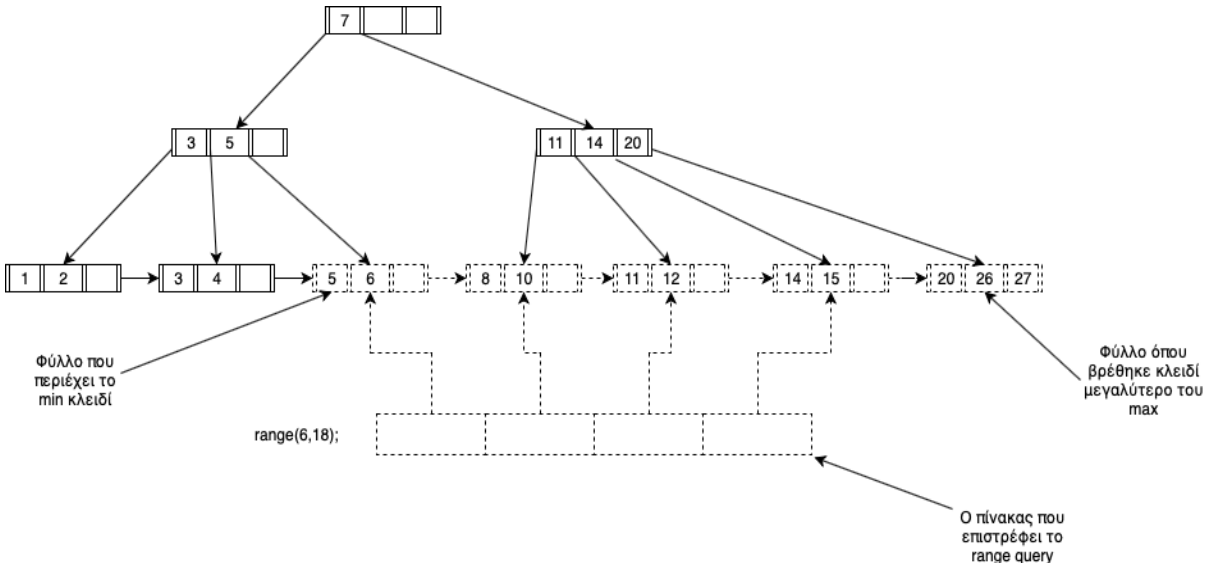


Σχήμα 5.5. *Η διαγραφή του κλειδιού 5 από ένα B+ δέντρο με χρήση της τεχνικής RCU-HTM. (α) Το B+ δέντρο πριν τη διαγραφή και το προς εισαγωγή αντίγραφο. (β) Το B+ δέντρο μετά την ολοκλήρωση της λειτουργίας.*

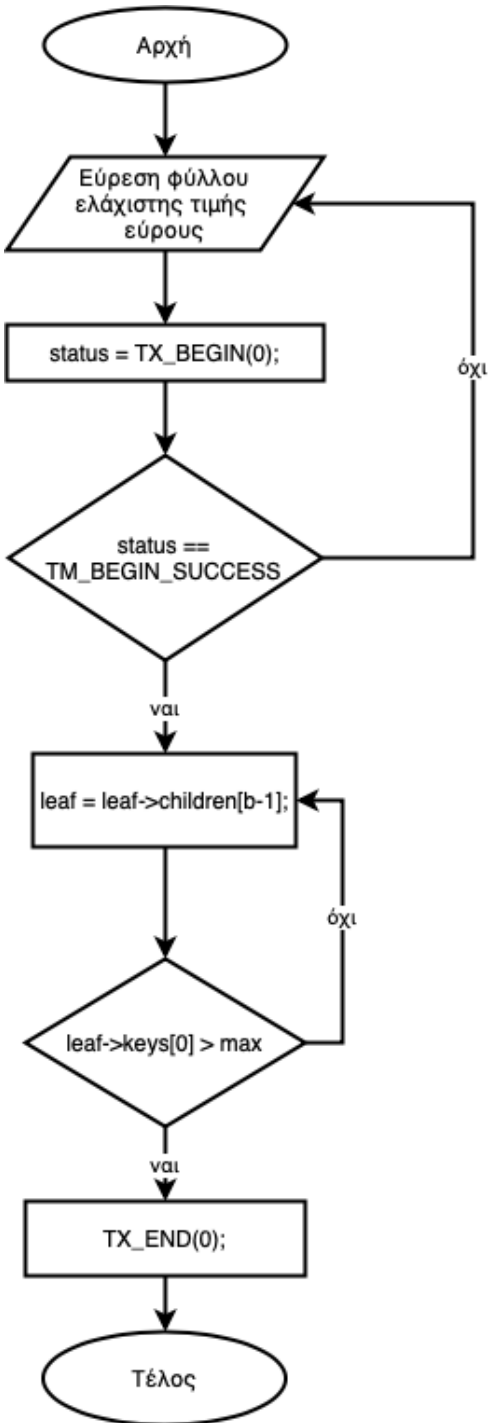
Η τελευταία λειτουργία των B+ δέντρων που θα υλοποιήσουμε με χρήση της τεχνικής RCU-HTM είναι αυτή της αναζήτησης εύρους (range query), η οποία είναι και το βασικό ερευνητικό πεδίο αυτής της διπλωματικής εργασίας. Όπως είδαμε και προηγουμένως, range query ονομάζεται η εύρεση των κλειδιών ενός δέντρου, τα οποία ανήκουν εντός ενός δοσμένου εύρους τιμών [min, max]. Το range query ξεκινάει με την αναζήτηση του φύλλου όπου θα βρίσκεται αν υπάρχει το κλειδί min. Αυτό γίνεται, όπως και στις άλλες λειτουργίες, χωρίς την ανάγκη ύπαρξης συγχρονισμού.

Αφού βρούμε το φύλλο με το μικρότερο κλειδί εντός του εύρους, ξεκινάμε το transaction. Εφόσον το transaction ξεκινήσει επιτυχώς ελέγχουμε εάν κάποιο άλλο νήμα κατέχει το καθολικό κλείδωμα του δέντρου, ενώ αυτό ισχύει κάνουμε abort. Σε αντίθετη περίπτωση η λειτουργία του range query συνεχίζεται με τη διάσχιση των φύλλων του δέντρου. Συγκεκριμένα, ξεκινάμε από το φύλλο που έχουμε εντοπίσει προηγουμένως και διασχίζουμε ένα-ένα τα φύλλα με τη σειρά έως ότου φτάσουμε σε κάποιο φύλλο όπου δεν περιέχει κλειδί εντός του ζητούμενου εύρους. Για να το βρούμε αυτό ελέγχουμε το πρώτο κλειδί κάθε φύλλου. Εάν αυτό είναι μεγαλύτερο του κλειδιού max, τότε σταματάμε τη διάσχιση του φύλλου και κάνουμε commit το transaction. Αξίζει να σημειώσουμε ότι για κάθε φύλλο το οποίο περιέχει κλειδιά εντός του ζητούμενου εύρους κρατάμε έναν δείκτη σε αυτό. Έτσι, στο τέλος του range query έχουμε έναν πίνακα από δείκτες στα φύλλα του B+ δέντρου που περιέχουν κλειδιά εντός του εύρους [min, max]. Στη συνέχεια μπορούμε να ανατρέξουμε στον πίνακα αυτό χωρίς την ανάγκη κάποιου συγχρονισμού και να πάρουμε το αποτέλεσμα με όποια μορφή θέλουμε. Μπορούμε για παράδειγμα να μετρήσουμε τα κλειδιά εντός του εύρους, να τα τυπώσουμε ή απλά να επιστρέψουμε μία εκ των τιμών 1 και 0 ανάλογα με το αν εντοπίσαμε κλειδιά εντός του ζητούμενου εύρους ή όχι αντίστοιχα.

Στην περίπτωση που γίνει abort το transaction η λειτουργία του range query επανεκκινείται. Όπως και στις υπόλοιπες λειτουργίες, εάν το range query επανεκκινηθεί περισσότερες από έναν προκαθορισμένο αριθμό φορών, τότε το range query εκτελείται με τη χρήση κλειδώματος. Στο σχήμα 5.6 βλέπουμε ένα παράδειγμα εκτέλεσης range query με χρήση του αλγορίθμου RCU-HTM. Συγκεκριμένα με διακεκομμένη γραμμή εμφανίζονται τα φύλλα τα οποία επισκεφτήκαμε εντός του transaction, καθώς και ο πίνακας με δείκτες ο οποίος είναι το αποτέλεσμα της εκτέλεσης. Στο σχήμα 5.7 βλέπουμε ένα διάγραμμα ροής των range queries.



Σχήμα 5.6. *Παράδειγμα range query με RCU-HTM.*



Σχήμα 5.7. *Διάγραμμα ροής των λειτουργιών αναζήτησης εύρους.*

Κεφάλαιο 6
Πειραματική αξιολόγηση

6.1 Γενικά

Στο κεφάλαιο αυτό θα ασχοληθούμε με την αξιολόγηση της τεχνικής που προτείναμε στο προηγούμενο κεφάλαιο, δηλαδή την εφαρμογή RCU-HTM σε B+-Δέντρα για την υλοποίηση παράλληλων λειτουργιών αναζήτησης εύρους. Για την αξιολόγηση θα πραγματοποιήσουμε μία σειρά από διαφορετικές μετρήσεις στις οποίες θα υποβληθεί τόσο ο αλγόριθμος που αναπτύξαμε, όσο και μερικοί ακόμα υπάρχοντες αλγόριθμοι με τους οποίους και θα συγκριθεί. Η σύγκριση αυτή θα μας επιτρέψει να βγάλουμε συμπεράσματα για την επίδοση της τεχνικής RCU-HTM σε B+-Δέντρα.

6.2 Χαρακτηριστική του συστήματος

Για να εξάγουμε τις μετρήσεις που πρόκειται να παρουσιάσουμε στη συνέχεια χρησιμοποιήθηκε μία 28-πύρηνη πλατφόρμα NUMA αρχιτεκτονικής με τα εξής χαρακτηριστικά:

- 2 sockets (Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz)
- 14 πυρήνες ανά socket (28 νήματα με hyperthreading)
- 32KB L1 data cache ανά πυρήνα
- 32KB L1 instruction cache ανά πυρήνα
- 256KB L2 cache ανά πυρήνα
- 35MB L3 cache ανά socket
- 128GB RAM
- Hardware Transactional Memory:
    - Lazy data versioning
    - Eager conflict detection
    - Best effort HTM
    - Strong isolation
    - Cache line granularity
    - 4MB read set
    - 22KB read set

6.3 Αποτελέσματα

Στην ενότητα αυτή θα παρουσιάσουμε τα αποτελέσματα από τις μετρήσεις που πραγματοποιήσαμε, τα οποία και θα σχολιάσουμε. Συγκεκριμένα θα συγκρίνουμε την απόδοση της τεχνικής RCU-HTM σε B+-Δέντρα με τις υλοποιήσεις που παρουσιάσαμε στις ενότητες 4.3 και 4.5, δηλαδή με την υλοποίηση που χρησιμοποιεί CA-Trees, καθώς και αυτή που κάνει χρήση k-ary Search Trees. Επίσης στη σύγκριση θα προσθέσουμε και την υλοποίηση της τεχνική RCU σε B+-Δέντρα, η οποία προέκυψε ως ενδιάμεση υλοποίηση κατά την πραγματοποίηση της παρούσας διπλωματικής εργασίας. Αξίζει να σημειωθεί ότι ως ακολουθιακή δομή για το CA-Tree θα χρησιμοποιηθεί ένα Treap, το οποίο είναι ένα self-balancing δυαδικό δέντρο αναζήτησης με αποτελεσματικές λειτουργίες split και join, ενώ για το B+-Δέντρο στην τεχνική RCU-HTM θα δοκιμάσουμε

διαφορετικές τιμές για την τάξη m και θα παρουσιάσουμε τα αντίστοιχα αποτελέσματα. Συγκεκριμένα θεωρούμε ως τιμή του m που δίνει ένα αποδοτικό αποτέλεσμα σε όλα τα σενάρια την τιμή 16 και ανάλογα τις απαιτήσεις της κάθε μέτρησης βλέπουμε πως διαφορετικές τιμές για το m μπορούν να αυξήσουν ακόμα περισσότερο την απόδοση.

