ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων (CSLab)

SafeTree - Σύστημα Αυτόματης Παραλληλοποίησης Δέντρικών Δομών Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σωτήρη Π. Δραγώνα

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

Αθήνα, Μάρτιος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων
(CSLab)

SafeTree - Σύστημα Αυτόματης Παραλληλοποίησης Δέντρικών Δομών Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σωτήρη Π. Δραγώνα

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3 Απριλίου 2020.

..... Γεώργιος	..... Νεκτάριος	..... Διονύσιος
Γκούμας Επίκουρος Καθηγητής Σχ. Ηλ. Μηχανικών και Μηχ. Υπολογιστών	Κοζύρης Καθηγητής Σχ. Ηλ. Μηχανικών και Μηχ. Υπολογιστών	Πνευματικάτος Καθηγητής Σχ. Ηλ. Μηχανικών και Μηχ. Υπολογιστών

Αθήνα, Μάρτιος 2020.

.....

Σωτήριος Π. Δραγώνας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Περίληψη

Στην παρούσα διπλωματική εργασία, παρουσιάζουμε μία βιβλιοθήκη αυτόματης παραλληλοποίησης βασισμένη στην τεχνική RCU-HTM η οποία επιτρέπει την δημιουργία thread-safe δεντρικών δομών παράγοντας κώδικα πολύ παρόμοιο με των αντίστοιχων σειριακών εκδόσεων. Η βιβλιοθήκη είναι γραμμένη σε C++ και απλοποιεί το σχεδιασμό παράλληλων δεντρικών δομών κρύβοντας την πολυπλοκότητα και τους κινδύνους του παράλληλου προγραμματισμού, προσφέροντας ταυτόχρονα πανομοιότυπες επιδόσεις και κλιμάκωση με την χειροκίνητη εφαρμογή της τεχνικής RCU-HTM.

Λέξεις Κλειδιά--- rcu-htm, παράλληλος προγραμματισμός, παράλληλες δομές δεδομένων, hardware transactional memory, read-copy-update, αυτόματη παραλληλοποίηση, c++ βιβλιοθήκη

Abstract

In this thesis, we present an automatic parallelization framework based on the RCU-HTM technique which allows the creation of thread-safe tree data structures while producing code very similar to their serial counterparts. The framework is designed as a C++ library and simplifies the design of parallel tree structures by hiding the complexity and hazards of parallel programming, while offering performance and scaling identical to the manual application of the RCU-HTM technique.

Keywords— rcu-htm, parallel programming, concurrent data structures, hardware transactional memory, read-copy-update, automatic parallelization, c++ library

Ευχαριστίες

Θέλω να ευχαριστήσω τον Δημήτρη Σιακαβάρα για την ευκαιρία να αναλάβω μία τόσο ενδιαφέρουσα διπλωματική εργασία και για την πολύτιμη βοήθεια του για την εκπόνηση της. Επιπλέον, θέλω να ευχαριστήσω και τον καθηγητή Γ. Γκούμα για την υποστήριξη του αλλά και για την εισαγωγή στον κόσμο του παράλληλου προγραμματισμού.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Ορισμός Παράλληλων Δομών Δεδομένων

Ως παράλληλες δομές δεδομένων ορίζουμε τις δομές δεδομένων που επιτρέπουν την ταυτόχρονη πρόσβαση από πολλαπλά νήματα ή διεργασίες χωρίς να καταστρέφεται η δομή τους και παρέχοντας έγκυρα αποτελέσμα

1.2 Αναγκαίοτητα και Χρησιμότητα

Με αφορμή την σταδιακή παύση της ισχύος του νόμου του Moore, οι μοντέρνοι σχεδιαστές μικροεπεξεργαστών αλλά και γενικότερα υπολογιστικών συστημάτων έχουν στραφεί προς συστήματα με πολλαπλούς επεξεργαστικούς πυρήνες ή ακόμα και στην ενσωμάτωση ετερογενών συστημάτων για την επιτάχυνση εξειδικευμένων εργασιών, με σκοπό να συνεχιστεί η αύξηση της απόδοσης των συστημάτων. Εμείς θα επικεντρωθούμε στα συστήματα πολλαπλών πυρήνων, τα οποία αφορά και η παρούσα εργασία.

Ωστόσο, για τους προγραμματιστές που υλοποιούν λογισμικό για αυτα τα συστημάτα, οι παράλληλες αρχιτεκτονικές εισάγουν αρκετές προκλήσεις όσον αφορά την ορθότητα λειτουργίας, την κατανόηση των τεχνικών προγραμματισμού αλλά και την πλήρη αξιοποίηση των διαθέσιμων πόρων,

Εστιάζοντας στις δομές δεδομένων, ένας προγραμματιστής καταλαβαίνει πως οι παραδοσιακές δομές δεδομένων ,πχ. ένα δέντρο αναζήτησης, δεν λειτουργούν αυτούσιες σε ένα περιβάλλον παράλληλου προγραμματισμού εξαιτίας θεμάτων όπως ο συγχρονισμός μεταξύ των νημάτων τα οποία θα συζητηθούν στην επόμενη ενότητα.

Αδιαμφισβήτητα οι δομές δεδομένων αποτελούν το θεμέλιο των περισσότερων προγραμμάτων. Οι βάσεις δεδομένων απαιτούν κάποιες δομές για την αποθήκευση δεδομένων στο δίσκο και άλλες δομές για τα ευρετήρια τους, τα

λειτουργικά συστήματα απαιτούν ουρές προτεραιότητας για την χρονοδρομολόγηση διεργασιών, οι περισσότεροι αλγόριθμοι που χρησιμοποιούνται για την επίλυση σημαντικών προβλημάτων απαιτούν κάποιο είδος δομής δεδομένων. Για να λειτουργούν τα προγράμματα σε παράλληλες αρχιτεκτονικές λοιπόν, χρειάζονται παράλληλες δομές δεδομένων.

1.3 Ιδιαιτερότητες παράλληλου προγραμματισμού

1.3.1 Συγχρονισμός

Ο συγχρονισμός αναφέρεται στην απαίτηση ενός προγράμματος να εκτελούνται συγκεκριμένα τμήματα του είτε με συγκεκριμένη σειρά, είτε με συγκεκριμένη σχετική σειρά μεταξύ τους. Παραδείγματος χάρη, σε ένα πρόγραμμα που υπολογίζει την μέση τιμή των τιμών ενός πίνακα και την εκτυπώνει, είναι επιθυμητό η τιμή να εκτυπώνεται αφότου έχει υπολογιστεί. Προγραμματίζοντας με παράλληλο κώδικα ωστόσο κάτι τέτοιο δεν είναι αυτονόητο:

```
double average(double array[], int length, double* result) {
    *result = sum(array)/length;
}

int main() {
    double avg;
    double nums[100]

    /* initialize nums with numbers from 1 to 100 */
    thread avg_thread(average, nums, 100, avg);

    std::cout << avg << std::endl;
}
```

Στο παραπάνω απόκομμα κώδικα, αν ήταν σειριακό, θα περίμενε ο προγραμματιστής ότι θα εκτυπώσει τον μέσο όρο, ωστόσο αυτό που θα έβλεπε με μεγαλύτερη πιθανότητα θα ήταν κάποια τυχαία τιμή που θα περιείχε η μεταβλητή avg μιας και το αρχικό νήμα εκτέλεσης θα δημιουργούσε ένα νέο νήμα για τον υπολογισμό του μέσου όρου αλλά δεν θα περίμενε την ολοκλήρωση του υπολογισμού για να εκτυπώσει.

```
double average(double array[], int length, double* result) {
    *result = sum(array)/length;
}

int main() {
    double avg;
    double nums[100]

    * initialize nums with numbers from 1 to 100 */
    thread avg_thread(average, nums, 100, avg);

    avg_thread.join();

    std::cout << avg << std::endl;
}
```

Με την εισαγωγή της νέας εντολής, το αρχικό νήμα περιμένει την ολοκλήρωση του υπολογισμού.

Το παραπλανω απλοϊκό σενάριο αποτελεί μία από τις ευκολότερες περιπτώσεις αναγκαιότητας συγχρονισμού. Σε πιο πολύπλοκες καταστάσεις όπως τα προβλήματα παραγωγών-καταναλωτών, το πρόβλημα των συνδαιτημόνων φιλοσόφων (Dining Philosophers) σαν θεωρητικά προβήματα και φυσικά σε συστήματα που βρίσκονται στην παραγωγή και πιθανότατα επιτελούν κρίσιμες λειτουργίες όπως οικονομικές συναλλαγές, η υλοποίηση σωστού συγχρονισμού αποτελεί πραγματική πρόκληση.

1.3.2 Απόδοση

Όπως αναφέρθηκε, βασικό κίνητρο της αξιοποίησης παράλληλων συστημάτων είναι η αύξηση της απόδοσης. Ιδανικά η εισαγωγή υπολογιστικών μονάδων πχ. πυρήνων επεξεργαστή θα επέφερε γραμμική αύξηση της απόδοσης ίση με τον αριθμό των μονάδων επί την απόδοση της μονάδας. Αντί για αυτό, χωρίς σωστό προγραμματισμό, είναι πιθανή ακόμα και η μείωση της απόδοσης σε σχέση με την μονάδα. Μερικοί παράγοντες που επηρεάζουν την απόδοση είναι:

1. **Ο νόμος του Amdahl**

   Ο νόμος του Amdahl εκφράζει το εξής στα παράλληλα συστήματα:
   Αν κάποιο έργο μπορεί να διαμεριστεί σε διαδικασίες που μπορούν να γίνουν παράλληλα και διαδικασίες που αναγκαστικά πρέπει να γίνουν σειριακά, υποθέτωντας ότι το ποσοστό των παράλληλων διαδικασιών απαιτεί ποσοστό p του συνολικού χρόνου σειριακής εκτέλεσης, τότε η μέγσιστη δυνατή επιτάχυνση από την εισαγωγή n παράλληλων μονάδων θα είναι:

   $$\frac{1}{(1-p)+p/n}$$

2. **Το σύστημα μνήμης**
   Μία λάθος υπόθεση που μπορεί να γίνει στον παράλληλο προγραμματισμό είναι πως δεν έχει σημασία το σύστημα μνήμης και πως όλες οι προσβάσεις της μνήμης συμβαίνουν ακριβώς όπως σε ένα κλασσικό σύστημα με σειριακή αρχιτεκτονική. Αντίθετα το σύστημα μνήμης μπορεί να παίζει σημαντικό ρόλο στην απόδοση ενός προγράμματος. Παραδείγματα αυτού είναι:

   - **Η κρυφή μνήμη**
     Για να προσπελάσει ένα νήμα κάποια θέση μνήμης που έχει τροποποιηθεί από κάποιο νήμα που εκτελούταν σε άλλον πυρήνα, για να διατηρηθεί η συνοχή της μνήμης (cache coherence), απαιτείται η μεταφορά δεδομένων μεταξύ των πυρήνων. Επιπλέον, η επικοινωνία μεταξύ των πυρήνων γίνεται μέσα από κάποιο σύστημα διασύνδεσης, σαν μία απλούστευση μπορεί να θεωρηθεί ως ένας δίαυλος επικοινωνίας, ο οποίος δίαυλος έχει συγκεκριμένη χωρητικότητα και δυνατότητα εξυπηρέτησης ταυτόχρονων αιτημάτων. Αυτό μεταφράζεται σε πραγματικές διαφορές στην απόδοση, αν το πρόγραμμα κάνει μη αποδοτικές προσβάσεις στην διαμοιραζόμενη μνήμη.

- **Η μη ομοιόμορφη πρόσβαση στην μνήμη (NUMA)**
  Σε κάποια συστήματα, πχ. σε συστήματα με πολλαπλά sockets, η επικοινωνία των επεξεργαστικών μονάδων με τη μνήμη μπορεί να μην είναι ομοιόμορφη. Ένα παράδειγμα είναι η πρόσβαση ενός πυρήνα ενός socket σε δεδομένα που βρίσκονται στην κρυφή μνήμη L3 ενός άλλου socket, όπου αναγκαστικά θα πρέπει να μεταφερθούν μέσω κάποιας διασύνδεσης με πολύ μεγαλύτερη καθυστέρηση από μία άμεση πρόσβαση στην μνήμη L3 του ίδιου socket.