Για να μπορέσουμε να εξάγουμε ασφαλή και πλήρη συμπεράσματα μελετήσαμε ένα πλήθος διαφορετικών παραγόντων που μπορούν να επηρεάσουν την απόδοση κάθε αλγορίθμου. Ένας σημαντικός παράγοντας είναι το μέγεθος της προς αξιολόγηση δομής, δηλαδή το πλήθος των κλειδιών που μπορούν να αποθηκευτούν σε αυτή. Έτσι, θα παρουσιάσουμε αποτελέσματα για τρία διαφορετικά μέγιστα μεγέθη δομών και συγκεκριμένα για max key = 10000, 1000000 και 10000000 κλειδιά. Όπως ενδείκνυται, για κάθε μέτρηση θα ξεκινάμε με τη δομή ακριβώς μισογεμάτη.

Ένας άλλος παράγοντας που λάβαμε υπόψη είναι τα ποσοστά των λειτουργιών που θα εκτελεστούν. Έτσι, πραγματοποιήσαμε μετρήσεις για τρεις διαφορετικές καταστάσεις:

- 10% εισαγωγή/διαγραφή κλειδιού – 40% αναζήτηση κλειδιού – 50% range queries
- 24% εισαγωγή/διαγραφή κλειδιού – 50% αναζήτηση κλειδιού – 26% range queries
- 50% εισαγωγή/διαγραφή κλειδιού – 35% αναζήτηση κλειδιού – 15% range queries

Αξίζει να σημειωθεί ότι τα ποσοστά για τις λειτουργίες εισαγωγής και διαγραφής κλειδιού διαιρούνται ακριβώς στις δύο αυτές λειτουργίες. Έτσι, το μέγεθος της δομής δε θα μεταβάλλεται σημαντικά κατά την πραγματοποίηση της εκάστοτε μέτρησης. Επίσης πραγματοποιήθηκαν και θα παρουσιαστούν μετρήσεις όπου το 100% των λειτουργιών που εκτελούνται στη δομή είναι λειτουργίες αναζήτησης εύρους, ούτως ώστε να μπορέσουμε να συγκρίνουμε τη δυναμική του αλγορίθμου αποκλειστικά στις λειτουργίες αυτές, το οποίο είναι και το θέμα της παρούσας διπλωματικής εργασίας.

Ακόμη μελετήθηκε ως παράγοντας που επηρεάζει την απόδοση του αλγορίθμου και το εύρος των range queries. Έτσι, για κάθε μέγεθος δομής δεδομένων και αναλογία λειτουργιών πραγματοποιήθηκαν μετρήσεις για διαφορετικές τιμές εύρους. Τέλος, ώστε να παρατηρηθεί η κλιμακωσιμότητα του αλγορίθμου, πραγματοποιήθηκαν μετρήσεις χρησιμοποιώντας 1, 2, 4, 8, 16, 28 και 56 threads.

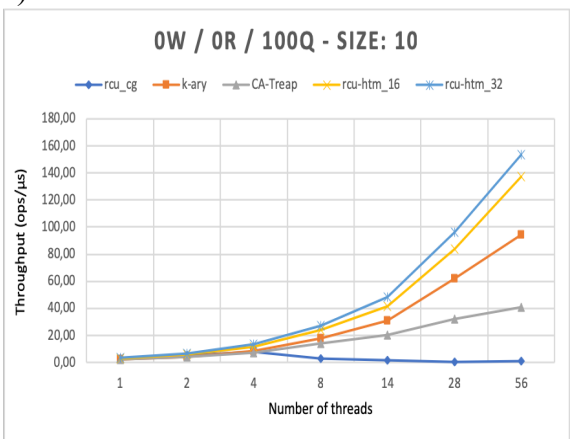
Στο σημείο αυτό πρέπει να σημειώσουμε ότι το μετρούμενο μέγεθος απόδοσης (throughput) είναι το πλήθος των λειτουργιών που εκτελούνται ανά μικροδευτερόλεπτο (ops/μs). Κάθε μέτρηση διήρκησε 5 δευτερόλεπτα, ενώ η τιμές throughput που παρουσιάζονται στη συνέχεια προέκυψαν παίρνοντας τον μέσο όρο πέντε μετρήσεων, ώστε να εξασφαλισθεί η ακρίβεια των αποτελεσμάτων. Τα αποτελέσματα θα παρουσιαστούν σε διαγράμματα, τα οποία θα δείχνουν τη μεταβολή του throughput σε συνάρτηση είτε με τον αριθμό των χρησιμοποιούμενων νημάτων είτε με το εύρος των range queries. Σε κάθε διάγραμμα ο τίτλος θα είναι της μορφής "xW / yR / zQ – fixed value", όπου το x θα είναι το ποσοστό των εισαγωγών/διαγραφών, το y το ποσοστό των λειτουργιών αναζήτησης και το z το ποσοστό των range queries της εκάστοτε μέτρησης. Στη θέση του fixed value θα βρίσκεται η τιμή κάποιου παράγοντα που παραμένει σταθερός για όλες τις τιμές του διαγράμματος και ο οποίος θα είναι είτε το εύρος των range queries είτε ο αριθμός των threads που χρησιμοποιούνται.

6.3.1 Για max key = 10^6

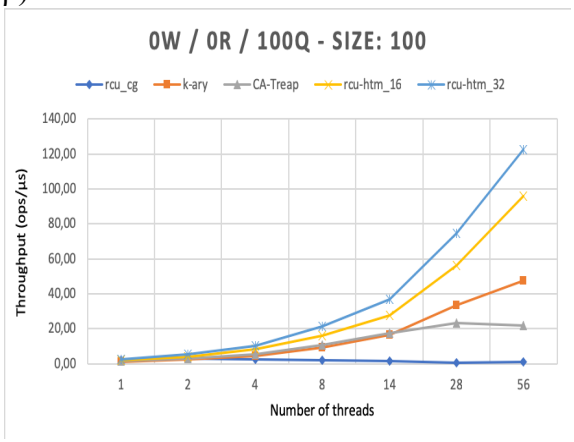
Αρχικά θα παρουσιάσουμε και θα σχολιάσουμε τα αποτελέσματα, στα οποία κάθε δομή θα μπορεί να έχει το πολύ 10^6 κλειδιά, μέγεθος το οποίο θεωρούμε και ως αντιπροσωπευτικότερο, ενώ, όπως ήδη αναφέρθηκε, αρχικά η δομή θα είναι μισογεμάτη,

δηλαδή θα έχει ακριβώς $5*10^5$ κλειδιά. Στις μετρήσεις αυτές για το B+-Δέντρο θα χρησιμοποιήσουμε m=16 και m=32. Όπως θα φανεί και στα αποτελέσματα, θεωρούμε την τιμή m=32 ως ιδανική για το συγκεκριμένο μέγεθος δέντρου.

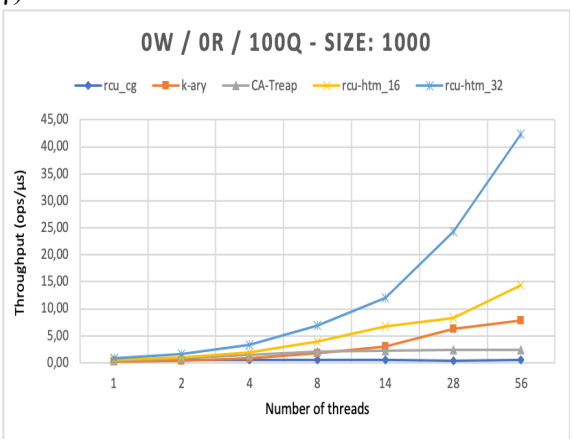
α)



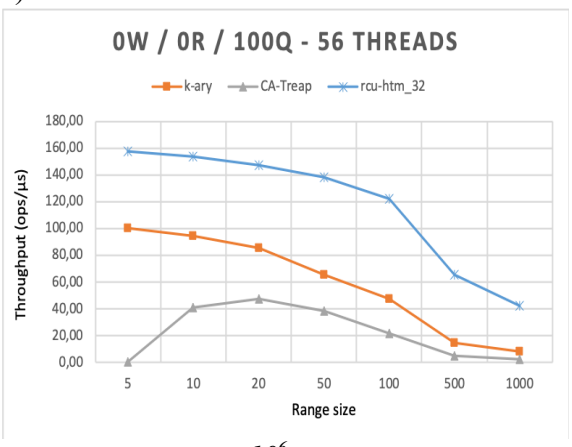
β)



γ)



δ)