3. **Ο ανταγωνισμός (contention) για τους διάφορους πόρους**
   Ο ανταγωνισμός για τους πόρους παίζει μεγάλο ρόλο στην τελική απόδοση ενός παράλληλου συστήματος. Οι πόροι του συστήματος, είτε αυτοί αποτελούν κάποια συσκευή εισόδου/εξόδου είτε κάποιες δομές δεδομένων στην μνήμη μπορούν συνήθως να υποστηρίξουν μέχρι κάποιο συγκεκριμένο αριθμό ταυτόχρονων προσβάσεων. Η αύξηση του ανταγωνισμού για αυτούς τους πόρους οδηγεί σε μειωμένη απόδοση. Ειδικά όσον αφορά τις παράλληλες δομές δεδομένων και την συγκεκριμένη εργασία, η μέγιστη απόδοση επιτυγχάνεται με την μείωση του ανταγωνισμού στον ελάχιστο δυνατό.

4. **Η δομή των δεδομένων** Η δομή των δεδομένων αποτελεί σημαντικό παράγοντα στην απόδοση ενός παράλληλου προγράμματος. Ιδανικά τα δεδομένα πρέπει να είναι δομημένα ώστε να υπάρχει ο μικρότερος δυνατός διαμοιρασμός μνήμης ανάμεσα στους επεξεργαστές ώστε να ελαχιστοποιείται ο ανταγωνισμός αλλά και οι συγκρούσεις στην κρυφή μνήμη. Αυτό επιτυγχάνεται μέσω διαφόρων τεχνικών, είτε μέσω έξυπνου προγραμματισμού, πχ. ξεχωριστά δεδομένα για κάθε επεξεργαστικό νήμα και ένωση των αποτελεσμάτων της επεξεργασίας, είτε μέσω αλλαγής της ίδιας της δομής των δεδομένων πχ. alignment των δομών για αποφυγή διαμοιρασμού της ίδιας γραμμής κρυφής μνήμης από διαφορετικούς πυρήνες (false sharing).

5. **Οι μηχανισμοί συγχρονισμού** Οι μηχανισμοί που χρησιμοποιούνται για την επίτευξη του συγχρονισμού επηρεάζουν την απόδοση εξαιτίας του ότι μπορεί να εισάγουν προσθετή καθυστέρηση ώστε να επιτευχθεί ο συγχρονισμός. Όπως θα δούμε και στην επόμενη ενότητα, η χρήση μηχανισμών που αξιοποιούν polling για την ενημέρωση τους χαραμίζει κύκλους επεξεργασίας στον έλεγχο αν έχουν αλλάξει κάποιες συνθήκες, ενώ σε περίπτωση event-based μηχανισμών που μπορεί να θέτουν σε αναμονή τα νήματα, υπάρχει η καθυστέρηση των context switches ώστε να αφυπνιστούν οι διεργασίες.

1.3.3 Debugging

Ίσως η μεγαλύτερη πρόκληση του παράλληλου προγραμματισμού. Εξαιτίας της πολυπλοκότητας των αλληλεπιδράσεων ανάμεσα σε πολλαπλά νήματα, πολύ συχνά αποδεικνύεται δύσκολο να αναπαραχθεί κάποια δυσλειτουργία ενός προγράμματος σε παραπάνω από μία εκτέλεση. Επιπλέον, ο αριθμός των πιθανών ροών εκτέλεσης που αυξάνεται πολλαπλασιαστικά με την αύξηση του αριθμού των νημάτων καθιστά δύσκολη την κατανόηση ενός παράλληλου προγράμματος. Το πιο χαρακτηριστικό παράδειγμα ειναι το πιο συχνό εργαλείο αποσφαλμάτωσης, δηλαδή τα print statements ανάμεσα στις εντολές, το οποίο καθίσταται δύσχρηστο μιας και χωρίς συγχρονισμό, ο χρήστης παρατηρεί στην οθόνη του ένα συνονθύλευμα από σειρές χαρακτήρων ανακατεμμένες μεταξύ τους.

1.4 Τύποι Παράλληλων Δομών Δεδομένων

1.4.1 Δομές Δεδομένων βασισμένες σε κλείδωμα

Οι δομές δεδομένων που χρησιμοποιούνται πιο ευρέως είναι οι δομές δεδομένων που βασίζονται σε κλειδώμα. Το κλείδωμα αναφέρεται στην χρήση primitives όπως mutexes ή spinlocks για τη δημιουργία περιοχών αμοιβαίου αποκλεισμού, δηλ. κώδικα όπου για κάθε δεδομένη χρονική στιγμή, μονάχα ένα από τα τρέχοντα νήματα εκτελεί.

Θα αναλύσουμε συνοπτικά αυτά τα δύο primitives, προτού παρουσιάσουμε παραδείγματα αυτών των δομών.

Mutex Το Mutex είναι ένα εργαλείο συγχρονισμού που συνήθως υλοποιείται με τη βοήθεια του λειτουργικού συστήματος. Προσφέρει δύο δυνατές λειτουργίες, το κλείδωμα(lock) και το ξεκλείδωμα(unlock).

Ένα νήμα που θα συναντήσει μία εντολή κλειδώματος, θα ελέγξει ατομικά μία θέση μνήμης ή κάποια δομή του λειτουργικού που υποδεικνύει εάν υπάρχει άλλο νήμα που εκτελείται στην περιοχή κώδικα που ακολουθεί. *Αν ναι*, αναστέλλεται η εκτέλεση του νήματος ή της διεργασίας (πχ. με την τοποθέτηση σε μία ουρά του λειτ. συστήματος), μέχρι να κληθεί μία εντολή ξεκλειδώματος. *Αν όχι*, τότε προχωράει καταγράφοντας όμως πρώτα πως εισήλθε στην περιοχή. Έτσι επιτυγχάνεται ο αμοιβάιος αποκλεισμός.

Το Mutex χρησιμοποιείται ευρέως για την υλοποίηση κλειδώματος και αποφεύγεται μόνο σε περιπτώσεις όπου δεν επιτρέπεται η αναστολή λειτουργίας (πχ. εντός ενός interrupt handler στον κώδικα του λειτ. συστήματος) ή σε περιπτώσεις όπου οι περιοχές αμοιβαίου αποκλεισμού έχουν χρονικό κόστος συγκρίσιμο με αυτό της εκτέλεσης του κώδικα του κλειδώματος. Σε αυτές τις περιπτώσεις χρησιμοποιούνται spinlocks.

Spinlock Το Spinlock είναι ένα πολύ παρόμοιο εργαλείο συγχρονισμού, μόνο που χρησιμοποιείται polling για τον έλεγχο της εισόδου στην περιοχή αμοιβαίου αποκλεισμού, πράγμα το οποίο σημαίνει ότι δεν αναστέλλεται ποτέ η λειτουργία κάποιου νήματος. Υλοποιείται συνήθως με τη χρήση ατομικών εντολών.

Πέρα από αυτά τα δύο είδη κλειδωμάτων, υπάρχουν πολλές παραλλαγές που αντιμετωπίζουν προκλήσεις του παράλληλου προγραμματισμού, όπως κλειδώματα για τη διαχείριση του συνωστισμού, κλειδώματα reader-writer που επιτρέπουν την παράλληλη εκτέλεση αναγνώσεων αλλά όχι αναγνώσεων και εγγραφών ή πολλών εγγραφών και πολλά άλλα. Αντίστοιχα υπάρχουν παράλληλες δομές δεδομένων βασισμένες σε αυτά. Ωστόσο, για τους σκοπούς αυτής της διπλωματικής θα χωρίσουμε όλες τις δομές που βασίζονται σε κλειδώματα σε δύο κατηγορίες υψηλότερου επιπέδου

1.4.1.1 Coarse-grained locking

Οι παράλληλες δομές που βασίζονται σε coarse-grained locking είναι η απλούστερη εφαρμογή των primitives που προαναφέραμε. Σε αυτές, ξεκινώντας από μία υλοποίηση σειριακής δομής δεδομένων, τοποθετείται ολόκληρος ο κώδικας των πράξεων τους -εισαγωγή, αναζήτηση κτλ- σε περιοχές αμοιβαίου αποκλεισμού. Συνηθως υπάρχει ένα primitive συγχρονισμού ανά στιγμιότυπο της δομής το οποίο επιτρέπει μόνο σε ένα νήμα να έχει πρόσβαση στη δομή για κάθε χρονική στιγμή.

Μπορούν να χρησιμοποιηθούν και άλλες υλοποιήσεις, πχ. ένα reader-writer lock για να υπάρχει δυνατότητα παράλληλων αναγνώσεων, ωστόσο ο βασικός παράγοντας της τεχνικής του Coarse-grained locking είναι ότι το κλείδωμα δεν αξιοποιεί την εσωτερική δομή της δομής δεδομένων για να επιτρέψει υψηλότερο βαθμό παραλληλίας.

Αυτό είναι τελικά και το μειονέκτημα της τεχνικής αφού επιτρέπει αλλαγή της δομής από μονάχα ένα ταυτόχρονο νήμα και άρα σειριοποιεί την πρόσβαση σε αυτή, με αποτέλεσμα να μην υπάρχει βελτίωση στην απόδοση των δομών με τη χρήση παραπάνω νημάτων.

Στα πλεονεκτήματα της αντίθετα ανήκουν το γεγονός ότι είναι πολύ έυκολη η υλοποίηση της και προσφέρει thread-safety, κάτι που μπορεί να είναι αποδεκτό για κάποια δομή της οποίας δεν έχει σημασία η ταχύτητα πρόσβασης αλλά μονάχα η διατήρηση της συνοχής της.

Παράδειγμα υλοποίησης Έστω μία συνδεδεμένη λίστα `LinkedList` με μεθόδους `void insert_at_end(int new_element)`, `void delete_at(int index)`, `int index_of(int element)`. Η υλοποίηση της παράλληλης δομής θα είχε την εξής μορφή σε C++:

```
// ThreadSafeLinkedList inherits from LinkedList
class ThreadSafeLinkedList: LinkedList {
    private:
        Mutex mut;
    public:
        // initialize a linked list and a mutex as unlocked
        ThreadSafeLinkedList(): LinkedList(), mut(UNLOCKED){}

        void insert_at_end(int new_element) {
            // lock
            mut.lock();

            // call method of linked list
            LinkedList::insert_at_end(new_element);

            // unlock
            mut.unlock();
        }

        void delete_at(int index) {
            mut.lock();
            LinkedList::delete_at(index);
            mut.unlock();
        }

        int index_of(int element) {
            mut.lock();
            int index = LinkedList::index_of(element);
            mut.unlock();

            return index;
        }


}
```

Listing 1: Coarse-Grained Locking Linked List

1.4.1.2 Fine-grained locking

Με τον όρο fine-grained locking αναφερόμαστε στο σύνολο των υλοποιήσεων παράλληλων δομών δεδομένων που **αξιοποιούν την εσωτερική δομή τους για την επίτευξη παράλληλης πρόσβασης, βασιζόμενες σε κλειδώματα**. Συχνά χρησιμοποιούνται περισσότερα primitives συγχρονισμού ανάλογα με την εσωτερική οργάνωση των δεδομένων. Σε δεντρικές δομές σαν παράδειγμα, μπορεί κάθε κόμβος να έχει το δικό του κλείδωμα και να πρέπει να **συντονιστούν τα κλειδώματα ώστε οι προσβάσεις να γίνονται με thread-safe τρόπο**. Αυτή η προσέγγιση επιτρέπει την υλοποίηση πολλών τεχνικών συγχρονισμού αφού μπορούν να συνδυαστούν τα primitives και να **αξιοποιηθούν ιδιότητες των δομών ώστε να επιτευγχθεί η μέγιστη δυνατή παραλληλία** [CGR13],[Bro+10],[WSJ18].