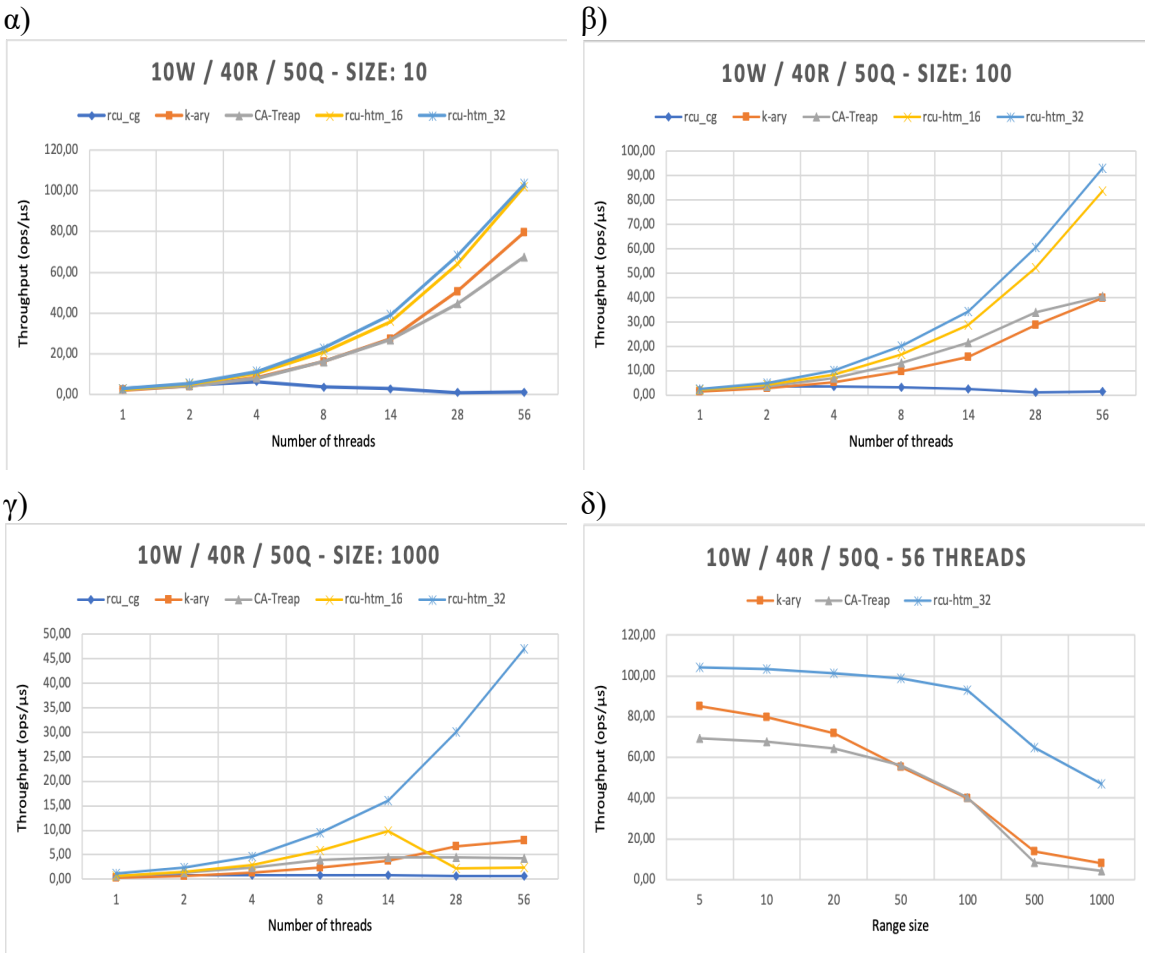


Σχήμα 6.1. *Αξιολόγηση αλγορίθμου για 100% range queries και max=10^6.*

Στο σχήμα 6.1 βλέπουμε συγκριτικά διαγράμματα που αναδεικνύουν την απόδοση του αλγορίθμου μας για λειτουργίες αναζήτησης εύρους. Όπως φαίνεται από τα σχήματα 6.1α-γ η τεχνική RCU-HTM παρουσιάζει σημαντικά καλύτερη κλιμακωσιμότητα από τις υπόλοιπες τεχνικές για διαφορετικές τιμές εύρους των range queries. Επίσης, είναι σημαντικό ότι η κλιμακωσιμότητα για τα range queries δεν επηρεάζεται σημαντικά για διαφορετικές τιμές εύρους. Αυτό είναι λογικό, καθώς η τεχνική RCU-HTM δεν προσθέτει σημαντικό κόστος συγχρονισμού στα range queries, παρά μόνο το κόστος εκτέλεσης του transaction. Στο συγκεκριμένο μάλιστα σενάριο, δεν αναμένεται να έχουμε conflicts που θα οδηγήσουν σε aborts και άρα επαναλήψεις στις εκτελέσεις των λειτουργιών. Σταθερή σχετικά κλιμακωσιμότητα φαίνεται να προσφέρει και το k-ST, το οποίο υστερεί όμως σε απόδοση. Αντίθετα, στο CA-Tree, βλέπουμε πως η κλιμακωσιμότητα μειώνεται σημαντικά με την αύξηση του εύρους των range queries, γεγονός που την καθιστά μη αποδοτική για χρήση σε περιπτώσεις όπου πραγματοποιούνται range queries μεγάλου εύρους.

Στο σχήμα 6.1δ εξετάζουμε τις δομές μετρώντας την απόδοσή τους συναρτήσει τους εύρους των range queries χρησιμοποιώντας όλα τα διαθέσιμα από την πλατφόρμα threads, δηλαδή 56. Με το διάγραμμα αυτό γίνεται σαφές πως η τεχνική RCU-HTM παρέχει αποδοτικότερα range queries ανεξαρτήτως του εύρους αυτών. Επιπλέον είναι

σημαντικό πως η μείωση που επέρχεται με την αύξηση του εύρους είναι μικρότερη στον αλγόριθμο που προτείνουμε σε σχέση με τις άλλες δομές που εξετάζονται. Αξίζει να σημειωθεί ότι για αρκετές τιμές εύρους από τις εξεταζόμενες η χρήση της τεχνικής RCU-HTM έχει ως αποτέλεσμα υπερδιπλάσια απόδοση απ' ότι οι ανταγωνιστές της.



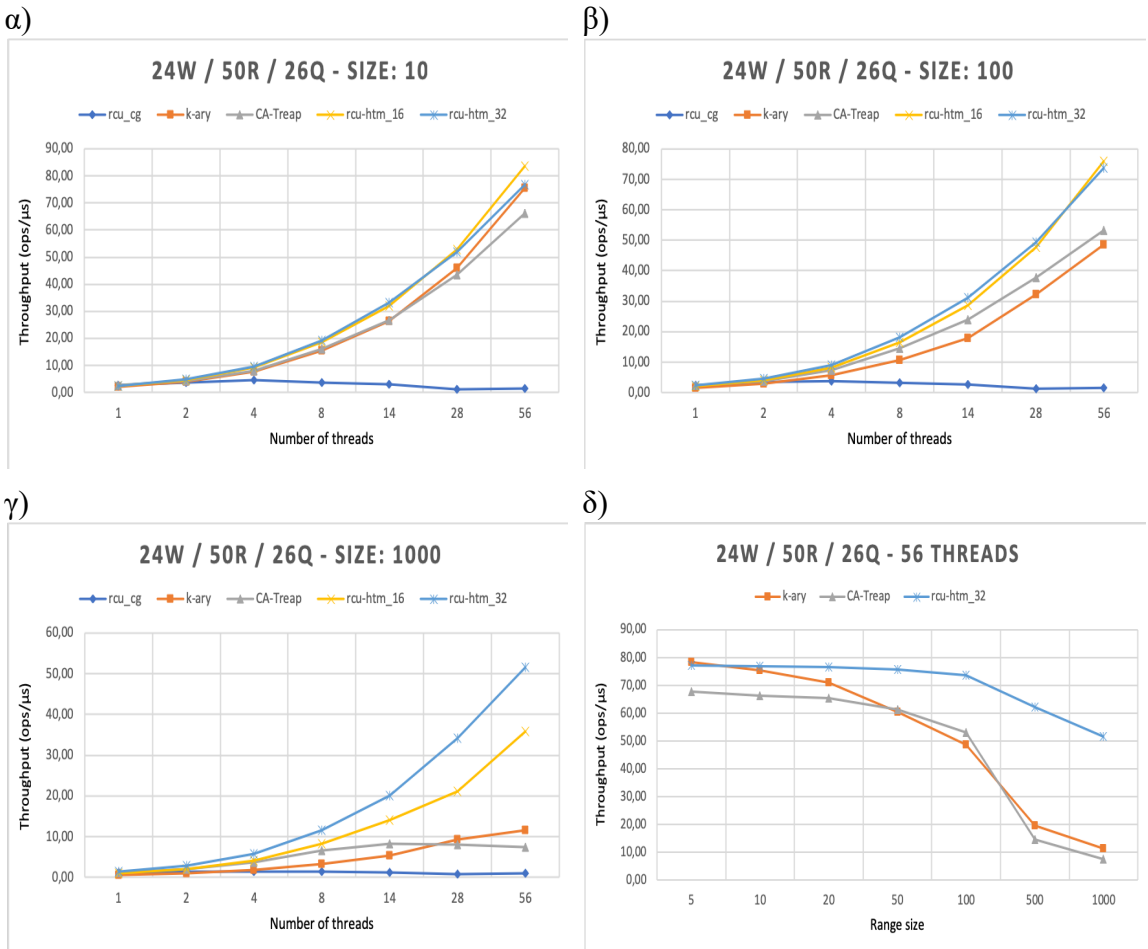
Σχήμα 6.2. *Αξιολόγηση αλγορίθμου για 10W/40R/50Q και max=10^6.*

Στο σχήμα 6.2 βλέπουμε τα αποτελέσματα μετρήσεων για ένα σενάριο εκτέλεσης όπου το 10% των λειτουργιών που εκτελούνται είναι είτε εισαγωγή είτε διαγραφή κάποιου κλειδιού από τη δομή, το 40% αφορά την αναζήτηση ενός κλειδιού, ενώ το υπόλοιπο 50% πρόκειται για λειτουργίες αναζήτησης εύρους. Είναι, λοιπόν, ένα σενάριο που αποτελείται κατά βάση από range queries, έχοντας παράλληλα σημαντικό ποσοστό εισαγωγών και διαγραφών. Στο σχήμα 6.1α βλέπουμε ότι οι τρεις υλοποιήσεις έχουν αντίστοιχη κλιμακωσιμότητα, με την τεχνική RCU-HTM να ξεχωρίζει. Στο σχήμα 6.2β βλέπουμε ότι η κλιμακωσιμότητα της RCU-HTM παραμένει καλή, ενώ των υπόλοιπων υλοποιήσεων πέφτει σημαντικά καθώς αυξάνεται το εύρος των range queries.

Στο διάγραμμα του σχήματος 6.2γ παρατηρείται μία σημαντική διαφορά μεταξύ των δύο διαφορετικών B+-Δέντρων στα οποία χρησιμοποιείται η τεχνική RCU-HTM. Συγκεκριμένα το δέντρο με m=32 παρουσιάζει πολύ καλύτερη κλιμακωσιμότητα, σε αντίθεση με αυτό που έχει m=16, του οποίου η απόδοση πέφτει με την αύξηση των threads από 14 σε 28. Αυτό οφείλεται στο γεγονός ότι για m=32 έχουμε μεγαλύτερα φύλλα και άρα προκύπτει πιο σπάνια η ανάγκη να σπάσουμε ή να συνενώσουμε φύλλα, κάτι που θα οδηγούσε σε περισσότερα aborts και άρα μικρότερη απόδοση, όπως και γίνεται στην

περίπτωση όπου m=16. Αυτό φαίνεται να συμβαίνει μόνο στην περίπτωση που εξετάζουμε range queries μεγάλου εύρους, καθώς σε αυτή την περίπτωση εμπλέκονται σημαντικά περισσότερα φύλλα στο transaction. Ένας ακόμα λόγος είναι το ότι για m=32 χρειάζεται να εξετάσουμε λιγότερα φύλλα απ' ότι για m=16 -περίπου τα μισά- και άρα θα χρειαστεί λιγότερος χρόνος για την προσπέλασή τους.

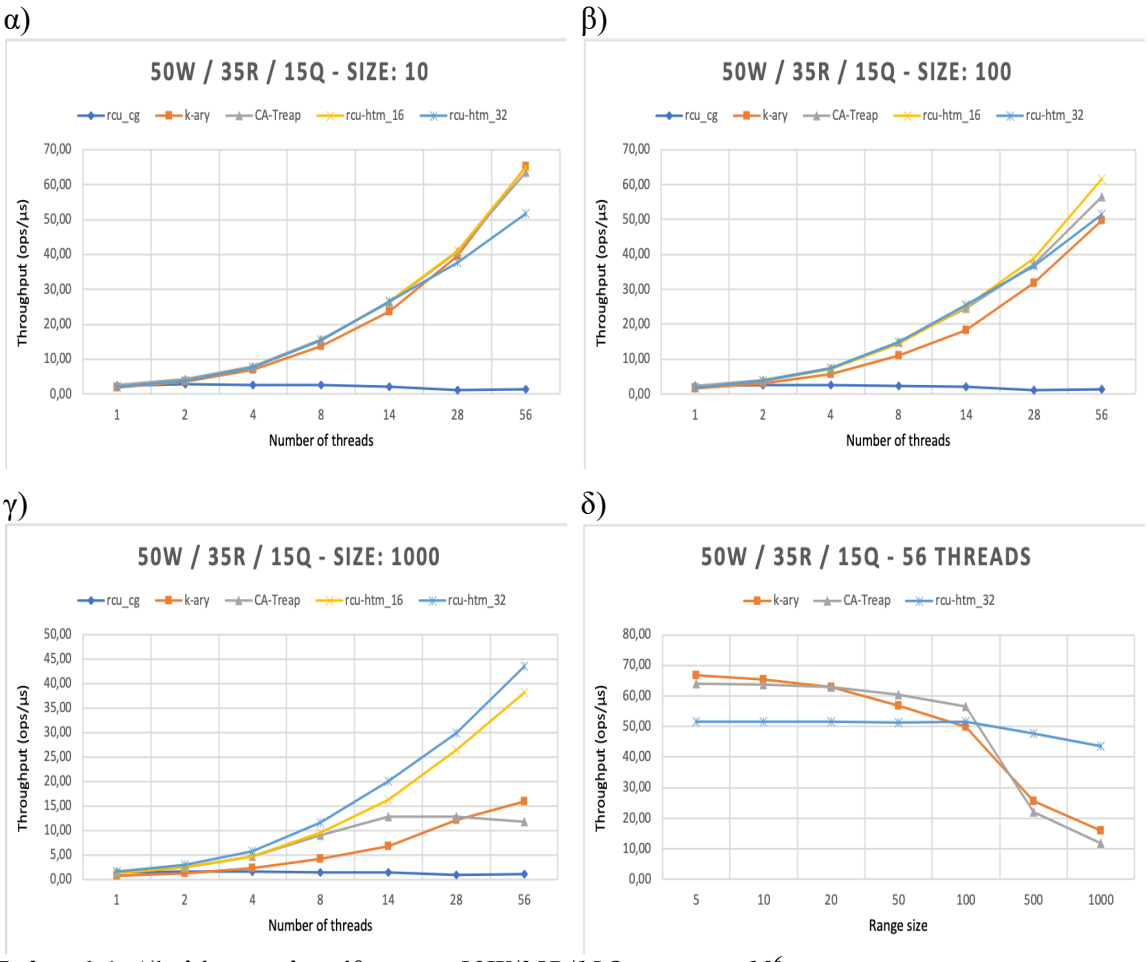
Το διάγραμμα 6.2δ δείχνει πως η τεχνική RCU-HTM έχει καλύτερη απόδοση για 56 threads για τις διάφορες τιμές εύρους που εξετάζονται. Επίσης η απόδοση της μειώνεται σημαντικά λιγότερο με την αύξηση του εύρους σε σχέση με τις δύο άλλες υλοποιήσεις. Επομένως, συνολικά μπορούμε να πούμε ότι η τεχνική RCU-HTM στο συγκεκριμένο σενάριο υπερνικά τις υπόλοιπες υλοποιήσεις τόσο για διαφορετικά εύρη, όσο και για διαφορετικά πλήθη νημάτων που χρησιμοποιούνται.



Σχήμα 6.3. *Αξιολόγηση αλγορίθμου για 24W/50R/26Q και max=10^6.*

Στο σχήμα 6.3 παρατηρούμε σενάρια μετρήσεων της μορφής 24W/50R/26Q. Συγκεκριμένα στο διάγραμμα 6.3α, όπου μελετάται ένα μικρό εύρος για τα range queries φαίνεται όλες οι υλοποιήσεις να έχουν παρόμοια απόδοση και κλιμακωσιμότητα, με την τεχνική RCU-HTM ναι είναι ελαφρώς πιο αποδοτική. Στα διαγράμματα 6.3β, όπου μελετάμε μεσαίου μεγέθους range queries, η RCU-HTM παρουσιάζει σαφώς καλύτερη κλιμακωσιμότητα σε σχέση με τους ανταγωνιστές της, ενώ η απόδοση της για 56 threads αξίζει να σημειωθεί πως είναι σημαντικά υψηλότερη. Αντίστοιχα αποτελέσματα παρατηρούνται και στο διάγραμμα 6.3γ, όπου εξετάζουμε λειτουργίες αναζήτησης εύρους μεγαλύτερου μεγέθους.

Στο διάγραμμα του σχήματος 6.3δ φαίνεται πως η απόδοση της τεχνικής RCU-HTM είναι καλύτερη από αυτή των υπόλοιπων υλοποιήσεων συναρτήσει του εύρους των range queries στα 56 threads και για αυτό το σενάριο μετρήσεων. Καλύτερος παραμένει και ο ρυθμός μείωσης της απόδοσης με την αύξηση του εύρους, γεγονός που δείχνει την ικανότητα του αλγορίθμου να ανταπεξέλθει εξίσου ικανοποιητικά στα διάφορα μεγέθη range queries.



Σχήμα 6.4. *Αξιολόγηση αλγορίθμου για 50W/35R/15Q και max=10^6.*

Το τελευταίο σενάριο που θα μελετήσουμε για το συγκεκριμένο μέγεθος δομής είναι ένα σενάριο που αποτελείται από μεγάλο όγκο λειτουργιών εισαγωγής και διαγραφής στοιχείων στην προς εξέταση δομή, σενάριο που γενικά δεν ευνοεί ούτε τη χρήση RCU-HTM, ούτε το B+-Δέντρο, λόγω των πολλών αντιγράφων, καθώς και των συνενώσεων και σπασιμάτων κόμβων που ενδέχεται να χρειαστούν. Παρ' όλα αυτά, από τα διαγράμματα του σχήματος 6.4 βλέπουμε ότι η RCU-HTM ανταπεξέρχεται ιδιαίτερα αποτελεσματικά ακόμη και σε αυτό το σενάριο. Συγκεκριμένα, για μικρά και μεσαία εύρη range queries, όπως φαίνεται στα διαγράμματα 6.3α-β οι τρεις υλοποιήσεις παρουσιάζουν παρόμοια κλιμακωσιμότητα και κοντινές αποδόσεις για όλους τους εξεταζόμενους αριθμούς νημάτων που χρησιμοποιούνται.

Στο διάγραμμα του σχήματος 6.3γ όμως, όπου και εξετάζουμε το ίδιο σενάριο χρησιμοποιώντας μεγαλύτερο εύρος για τα range queries, η τεχνική RCU-HTM παρουσιάζει πολύ καλύτερη κλιμακωσιμότητα, καθώς και υπερδιπλάσια απόδοση όταν χρησιμοποιούμε 28 και 56 threads. Αυτό είναι απόρροια των ιδιαίτερα αποδοτικών range

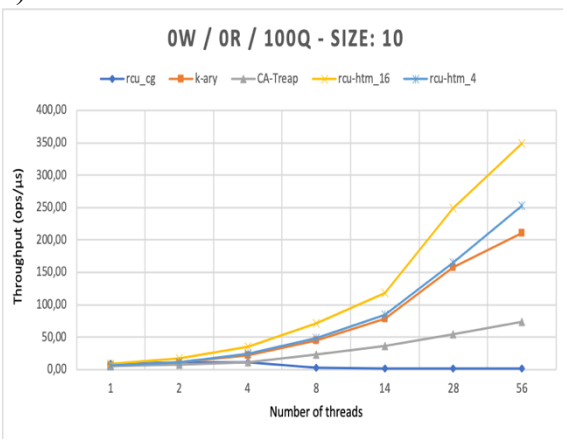
queries που παρέχει η RCU-HTM σε συνδυασμό με το B+-Δέντρο. Η σταθερότητα της τεχνικής συναρτήσει του εύρους των range queries φαίνεται και στο σχήμα 6.4δ, όπου βλέπουμε πως η απόδοσή της είναι σχεδόν ίδια για όλες τις εξεταζόμενες τιμές εύρους σε αντίθεση με τις υπόλοιπες υλοποιήσεις όπου η απόδοση μειώνεται σημαντικά με την αύξηση της τιμής του εύρους.

Συμπερασματικά, μπορούμε πλέον με ασφάλεια να πούμε πως για το συγκεκριμένο μέγεθος δομής η τεχνική RCU-HTM υπερτερεί συνολικά έναντι των υπόλοιπων τεχνικών που εξετάστηκαν. Ιδιαίτερα σημαντικό είναι το γεγονός πως το συμπέρασμα αυτό επιβεβαιώνεται από διαφορετικά σενάρια μετρήσεων, τα οποία έχουν επιλεγεί έτσι ώστε να αντιπροσωπεύουν διαφορετικές ανάγκες, τις οποίες μπορεί να καλύψει αποδοτικά η τεχνική RCU-HTM. Εξίσου σημαντικό είναι επίσης και το ότι η κλιμακωσιμότητα της τεχνικής RCU-HTM δεν επηρεάζεται σημαντικά από το εύρος των λειτουργιών αναζήτησης εύρους, ιδιαίτερα όταν επιλέγουμε m=32 για το B+-Δέντρο.

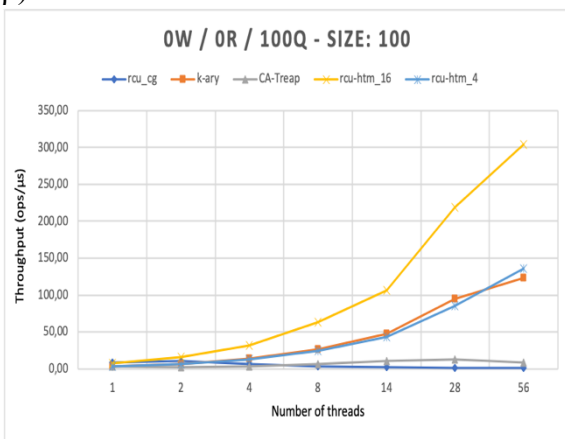
6.3.2 Για max key = 10^4

Στην υποενότητα αυτή θα εξετάσουμε την περίπτωση όπου η κάθε δομή μπορεί να έχει το πολύ 10^4 κλειδιά. Στην περίπτωση αυτή, για το B+-Δέντρο θα χρησιμοποιήσουμε m=4 και m=16.

α)



β)

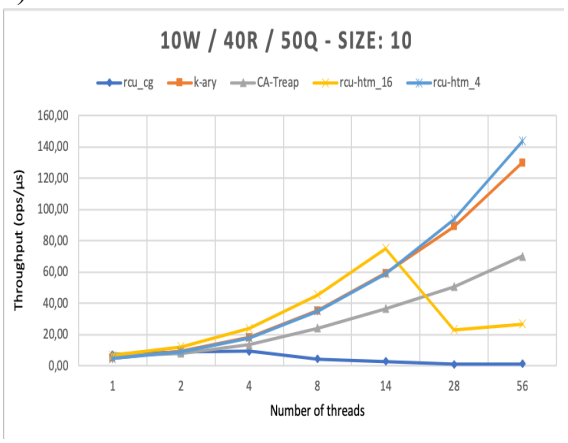


γ)

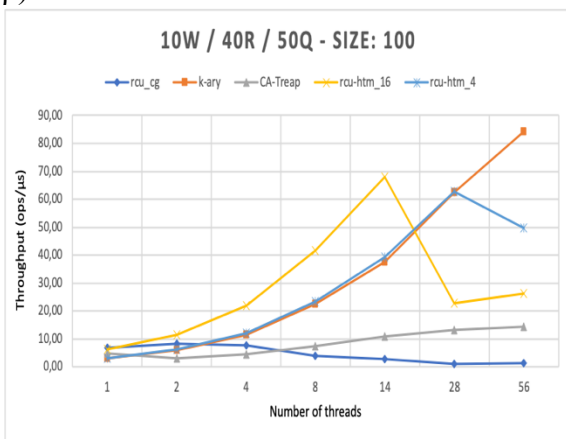
Σχήμα 6.5. *Αξιολόγηση αλγορίθμου για 100% range queries και max=10^4.*

Στα διαγράμματα του σχήματος 6.5 φαίνεται η απόδοση των εξεταζόμενων δομών όταν εκτελούνται μόνο λειτουργίες αναζήτησης εύρους. Αρχικά παρατηρούμε ότι στο σενάριο αυτό το CA-Tree δεν κλιμακώνει καλά, ενώ αντίθετα το k-ST παρουσιάζει σχετικά καλή κλιμακωσιμότητα για τις διαφορετικές τιμές εύρους που εξετάζονται. Ιδιαίτερα σημαντικό είναι το γεγονός ότι η τεχνική RCU-HTM όταν χρησιμοποιείται B+-Δέντρο με m=16 έχει σημαντικά υψηλότερη απόδοση από τις υπόλοιπες τεχνικές, καθώς και σαφώς καλύτερη κλιμακωσιμότητα. Αντίθετα, όταν χρησιμοποιείται m=4, βλέπουμε ότι η τεχνική δεν κλιμακώνει για όταν το εύρος αυξάνεται. Αξίζει βέβαια να λάβουμε υπόψη ότι η τιμή εύρους 1000 είναι ιδιαίτερα μεγάλη αν αναλογιστούμε το μέγεθος της δομής, η οποία περιέχει 5000 κλειδιά.