Η προσέγγιση αυτή ωστόσο παρουσιάζει αρκετές δυσκολίες. Ο σχεδιασμός των δομών coarse-grained συγχρονισμού είναι τόσο απλός που ακόμα και ένας αρχάριος προγραμματιστής θα μπορούσε να τον υλοποιήσει, έχοντας στα χέρια του την σειριακή δομή. Αντίθετα, ο σχεδιασμός των fine-grained δομών τείνει περισσότερο προς τον τομέα των κουρασμένων PHD φοιτητών και ερευνητών.

Η αλληλεπίδραση μεταξύ κώδικα εκτός των περιοχών αμοιβαίου αποκλεισμού με τον κώδικα εντός σε συνδυασμό με την **αλληλεπίδραση μεταξύ των περιοχών αμοιβαίου αποκλεισμού**, δημιουργεί μία αρκετά χαοτική κατάσταση. Γίνεται **δύσκολη η κατανόηση της σειράς εκτέλεσης του κώδικα, δημιουργούνται deadlocks, γίνονται εύκολα λάθη που καταστρέφουν τη δομή** και σαν κερασάκι στο παράλληλο κέϊκ, **το debugging γίνεται δύσκολο αφού συχνά η ίδια εκτέλεση κώδικα έχει εντελώς διαφορετικά αποτελέσματα**.

Πέραν αυτού, απαιτείται και δεξιοτεχνία για την επίτευξη κώδικα που θα επιτρέπει περισσότερη κλιμάκωση από το coarse-grained locking, αφού **πρέπει να χρησιμοποιηθεί ο ελάχιστος αριθμός κλειδωμάτων**, αφού κάθε επιπλέον κλείδωμα σπρώχνει την απόδοση προς αυτή του προαναφερθέντος. Τέλος, υπάρχει και **κόστος απόδοσης για την επίτευξη των κλειδωμάτων**, όπου χαραμίζονται κύκλοι επεξεργαστή εντός της υλοποίησης των primitives αλλά και σε καταστάσεις όπου τα primitives δεν επιτρέπουν με τον βέλτιστο τρόπο την πρόσβαση στα νήματα, πχ. επειδή δεν διαχειρίζονται αποδοτικά το συνωστισμό.

Στα πλεονεκτήματα τους φυσικά ανήκει η δυνατότητα επίτευξης υψηλής παραλληλίας και αντίστοιχης απόδοσης, η ύπαρξη βιβλιοθηκών για κλειδώματα στις περισσότερες υπάρχουσες γλώσσες προγραμματισμού και η δοκιμασμένη χρήση τους, μιας και είναι ο πιο διαδεδομένος τρόπος υλοποίησης παράλληλων δομών δεδομένων.

1.4.2 Lock Free Δομές Δεδομένων

Οι Lock Free δομές δεδομένων έχουν - όπως δηλώνει και το όνομα τους- σκοπό την αποφυγή χρήσης κλειδώματος στην υλοποίηση τους και κατ'επέκταση, την αποφυγή του blocking,δηλαδή της αναμονής που δημιουργείται από τα κλειδώματα.

Υλοποιούνται συνήθως με τη χρήση ατομικών εντολών και μπορούν να παρέχουν κάποιες πολύ χρήσιμες εγγυήσεις σε σχέση με τις δομές που είναι βασισμένες σε κλειδώματα. Συνήθως αυτές οι εγγυήσεις χωρίζονται σε τρεις κλάσεις. Συγκεκριμένα[Nam16]:

- **Obstruction Freedom**, ο αλγόριθμος εγγυάται ότι ένα νήμα θα παρουσιάζει πρόοδο στην πράξη την οποία εκτελεί, εφόσον δεν υπάρχει κάποια σύγκρουση, δηλ. δε θα χρειάζεται ποτέ να περιμένει δράσεις άλλων νημάτων
- **Lock Freedom**, θα υπάρχει πρόοδος στην εκτέλεση όλων των πράξεων της δομής, δηλ. τουλάχιστον μία πράξη εγγυημένα θα ολοκληρώνεται σε πεπερασμένο αριθμό βημάτων για κάθε δεδομένη χρονική στιγμή. Συνδυασμός του Obstruction-Freedom με την μη ύπαρξη livelocks
- **Wait Freedom**, η δομή παρέχει εγγυήσεις για την πρόοδο της κάθε πράξης. Κάθε πράξη ολοκληρώνεται σε πεπερασμένο αριθμό βημάτων. Συνδυάζει lock freedom και μη ύπαρξη λιμοκτονίας (starvation freedom)

Φαινομενικά, αποτελούν την ιδανική υλοποίηση μίας παράλληλης δομής αλλά πρακτικά παρουσιάζουν αρκετές δυσκολίες.

- **Δυσκολία υλοποίησης**, οι αλγόριθμοι των lock-free δομών παρουσιάζουν αυξημένη δυσκολία κατανόησης αλλά και επιβεβαίωσης ορθής λειτουργίας. Η μη ύπαρξη περιοχών αποκλεισμού σημαίνει ότι δεν υπάρχει άμεσος τρόπος καθορισμού της σειράς εκτέλεσης των εντολών. Το thread-safety επιτυγχάνεται με την διενέργεια κατάλληλων ατομικών ελέγχων κατά την εκτέλεση ώστε να εξασφαλίζεται η συνοχή της δομής. Η μεγαλύτερη δυσκολία,ωστόσο, είναι το γεγονός ότι οι ατομικές εντολές δεν επιτρέπουν την σύνθεση τους σε ατομικές πράξεις.
- **Απόδοση**, συχνά οι αλγόριθμοι συγχρονισμού των lock-free δομών προσπαθούν πολλαπλές φορές να ολοκληρώσουν μία πράξη ώσπου να μην υπάρχει σύγκρουση(retries) ή είναι αρκετά πολύπλοκοι με αποτέλεσμα να οδηγούν σε μικρότερη απόδοση από lock-based υλοποιήσεις.
- **Σημαντικές διαφορές στη λογική από δομή σε δομή**, οι lock-free υλοποιήσεις συχνά έχουν πολύ διαφορετική μορφή ανάλογα με τη δομή που υλοποιείται. Αυτό έχει ως αποτέλεσμα να μην είναι μεταφέρσιμη η λογική υλοποίησης από τη μία δομή στην άλλη. Έτσι καταλήγουν να γίνονται αποκλειστικά τομέας κουρασμένων PHD φοιτητών.

Παρ'όλα αυτά έχει γίνει εκτενής έρευνα και στον τομέα (βλ. [CNT14], [RM15]) και ειδικά η εργασία πάνω σε ένα γενικό framework δημιουργίας lock-free δέντρων([BER17]) ενέπνευσε την οργάνωση δεδομένων της βιβλιοθήκης της παρούσας εργασίας.

1.5 Transactional Memory

Με τον ορο *Transactional Memory* αναφερόμαστε στην χρήση συναλλαγών για την επίτευξη συγχρονισμού αντί για κλειδώματα ή ατομικές εντολές. Οι συναλλαγές είναι σύνολα προσβάσεων στην μνήμη (reads και writes) που γίνονται με ατομικό τρόπο.

Υπάρχουν δύο προσεγγίσεις για την υλοποίηση αυτών των συναλλαγών:

- **Hardware Transactional Memory (HTM)**, όπου αξιοποιούνται μηχανισμοί του επεξεργαστή για την υλοποίηση των συναλλαγών
- **Software Transactional Memory**, όπου η λογική υλοποιείται σε επίπεδο λογισμικού, χρησιμοποιώντας άλλες μορφές συγχρονισμού ως δομικά εργαλεία

Μιας και η εργασία βασίζεται στην HTM, θα αναλύσουμε μόνο αυτή τη μέθοδο.

Πως γίνεται ο συγχρονισμός; Αρχικά υπάρχει κάποιος τρόπος να ορισθεί πως ένα τμήμα κώδικα θα εκτελεστεί εντός συναλλαγής. Στην περίπτωση της υλοποίησης σε Intel επεξεργαστές, αυτός είναι ειδικές εντολές αρχής και τέλους συναλλαγής,ως μέλη του ISA, που ορίζουν πως οι προσβάσεις ανάμεσα τους ανήκουν σε συναλλαγή.

Κάθε πρόσβαση σε μία περιοχή μνήμης (μεγέθους μίας γραμμής cache[Sia+15] στην υλοποίηση της intel, μιας και ο μηχανισμος χτίζεται ως επέκταση του cache coherence μηχανισμού), *την πρώτη φορά*, ανάλογα με το αν είναι ανάγνωση η εγγραφή τοποθετείται σε ειδικά σύνολα, *το σύνολο αναγνώσεων και το σύνολο εγγραφών*. Εάν κατά τη διάρκεια της εκτέλεσης κώδικα από ένα νήμα, γίνει μία εγγραφή σε θέση μνήμης που βρίσκεται εντός ενός από τα δύο σύνολα, αποτυγχάνει ολόκληρη η συναλλαγή και η κατάσταση των συμμετέχουσων θέσεων μνήμης επαναφέρεται στην κατάσταση που ήταν πριν ξεκινήσει η συναλλαγή και μπορεί να γίνει εκ νέου προσπάθεια εκτέλεσης της. Έτσι οι εντολές που εκτελούνται εντός συναλλαγής, εκτελούνται ατομικά.

Το πλεονέκτημα αυτής της προσέγγισης είναι ότι οι κρίσιμες περιοχές που με την υλοποίηση των κλειδωμάτων θα ήταν σε περιοχή αμοιβαίου αποκλεισμού, μπορούν να εκτελούνται παράλληλα αν δεν υπάρχει σύγκρουση των

προσβάσεων. Με αυτό τον τρόπο, σε ένα ιδανικό σύστημα, μπορεί να τοποθετηθεί όλος ο κώδικας μίας δομής εντός συναλλαγής και να έχουμε απόδοση ανάλογη του fine-grained locking με δυσκολία υλοποίησης ισοδύναμη με το coarse-grained locking.

Δυστυχώς για την απόδοση και ευτυχώς για εμάς τους μελετητές, οι πραγματικές υλοποιήσεις έχουν περιορισμούς που εμποδίζουν αυτή την ιδανική συμπεριφορά και απαιτούν προσπάθεια για να οδηγηθούμε σε αποδοτική εκτέλεση.

Ας δούμε μερικούς από αυτούς τους περιορισμούς:

1. **Μέγεθος συνόλων**, τα σύνολα που αναφέρθηκαν έχουν ένα περιορισμένο μέγεθος στην υλοποίηση του επεξεργαστή. Μία συναλλαγή που απαιτεί περισσότερο χώρο, αναγκαστικά αποτυγχάνει, άρα μόνο περιορισμένος αριθμός προσβάσεων χωράει εντός μίας συναλλαγής. Οι αποτυχίες για αυτο το λόγο ονομάζονται capacity conflicts.

2. **Πρόοδος**, οι υλοποιήσεις που κυκλοφορούν[Sia+15] έχουν την προσέγγιση best-effort στην ολοκλήρωση των συναλλαγών και δεν εγγυώνται πως μία συναλλαγή θα επιτύχει όσες φορές και εάν τρέξει. Έτσι απαιτείται κάποιος άλλος μηχανισμός ως τελευταία λύση που να εγγυάται την ολοκλήρωση, όπως ένα κλείδωμα μετά από συγκεκριμένο αριθμό αποτυχιών

3. **Λόγοι αποτυχίας συναλλαγών**, στις πραγματικές υλοποιήσεις οι συναλλαγές αποτυγχάνουν και για πολλούς άσχετους με τις προσβάσεις λόγους όπως system interrupts, page faults, exceptions, context switches αλλά και από επιλογή του χρήστη. Επομένως ο κώδικας που συγχρονίζεται παίζει σημαντικό ρόλο στο αν μπορεί να εκτελεστεί εντός συναλλαγής, πόσο μάλλον αποδοτικά.