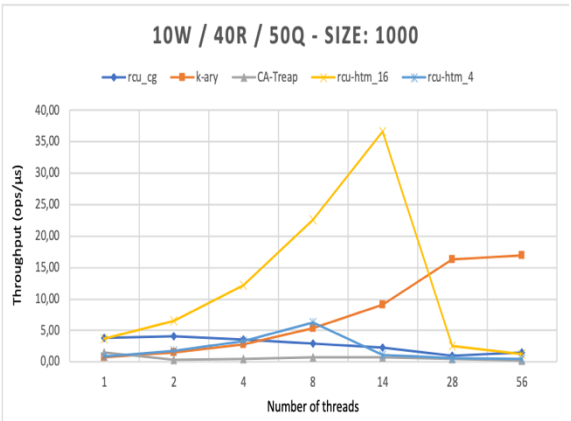
α)



β)



γ)



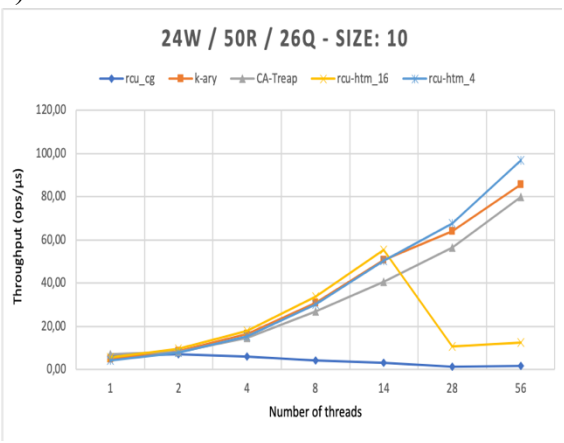
Σχήμα 6.6. *Αξιολόγηση αλγορίθμου για 10W/40R/50Q και max=10^4.*

Στα διαγράμματα του σχήματος 6.6 παρατηρούμε τα αποτελέσματα μετρήσεων που προκύπτουν από σενάρια της μορφής 10W/40R/50Q. Συγκεκριμένα, στο διάγραμμα 6.6α βλέπουμε πως τόσο η RCU-HTM για m=4 όσο και το k-ST παρουσιάζουν πολύ καλή κλιμακωσιμότητα. Αντίθετα το CA-Tree κλιμακώνει χωρίς όμως να πλησιάζει την απόδοση των προαναφερθέντων τεχνικών. Ενδιαφέρον παρουσιάζει και η καμπύλη που αντιπροσωπεύει την απόδοση της τεχνικής RCU-HTM για m=16. Συγκεκριμένα φαίνεται πως κλιμακώνει πολύ καλά μέχρι τα 14 threads, ενώ μετά «γονατίζει», δηλαδή πέφτει σημαντικά η απόδοση του. Αυτό οφείλεται στο γεγονός πως το συγκεκριμένο μέγεθος δομής σε συνδυασμό με την τιμή m=16 έχει ως αποτέλεσμα ένα μικρό B+-Δέντρο, δηλαδή

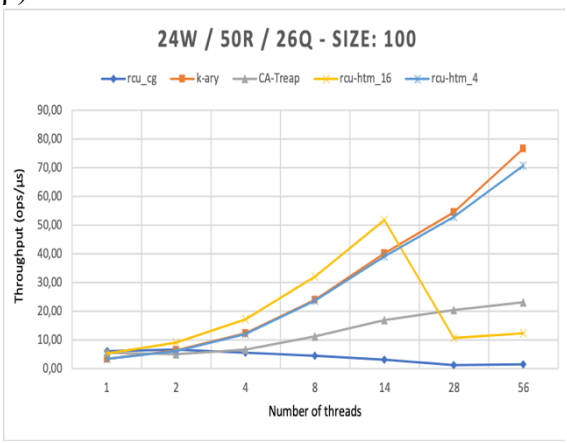
ένα δέντρο με λίγους κόμβους. Έτσι, τα πολλά threads συνωστίζονται με αποτέλεσμα να έχουμε πολλά aborts και να πέφτει πολύ η απόδοση.

Στο διάγραμμα 6.6β βλέπουμε αντίστοιχη εικόνα με το διάγραμμα 6.6α μέχρι τα 28 threads, ενώ όταν ανεβαίνουμε στα 56 threads βλέπουμε πως πέφτει η απόδοση της RCU-HTM για m=4, όπως συνέβη και στο διάγραμμα 6.6α για m=16. Αυτό συμβαίνει με την αύξηση του εύρους, καθώς στην περίπτωση αυτή στο transaction για τα range queries θα εμπλέκονται περισσότερα φύλλα. Το k-ST διατηρεί την πολύ καλή του κλιμακωσιμότητα και για αυτό το εύρος, ενώ αυτή του CA-Tree μειώνεται σημαντικά. Στο διάγραμμα 6.6γ βλέπουμε πως η αύξηση της τιμής του εύρους άφησε ανεπηρέαστο μόνο το k-ST, το οποίο και κλιμακώνει καλά. Η κακή κλιμακωσιμότητα του CA-Tree οφείλεται στα πολλά κλειδώματα που πρέπει να αποκτηθούν, ενώ της τεχνικής RCU-HTM στα πολλά φύλλα που εμπλέκονται στο transaction. Όπως και στα διαγράμματα 6.6α-β το B+-Δέντρο για m=16 έχει πολύ καλή απόδοση μέχρι τα 14 threads, αλλά μετά πέφτει.

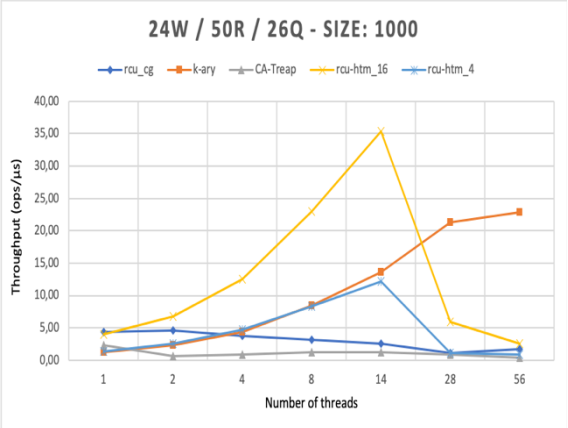
α)



β)



γ)



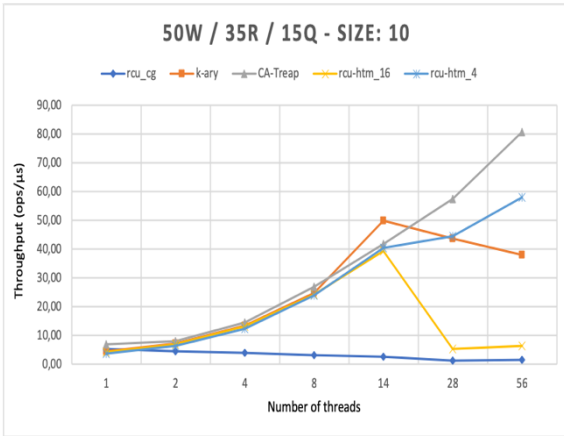
Σχήμα 6.7. *Αξιολόγηση αλγορίθμου για 24W/50R/26Q και max=10^4.*

Στο σχήμα 6.7 παρουσιάζονται αποτελέσματα για το σενάριο 24W/50R/26Q. Συγκεκριμένα, στο διάγραμμα 6.7α βλέπουμε ένα αποτέλεσμα παρόμοιο με το αντίστοιχο διάγραμμα του σχήματος 6.6α με μόνη διαφορά τη βελτίωση της κλιμακωσιμότητας για το CA-Tree, το οποίο ευνοείται από το μεγαλύτερο ποσοστό εισαγωγών/διαγραφών και τα λιγότερα range queries. Με την αύξηση όμως του εύρους στο διάγραμμα 6.7β βλέπουμε πως η απόδοση του CA-Tree μειώνεται σημαντικά. Αυτό συνεχίζεται και στο 6.7γ όπου η απόδοση του CA-Tree είναι χαμηλότερη ακόμα και από αυτή της απλής RCU υλοποίησης,

που χρησιμοποιεί ένα coarse-grain κλείδωμα για ολόκληρη τη δομή για όλες τις λειτουργίες εκτός αυτή της αναζήτησης.

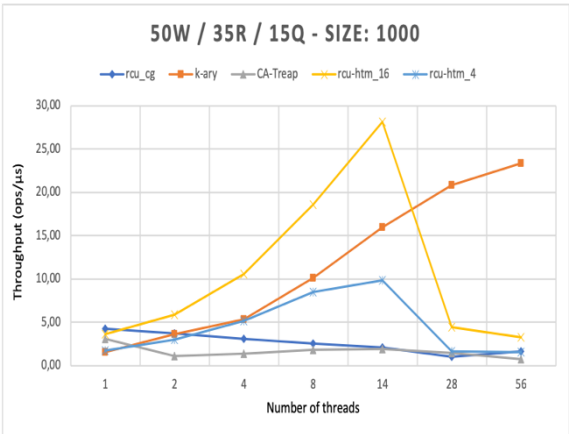
Όσον αφορά το k-ST, αυτό παρουσιάζει σταθερά καλή κλιμακωσιμότητα και για τα τρία εξεταζόμενα εύρη των range queries. Η RCU-HTM για m=4 παρουσιάζει εξίσου καλή κλιμακωσιμότητα με το k-ST για εύρος ίσο με 10 και 100, ενώ για εύρος ίσο με 1000, δηλαδή όταν ψάχνουμε εύρος κλειδιών που αντιστοιχεί περίπου στο 10% των συνολικών κλειδιών του δέντρου, η απόδοσή της πέφτει όταν μεταβαίνουμε από τα 14 στα 28 threads. Τέλος, για m=16, έχουμε πολύ καλή απόδοση μέχρι και τα 14 threads, η οποία όμως μετά μειώνεται πάρα πολύ, όπως έγινε και στα προηγούμενα σενάρια που μετρήσαμε.

α)



β)

γ)



Σχήμα 6.8. *Αξιολόγηση αλγορίθμου για 50W/35R/15Q και max=10^4.*

Στο τελευταίο σενάριο που μελετάμε για μέγεθος δομής ίσο με 10^4 έχουμε 50% των λειτουργιών να είναι λειτουργίες εισαγωγής και διαγραφής κλειδιών, το 35% λειτουργίες αναζήτησης και το υπόλοιπο 15% range queries. Στο διάγραμμα 6.8α φαίνεται πως για το συγκεκριμένο σενάριο και μικρό εύρος range query το CA-Tree παρουσιάζει την καλύτερη κλιμακωσιμότητα με την τεχνική RCU-HTM για m=4 να έχει καλή κλιμακωσιμότητα. Το k-ST έχει καλή απόδοση μέχρι τα 14 threads, όμως αυτή μειώνεται για περισσότερα threads, κάτι που συμβαίνει και για την RCU-HTM με m=16.