Παρά τους περιορισμούς, με έξυπνη χρήση του μηχανισμού μπορούν να δημιουργηθούν παράλληλες δομές με **υψηλή απόδοση, γραμμικά αυξανόμενη με τον αριθμό νημάτων που τις προσπελαύνουν**, εφόσον οι προσβάσεις αφορούν ανεξάρτητες περιοχές μνήμης. Επιπλέον, ο παραγόμενος κώδικας μπορεί όπως θα δείξουμε να είναι πολύ παρόμοιος με τη σειριακή έκδοση, που αποτελεί μεγάλο προτέρημα για αυτή τη μέθοδο.

Κεφάλαιο 2

Η Βιβλιοθήκη Αυτόματης Παραλληλοποίησης

2.1 Εισαγωγή και Στόχοι

Με την παρούσα διπλωματική παρουσιάζουμε μία αυτόματη πλατφόρμα(framework) παραλληλοποίησης που διευκολύνει σε τεράστιο βαθμό την υλοποίηση παράλληλων δομών δεδομένων που βασίζονται σε δενδρικές δομές. Η πρακτική της υλοποίηση είναι μία βιβλιοθήκη C++, η οποία μπορεί να βρεθεί εδώ: https://github.com/BazookaMusic/RcuHTMParallelTrees.

Οι σχεδιαστικοί στόχοι της είναι:

1. Η διαχείριση των κινδύνων της παραλληλίας από την βιβλιοθήκη και όχι από το χρήστη
2. Τελικός κώδικας όσο το δυνατόν πιο κοντά στη σειριακή μορφή του σε επίπεδο κατανόησης αλλά και εμφάνισης
3. Κανένας συμβιβασμός στην απόδοση συγκριτικά με χειροκίνητες υλοποίησεις της τεχνικής RCU-HTM (cost-free abstraction)
4. Ευκολία στην αποσφαλμάτωση για τον τελικό χρήστη

2.2 Τεχνική RCU-HTM

Η παρούσα διπλωματική εργασία βασίζεται στην τεχνική RCU-HTM, μία τεχνική που αναπτύχθηκε από τον εισηγητή μου Δημήτρη Σιακαβάρα σε συνεργασία με το εργ. Υπολ. Συστημάτων (CSLab). Η περιγραφή που ακολουθεί βασίζεται στο επίσημο paper της εργασίας[Sia+17].

2.2.1 Γενική Περιγραφή

Η τεχνική RCU-HTM είναι μία τεχνική που συνδυάζει την τεχνική Read-Copy-Update (RCU) [Des+12] και Hardware Transactional Memory (HTM) για την **υλοποίηση αποδοτικών παράλληλων δέντρων αναζήτησης**.

Η παρούσα διπλωματική **επεκτείνει τη χρήση της σε οποιαδήποτε δεντρική δομή**, δεδομένων κάποιων περιορισμών. Παρόμοια με τους αλγορίθμους που βασίζονται στην τεχνική RCU, **οι αλλαγές στο δέντρο συμβαίνουν σε αντίγραφα στην ιδιωτική μνήμη των νημάτων** (βλ. δέντρα των αντιγράφων) αντί να γίνονται στο ίδιο το δέντρο, εξ'ου και εφαρμογή της τεχνικής RCU. Αυτό επιτρέπει σε νήματα που διασχίζουν το δέντρο να το κάνουν χωρίς συγχρονισμό, χωρίς να επηρεάζονται από αλλαγές άλλων νημάτων. Ταυτόχρονα, **με τη χρήση της HTM, τα αντίγραφα αυτά μπορούν να συνδεθούν ατομικά από πολλαπλούς εγγραφείς**, επιτρέποντας την ύπαρξη παραπάνω από ενός νημάτων-εγγραφέων.

Αυτό είναι και το βασικό πλεονέκτημα της τεχνικής ενάντια στους προϋπάρχοντες αλγορίθμους βασισμένους στην τεχνική RCU. **Αφού γίνουν οι κατάλληλες αλλαγές στα ιδιωτικά αντίγραφα, εκτελείται μία ατομική συναλλαγή μέσα στην οποία επιβεβαιώνεται πως όλα τα μέλη του δέντρου που επηρεάστηκαν από την αλλαγή παραμένουν αναλλοίωτα από την ανάγνωση τους και εφόσον επιτύχει η επιβεβαίωση, εγκαθίστανται τα αντίγραφα**.

2.2.2 Ο αλγόριθμος

- **Διάσχιση**, Η πρώτη φάση του αλγορίθμου είναι η διάσχιση. Το δέντρο διασχίζεται εκτός συναλλαγής ώστε να εντοπιστεί ένας κόμβος στον οποίο θα συνδεθεί στη συνέχεια το ιδιωτικό αντίγραφο. Αυτός ο κόμβος ονομάζεται σημείο σύνδεσης. Για πράξεις αναζήτησης, εδώ ολοκληρώνεται ο αλγόριθμος, για πράξεις αλλαγής, αποθηκεύεται η διαδρομή προς το σημείο σύνδεσης.

- **Δημιουργία Αντιγράφου**, Κάθε νήμα δημιουργεί ένα τοπικό αντίγραφο του υποδέντρου που θα αλλάξει στο δέντρο και υλοποιεί τις αλλαγές πάνω του. Ταυτόχρονα διατηρεί αναφορές στους αρχικούς κόμβους που οδήγησαν στο αντίγραφο και αντίγραφα των διευθύνσεων των παιδιών τους για την επιβεβαίωση.