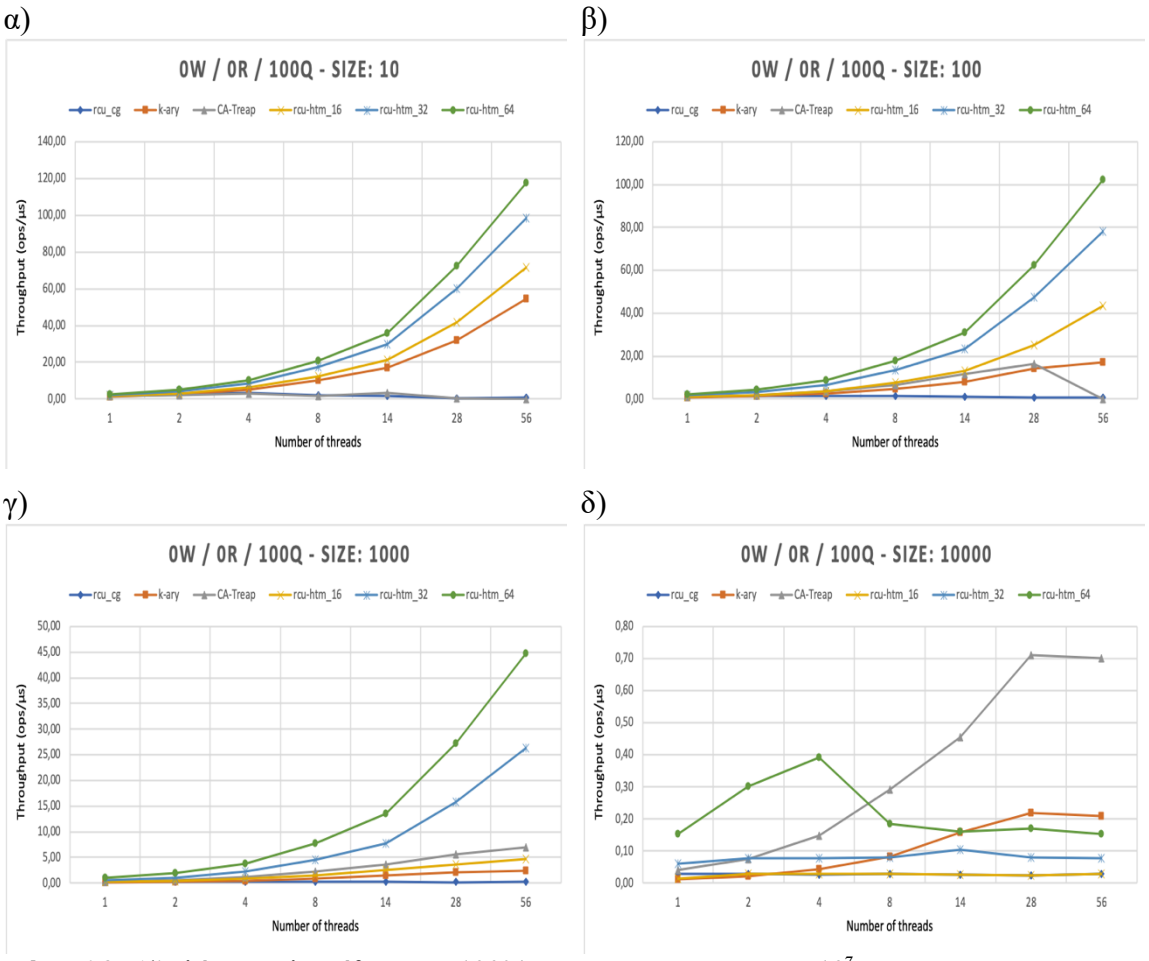
Όταν το εύρος αυξάνεται στα 100 κλειδιά, ευνοείται η τεχνική RCU-HTM με m=4, η οποία έχει την καλύτερη κλιμακωσιμότητα για αυτό το εύρος, όπως φαίνεται στο διάγραμμα 6.8β. Αντίθετα, η απόδοση του CA-Tree πέφτει αισθητά σε σύγκριση με τις

άλλες υλοποιήσεις, αναδεικνύοντας την αδυναμία της τεχνικής για μεγαλύτερα εύρη λειτουργιών αναζήτησης εύρους. Αυτό φαίνεται ακόμα περισσότερο στο διάγραμμα 6.8γ. Αντίθετα το k-ST βελτιώνει την κλιμακωσιμότητά του με την αύξηση του εύρους, ενώ η απόδοσή του μειώνεται λιγότερο σε σχέση με τις υπόλοιπες υλοποιήσεις. Όσον αφορά την RCU-HTM με m=4 φαίνεται πως διατηρεί την καλή της κλιμακωσιμότητα για τη μεσαία τιμή εύρους, έχοντας μάλιστα και πολύ καλή απόδοση για 56 threads. Στο διάγραμμα 6.8γ όμως φαίνεται πως με την περαιτέρω αύξηση του εύρους, η RCU-HTM παρουσιάζει το ίδιο πρόβλημα που έχει παρατηρηθεί και όταν έχουμε m=4.

Συμπερασματικά, βλέπουμε πως συνολικά, για το συγκεκριμένο μέγεθος δομής και λαμβάνοντας υπόψη όλα τα σενάρια μετρήσεων, πιο σταθερή κλιμακωσιμότητα παρουσιάζει το k-ST. Πολύ καλή απόδοση -σε αρκετά σημεία την καλύτερη- παρουσιάζει και η τεχνική RCU-HTM με m=4, το οποίο θεωρούμε και ιδανική τιμή m για το B+-Δέντρο, όταν έχουμε το συγκεκριμένο μέγεθος δομή. Ιδιαίτερα όταν έχουμε μεγάλο όγκο λειτουργιών αναζήτησης εύρους η τεχνική RCU-HTM αντεπεξέρχεται εξαιρετικά. Στον αντίποδα, το CA-Tree έχει καλή απόδοση συγκριτικά με τις υπόλοιπες υλοποιήσεις μόνο όταν έχουμε μεγάλο όγκο εισαγωγών/ διαγραφών κλειδιών και μικρό εύρος για τα range queries.

6.3.3 Για max key = 10^7

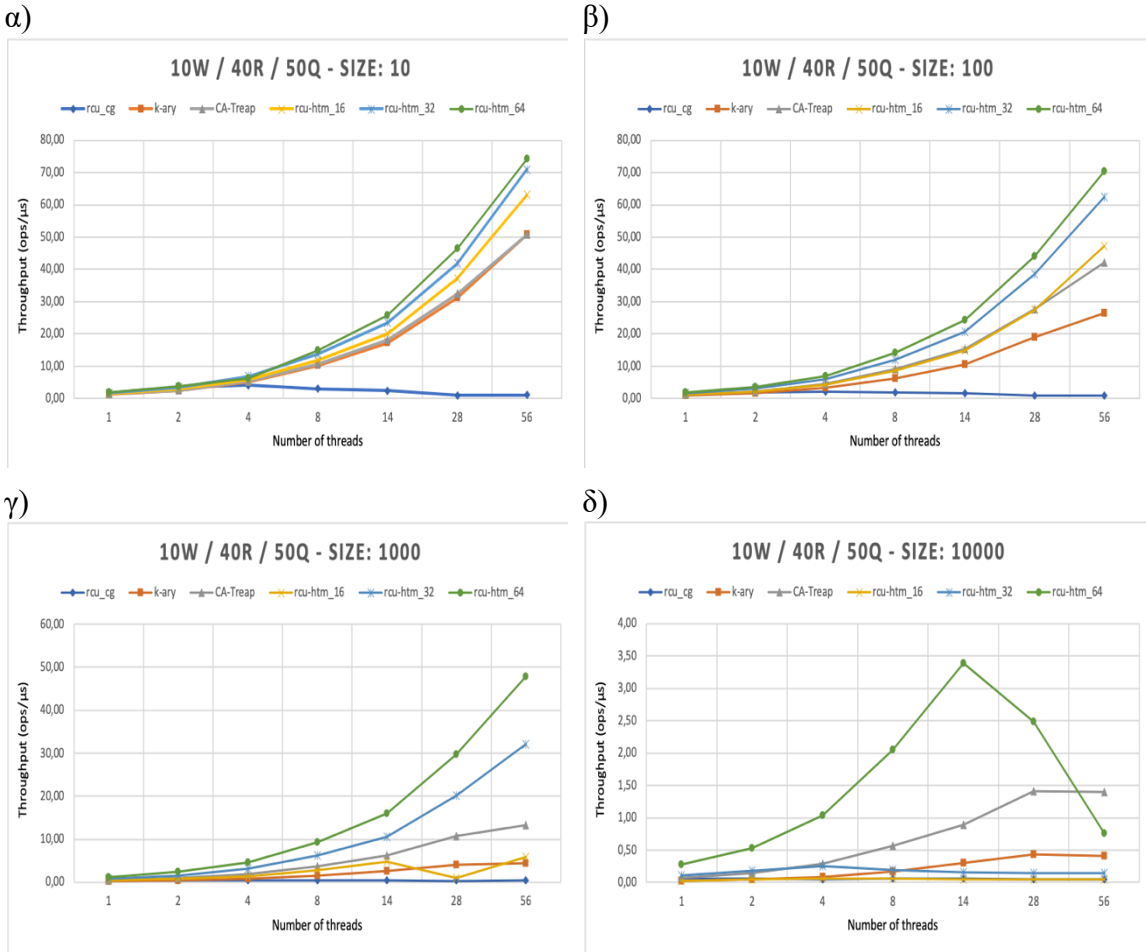
Στην υποενότητα αυτή θα εξετάσουμε την περίπτωση όπου η κάθε δομή μπορεί να έχει το πολύ 10^7 κλειδιά. Στην περίπτωση αυτή, για το B+-Δέντρο θα χρησιμοποιήσουμε m=16, m=32 και m=64.



Σχήμα 6.9. *Αξιολόγηση αλγορίθμου για 100% range queries και max=10^7.*

Στο σχήμα 6.9 βλέπουμε σε διαγράμματα τα αποτελέσματα μετρήσεων για την απόδοση των λειτουργιών αναζήτησης εύρους όταν το μέγεθος της δομής είναι 10^7. Συγκεκριμένα, στο διάγραμμα 6.9α βλέπουμε τόσο οι RCU-HTM υλοποιήσεις όσο και το k-ST παρουσιάζουν καλή κλιμακωσιμότητα με την απόδοση της RCU-HTM να είναι αρκετά καλύτερη. Αντίθετα, το CA-Tree φαίνεται να μην κλιμακώνει καθόλου με την απόδοσή του να παραμένει χαμηλή ανεξάρτητα του αριθμού των threads. Όταν αυξάνουμε το εύρος στην τιμή 100 φαίνεται πως οι υλοποιήσεις RCU-HTM διατηρούν την καλή τους κλιμακωσιμότητα με αυτές που έχουν μεγαλύτερη τιμή τάξης m για το B+-Δέντρο να έχουν λογικά και καλύτερη απόδοση, καθώς σε αυτά τα δέντρα το range query αρκεί να προσπελάσει λιγότερα φύλλα για το ίδιο εύρος αναζήτησης. Για το ίδιο εύρος το k-ST κλιμακώνει καλά με λογικά μειωμένη απόδοση σε σχέση με τη μικρότερη τιμή εύρους, ενώ το CA-Tree κλιμακώνει καλύτερα σε σχέση με το μικρότερο εύρος μέχρι τα 28 threads, με την απόδοση του να πέφτει όμως όταν ο αριθμός των threads αυξάνεται σε 56. Η καλύτερη κλιμακωσιμότητα σε σχέση με το μικρότερο εύρος οφείλεται στο ότι το μεγαλύτερο εύρος επιτρέπει στο δέντρο καλύτερη προσαρμοστικότητα στον ανταγωνισμό και άρα μεγαλύτερο παραλληλισμό.

Στο διάγραμμα 6.9γ βλέπουμε πως η τεχνική RCU-HTM με μεγάλα m συνεχίζει να δίνει αποδοτικές και κλιμακώσιμες παράλληλες λειτουργίες αναζήτησης εύρους με την αύξηση του εύρους αναζήτησης σε 1000. Οι υπόλοιπες υλοποιήσεις, αντίθετα, έχουν πολύ χαμηλή απόδοση. Άξια αναφοράς είναι η κλιμακωσιμότητα του CA-Tree όταν ανεβάζουμε περεταίρω το εύρος αναζήτησης των range queries, όπως φαίνεται στο διάγραμμα 6.9δ, όπου οι υπόλοιπες υλοποιήσεις παρουσιάζουν χαμηλότερη κλιμακωσιμότητα. Από το σχήμα 6.9 μπορούμε γενικά να συμπεράνουμε ότι για τις περισσότερες τιμές εύρους αναζήτησης η τεχνική RCU-HTM δίνει πολύ αποδοτικές λειτουργίες αναζήτησης εύρους και γι' αυτό το μέγεθος δομής, το οποίο θεωρούμε μεγάλο. Αντίθετα, οι ανταγωνιστικές υλοποιήσεις δεν παρέχουν συνολικά καλή απόδοση και κλιμακωσιμότητα.

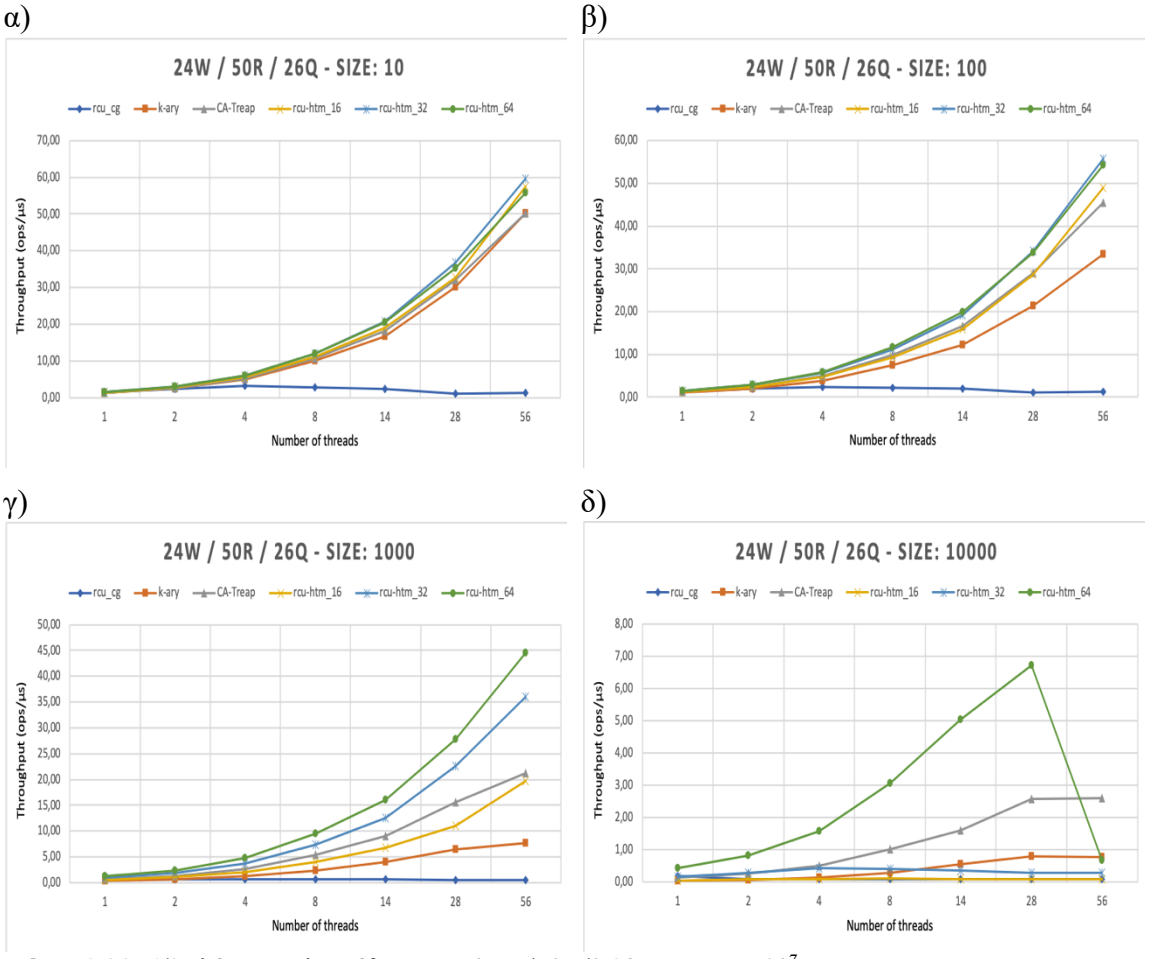


Σχήμα 6.10. *Αξιολόγηση αλγορίθμου για 10W/40R/50Q και max=10^7.*

Στο σχήμα 6.10 παρατηρούμε διαγράμματα της απόδοσης των υλοποιήσεων για σενάρια της μορφής 10W/40R/50Q. Συγκεκριμένα, στο διάγραμμα 6.10α βλέπουμε ότι για μικρό εύρος αναζήτησης η τεχνική RCU-HTM δίνει μεγαλύτερη απόδοση με την αύξηση των threads που χρησιμοποιούμε και άρα καλύτερη κλιμακωσιμότητα, ενώ και οι άλλες δύο υλοποιήσεις έχουν πολύ καλά αποτελέσματα. Αξίζει να τονιστεί ότι μεταξύ των τριών διαφορετικών m που εξετάζονται για το B+-Δέντρο, καλύτερο αποτέλεσμα έχει η τιμή m=64. Στο διάγραμμα 6.10β η εικόνα που βλέπουμε είναι αντίστοιχη αυτής τους διαγράμματος 6.10α με εξαίρεση την πτώση της κλιμακωσιμότητας του k-ST.

Για εύρος αναζήτησης ίσο με 1000, οι υλοποιήσεις RCU-HTM με m=64 και m=32 συνεχίζουν να έχουν την υψηλότερη κλιμακωσιμότητα σύμφωνα με τα αποτελέσματα που

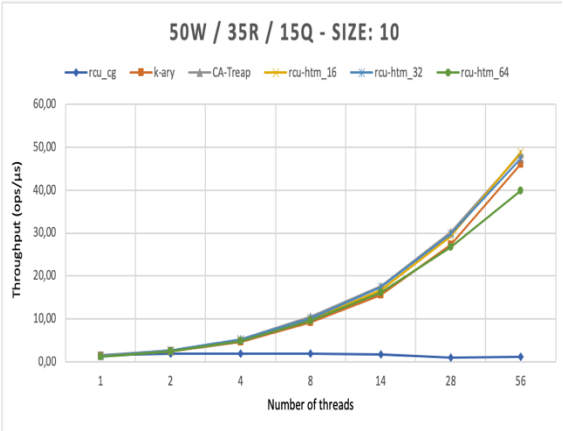
παρουσιάζονται στο διάγραμμα 6.10γ. Αντίθετα, για m=16 φαίνεται να μειώνεται αρκετά η απόδοση της RCU-HTM, η οποία είναι παρόμοια με αυτή του k-ST για τη συγκεκριμένη τιμή εύρους. Αυτό οφείλεται, όπως και σε προηγούμενες περιπτώσεις, στο γεγονός ότι για m=16 έχουμε μικρότερους κόμβους σε σχέση με τις άλλες εξεταζόμενες τιμές της τάξης του B+-Δέντρου και έτσι το range query πρέπει να προσπελάσει σημαντικά μεγαλύτερο αριθμό φύλλων. Όταν αυξάνουμε περεταίρω το εύρος, όπως στο διάγραμμα 6.10δ, το ίδιο συμβαίνει και για το B+-Δέντρο με m=32. Αντίθετα, η τιμή m=64 δίνει πολύ καλή απόδοση μέχρι και τα 14 threads, αλλά μετά δίνει και αυτή χαμηλή απόδοση, όπως και οι υπόλοιπες υλοποιήσεις. Για αυτή την τιμή εύρους σχετικά καλή κλιμακωσιμότητα έχει το CA-Tree, το οποίο υστερεί όμως σε απόδοση.



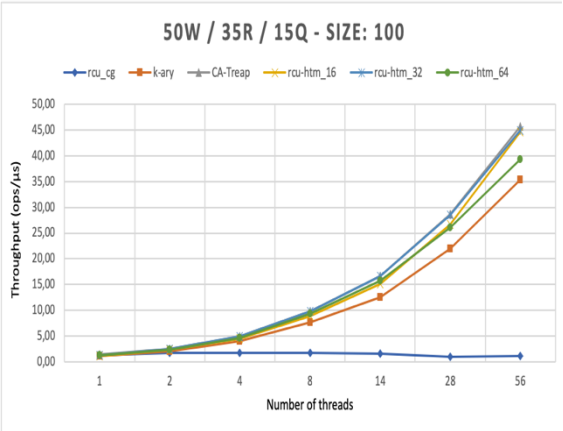
Σχήμα 6.11. *Αξιολόγηση αλγορίθμου για 24W/50R/26Q και max=10^7.*

Η εικόνα που βλέπουμε στα διαγράμματα του σχήματος 6.11, όπου φαίνονται τα αποτελέσματα του σεναρίου μετρήσεων 24W/50R/26Q, μοιάζει αρκετά με αυτή που των διαγραμμάτων του σχήματος 6.10, τα οποία και αναλύθηκαν προηγουμένως. Συγκεκριμένα βλέπουμε πάλι τις RCU-HTM -ιδιαίτερα για m=64 και m=32- να υπερτερούν σε απόδοση και κλιμακωσιμότητα των υπόλοιπων. Εξαίρεση αποτελεί και εδώ το πολύ μεγάλο εύρος αναζήτησης, όπου για m=64 η απόδοση πέφτει όταν μεταβαίνουμε από τα 28 στα 56 threads. Επίσης, η τιμή m=16 φαίνεται πως δίνει καλύτερη κλιμακωσιμότητα σε σχέση με το προηγούμενο σενάριο.

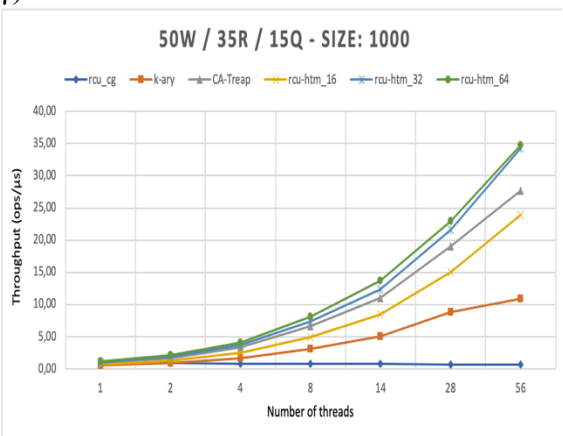
α)



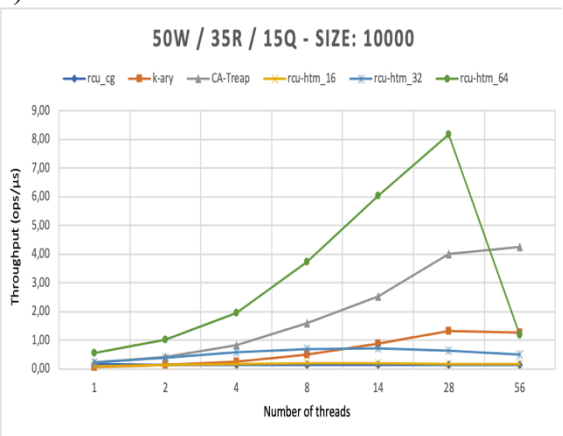
β)



γ)



δ)



Σχήμα 6.12. *Αξιολόγηση αλγορίθμου για 50W/35R/15Q και max=10^7.*

Για το σενάριο 50W/35R/15Q και μικρό ή μεσαίο εύρος λειτουργιών αναζήτησης εύρους οι τρεις διαφορετικές υλοποιήσεις έχουν πολύ κοντινή απόδοση, όπως φαίνεται και στα διαγράμματα 6.12α-β. Για εύρος αναζήτησης ίσο με 1000 οι τιμές m=64 και m=32 δίνουν την καλύτερη απόδοση για 56 threads με το CA-Tree να ακολουθεί, έχοντας χειρότερη κλιμακωσιμότητα, η οποία όμως παραμένει καλή. Αντίθετα, το k-ST φαίνεται να έχει σημαντικά χαμηλότερη κλιμακωσιμότητα, με αποτέλεσμα η απόδοσή του στα 56 threads να είναι έως και τρεις φορές χαμηλότερη σε σχέση με αυτή που έχει η τεχνική RCU-HTM εφαρμοσμένη σε ένα B+-Δέντρο. Τέλος, για πολύ μεγάλο εύρος αναζήτησης, το B+-Δέντρο με m=64 έχει σημαντικά καλύτερη απόδοση μέχρι και τα 28 threads, κάτι που δεν ισχύει για τα 56 threads, όπου το CA-Tree υπερέχει. Στο εύρος αυτό το CA-Tree παρουσιάζει σταθερή κλιμακωσιμότητα, ενώ οι υπόλοιπες υλοποιήσεις έχουν πολύ χαμηλή απόδοση.

Συμπερασματικά, για μέγεθος δομής ίσο με 10^7 η τεχνική RCU-HTM αποτελεί συνολικά την καλύτερη επιλογή. Αυτό επιβεβαιώνεται από την πλειονότητα των μετρήσεων που πραγματοποιήσαμε, τα οποία και ανέδειξαν την τιμή m=64 ιδανικότερη για το B+-Δέντρο, όταν αυτό έχει το συγκεκριμένο πλήθος κλειδιών.

Κεφάλαιο 7

Συμπεράσματα και μελλοντικές επεκτάσεις

7.1 Συμπεράσματα

Σκοπός της παρούσας διπλωματικής εργασίας ήταν η ανάπτυξη μίας τεχνικής συγχρονισμού που θα επιτρέπει αποδοτικές παράλληλες λειτουργίες αναζήτησης εύρους. Στα πλαίσια της παρούσας διπλωματικής εργασίας μελετήσαμε το B+-Δέντρο, στο οποίο και εφαρμόσαμε την τεχνική RCU-HTM. Προηγουμένως είχαμε αναφερθεί τόσο στο απαραίτητο θεωρητικό υπόβαθρο, όσο και σε ήδη υπάρχουσες υλοποιήσεις, με σκοπό την ορθή και πολύπλευρη προσέγγιση της υλοποίησης που αναπτύξαμε.

Στη συνέχεια αξιολογήσαμε πειραματικά της τεχνική που αναπτύξαμε, συγκρίνοντας την απόδοση της με αυτή άλλων υλοποιήσεων, που υποστηρίζουν λειτουργίες αναζήτησης εύρους. Για να μπορούμε να βγάλουμε πλήρη και ορθά συμπεράσματα μελετήσαμε πολλούς διαφορετικούς παράγοντες που μπορούν να επηρεάσουν την απόδοση των υλοποιήσεων, όπως το μέγεθος της προς μελέτη δομής δεδομένων και το εύρος των range queries που εκτελούνται.

Συνολικά, συμπεραίνουμε πως ο αλγόριθμος που αναπτύξαμε δίνει πολύ καλή κλιμακωσιμότητα για διάφορα σενάρια μετρήσεων. Όπως φάνηκε και από τα διαγράμματα του προηγούμενου κεφαλαίου, η τεχνική RCU-HTM έχει εξάλλου την υψηλότερη απόδοση για 56 threads στα περισσότερα από αυτά. Έχουμε, λοιπόν, μία τεχνική συγχρονισμού που σε συνδυασμό με ένα B+ Δέντρο δίνει μία πολύ αποδοτική παράλληλη δομή δεδομένων που εκτός από πολύ αποδοτικές λειτουργίες αναζήτησης εύρους, που ήταν και ο κύριος σκοπός της παρούσας εργασίας, προσφέρει πολύ καλή απόδοση σε περιπτώσεις, όπου έχουμε σημαντικό ποσοστό των συνολικών λειτουργιών να είναι εισαγωγές και διαγραφές κλειδιών.

Συγκεκριμένα η RCU-HTM έχει εξαιρετικά αποτελέσματα για μεσαίο και μεγάλο μέγεθος δομής, ενώ ανταπεξήλθε ικανοποιητικά και όταν μελετήθηκε «μικρό» δέντρο. Αξιοσημείωτο είναι το γεγονός ότι η απόδοση RCU-HTM δεν επηρεάστηκε από την αύξηση του εύρους στο βαθμό που επηρεάστηκαν οι υπόλοιπες υλοποιήσεις, δίνοντας σταθερά καλή απόδοση. Αυτό οφείλεται τόσο στην τεχνική RCU-HTM που δεν απαιτεί την απόκτηση κλειδωμάτων για την εκτέλεση range queries, όσο και στο B+ Δέντρο που προσφέρει πολύ γρήγορες λειτουργίες αναζήτησης εύρους. Αντίθετα, η αδυναμία του B+ Δέντρου στις λειτουργίες αντιγραφής και διαγραφής δεν αποτέλεσαν τροχοπέδη στην απόδοση του αλγορίθμου σε σενάρια με μεγάλο ποσοστό τέτοιων λειτουργιών. Το γεγονός αυτό αναδεικνύει τη δυναμική της τεχνικής RCU-HTM.

Τέλος, είδαμε ότι σημαντικό ρόλο για την απόδοση της RCU-HTM παίζει η τάξη του B+ Δέντρου που χρησιμοποιούμε, δηλαδή το μέγεθος του κόμβου του δέντρου. Παρατηρήσαμε ότι για διαφορετικό μέγεθος δομής, προτείνεται διαφορετικό μέγεθος κόμβου. Το ίδιο ισχύει σε μικρότερο βαθμό και για τα διαφορετικά σενάριο μετρήσεων που μελετήσαμε, καθώς διαφορετικό μέγεθος κόμβου μπορεί να είναι πιο αποτελεσματικό για διαφορετικές λειτουργίες του δέντρου. Θα μπορούσαμε να πούμε ότι η RCU-HTM εφαρμοσμένη σε B+ Δέντρο έχει καλύτερη απόδοση όταν έχουμε ένα συγκεκριμένο αριθμό κόμβων, το μέγεθος των οποίων μπορεί να είναι διαφορετικό ανάλογα τις απαιτήσεις του σεναρίου εκτέλεσης.

7.2 Μελλοντικές επεκτάσεις

Με βάση τα παραπάνω συμπεράσματα και όσα αναφέρθηκαν μέχρι στιγμής στην εργασία αυτή, προκύπτουν μερικές ενδιαφέρουσες πιθανές μελλοντικές επεκτάσεις. Αρχικά θα ήταν χρήσιμο να μελετήσουμε περισσότερους παράγοντες που μπορούν επηρεάσουν την απόδοση της τεχνικής που αναπτύξαμε. Για παράδειγμα θα μπορούσαμε να μελετήσουμε το πως επηρεάζεται η απόδοση από την αρχιτεκτονική του συστήματος που χρησιμοποιούμε για την πειραματική αξιολόγηση, χρησιμοποιώντας ένα σύστημα UMA αρχιτεκτονικής. Επίσης, χρήσιμη θα ήταν και η σύγκριση της τεχνικής RCU-HTM με περισσότερες υλοποιήσεις, οι οποίες υποστηρίζουν παράλληλες λειτουργίες αναζήτησης εύρους. Ακόμα, θα μπορούσαμε να εφαρμόσουμε την τεχνική RCU-HTM σε περισσότερες δομές δεδομένων για την εκτέλεση range queries και να συγκρίνουμε την απόδοσή τους με αυτή του B+ Δέντρου. Επιπλέον, για την υλοποίηση που αναπτύξαμε, χρησιμοποιήθηκε η υλοποίηση HTM που παρέχει η Intel. Θα ήταν σκόπιμο επομένως να χρησιμοποιηθεί και κάποια άλλη υλοποίηση, ώστε να μελετήσουμε την επιρροή του HTM συστήματος στον αλγόριθμό μας.

Τέλος, λόγω της μεγάλης σημασίας που φάνηκε να έχει το μέγεθος του κόμβου του B+ Δέντρου στην απόδοση της τεχνικής RCU-HTM στα διαφορετικά σενάρια μετρήσεων, προτείνεται η δημιουργίας μίας νέας δομής δεδομένων, που θα βασίζεται στο B+ Δέντρο, αλλά θα έχει μεταβλητό μέγεθος κόμβου. Θεωρούμε ότι η εφαρμογή RCU-HTM σε μία τέτοια δομή θα δώσει μία πολύ αποδοτική παράλληλη δομή δεδομένων, η οποία θα έχει εξίσου καλή απόδοση ανεξάρτητα παραγόντων όπως το μέγεθος της δομής και η αναλογία των λειτουργιών που εκτελούνται, καθώς θα προσπαθεί να προσαρμόζεται σε αυτά, αλλάζοντας το μέγεθος του κόμβου.

Βιβλιογραφία

[1]. Moore, Gordon E., *Cramming more components onto integrated circuits*, 1965

[2]. Amdahl, G., *The validity of the single processor approach to achieving large scale computing capabilities,* In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, April 1967

[3]. Flynn, Michael J., *Some Computer Organizations and Their Effectiveness,* 1972

[4]. Bayer, R., McCreight, E. M., *Organization and maintenance of large ordered indexes,* 1972

[5].https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

[6]. Sagonas, K., Winblad, K., *Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees,* 2015

[7]. Sagonas, K., Winblad, K., *Contention Adapting Trees,* 2015

[8]. Brown, T., Anvi, H., *Range Queries in Non-blocking k-ary Search Trees,* 2012

[9]. Brown, T., Helga, J., *Non-blocking k-ary Search Trees,* 2011

[10]. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F., *Non-blocking Binary Search Trees*, In Proc. 29th ACM Symposium on Principles of Distributed Computing, pages 131–140, 2010

[11]. Siakavaras, D., Nikas, K., Goumas, G., Koziris, N., *RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees,* 2017

[12]. Siakavaras, D., Nikas, K., Goumas, G., Koziris, N., *Combining RCU with HTM to Implement Highly Efficient Balanced Binary Search Trees,* 2017

[13]. McKenney, P. E., Slingwine, J. D., *Read-Copy Update: Using Execution History to Solve Concurrency Problems,* in Parallel and Distributed Computing and Systems, Las Vegas, NV, Oct. 1998, pp 509-519

[14]. Chatterjee, B., *Lock-free Linearizable 1-Dimensional Range Queries*, 2017

[15]. Winblad, K., *Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data*, 2017

[16]. Seidel, R., Aragon, C. R., *Randomized Search Trees*, 1996

[17]. Petrank, E., Timmat, S., *Lock-free data-structure iterators*, 2013

[18]. Attiya, H., Guerraoui, R., Ruppert, E., *Partial snapshot objects*, 2008

[19]. Basin, D., Bortnikov, E., Braginsky, A., Golan-Gueta, G., Hillel, E., Keidar, I., Sulamy, M., *Kiwi: A Key-Value Map for Scalable Real-Time Analytics*, 2017