- **Επιβεβαίωση**,Μετά τη δημιουργία του αντιγράφου, ξεκινάει μία συναλλαγή HTM. Η επιβεβαίωση χωρίζεται σε δύο βήματα. Το πρώτο είναι η αναζήτηση του σημείου σύνδεσης ώστε να επιβεβαιωθεί πως υπάρχει ακόμη στο αρχικό δέντρο. Χρησιμοποιείται το μονοπάτι που αποθηκεύτηκε στη διάσχιση ή αναζητείται με βάση τη δομή του δέντρου (πχ. με βάση το κλειδί του). Το δεύτερο βήμα είναι η επιβεβαίωση ότι δεν άλλαξαν τα παιδιά των αλλαγμένων κόμβων, όπου χρησιμοποιούνται οι αρχικοί κόμβοι και τα αντίγραφα των διευθύνσεων των παιδιών.

- **Σύνδεση**,Εφόσον επιτύχει η επιβεβαίωση, συνδέεται το αντίγραφο ατομικά και ολοκληρώνεται η συναλλαγή. Αλλιώς επανεκκινείται ολόκληρη η διαδικασία, ώσπου επιτύχει.

2.3 Απαιτήσεις από το χρήστη

2.3.1 Τεχνικές απαιτήσεις

Απαιτείται η χρήση ενός μεταγλωττιστή που υποστηρίζει τη μεταγλώττιση κώδικα C++ τουλάχιστον έκδοσης 11. Επιπλέον απαιτείται η χρήση επεξεργαστή που υποστηρίζει το TSX-NI instruction set της Intel. Μελλοντικές εκδόσεις πιθανότατα θα υποστηρίξουν και άλλες υλοποιήσεις Hardware Transactional Memory.

2.3.2 Ορισμός Δεντρικών Δομών

Ως δεντρικές δομές ορίζουμε τις δομές οι οποίες μπορούν να οριστούν αναδρομικά ως ένας κόμβος ρίζα και n παιδιά τα οποία είναι επίσης δέντρα,όπου n μέχρι κάποιο ορισμένο μέγιστο πλήθος παιδιών. Για τον ορισμό των κενών παιδιών χρησιμοποιούνται οι `nullptr` δείκτες.

Επιπλέον η υλοποίηση των δομών πρέπει να περιέχει κάποιο αντικείμενο που έχει το ρόλο του κόμβου και υποστηρίζει κάποιες συγκεκριμένες μεθόδους, στο οποίο θα αναφερόμαστε ως Node από εδώ και πέρα.

Δεν υποστηρίζονται:

1. Η ύπαρξη κύκλων
2. Οι γονικοί δείκτες (parent pointers) ή δείκτες προς ανώτερα επίπεδα του δέντρου
3. Δέντρα με κόμβους με απεριορίστο αριθμό παιδιών

Ένας καλός πρόχειρος κανόνας για το αν μία δομή υλοποιείται μέσω της βιβλιοθήκης είναι εάν όλες της οι πράξεις μπορούν να εκφραστούν ως: προσθήκη η αφαίρεση κάποιου υποδέντρου ως παιδί κάποιου κόμβου, αλλάζοντας ένα δείκτη.

Αυτός ο ορισμός επιτρέπει τη δημιουργία πληθώρας δομών όπως όλων των ειδών τα δυαδικά δέντρα αναζήτησης, b-trees, στοίβες, ουρές, μονά συνδεδεμένες λίστες και οτιδήποτε ικανοποιεί τον προηγούμενο ορισμό.

2.3.3 Μορφή Δέντρου

Το δέντρο πρέπει να αποτελείται από ένα αντικείμενο δέντρου που θα περιέχει την ρίζα. Η ρίζα όπως και όλοι οι κόμβοι θα πρέπει να είναι κάποιο αντικείμενο που υποστηρίζει συγκεκριμένες μεθόδους και το οποίο θα ονομάσουμε αντικείμενο **Node**.

2.3.4 Αρίθμηση Παιδιών Κόμβου

Ο μοναδικός περιορισμός στη μορφή του Node, δηλ. του κόμβου των δέντρων, είναι ότι πρέπει να υπάρχει μία νοητή αρίθμηση των δυνατών παιδιών του που ξεκινάει από το μηδέν και τελειώνει στο μέγιστο αριθμό παιδιών του μείον ένα. Πχ, σε ένα δυαδικό δέντρο μπορεί να θεωρηθεί το δεξί παιδί ως 1 και το αριστερό ως 0.

Πέραν αυτού αρκεί να υποστηρίζει τις μεθόδους της επόμενης ενότητας ανεξαρτήτως εσωτερικής υλοποίησης.

2.3.5 Απαιτούμενες μέθοδοι του αντικειμένου Node

2.3.5.1 Ελάχιστες απαιτούμενες μέθοδοι

Το αντικείμενο Node πρέπει να υποστηρίζει τουλάχιστον τις εξής μεθόδους:

1. Έναν **copy constructor**, ένα τελεστή **copy assignment** και ένα **destructor** (Κανόνας των τριών)
2. Μία μέθοδο **NodeType* getChild(int i)**, η οποία επιστρέφει το παιδί υπ'αριθμό i
3. Μία μέθοδο **void setChild(int i, NodeType * n)**, η οποία θέτει το n ως παιδί υπ'αριθμό i
4. Μία μέθοδο **NodeType** getChildPointer(int i)**, η οποία επιστρέφει δείκτη στο παιδί υπ'αριθμό i
5. Μία μέθοδο **static constexpr int maxChildren()**, η οποία επιστρέφει το μέγιστο αριθμό παιδιών ανα κόμβο και είναι compile-time σταθερά

2.3.5.2 Μέθοδοι δέντρου αναζήτησης

Με την υποστήριξη των παρακάτω επιπλέον μεθόδων, αν πρόκειται για δέντρο αναζήτησης, η βιβλιοθήκη παρέχει επιπλέον χρήσιμες μεθόδους αυτόματα στο χρήστη και πετυχαίνει και υψηλότερη απόδοση.

1. Μία μέθοδο **bool hasKey(int key)**, η οποία επιστρέφει αν ο κόμβος περιέχει το κλειδί key
2. Μία μέθοδο **bool traversalDone(int key)**, η οποία επιστρέφει αν η αναζήτηση για το κλειδί key ολοκληρώθηκε στον συγκεκριμένο κόμβο. Χρήσιμη αν το δέντρο περιέχει παραπάνω από μία φορές ένα κλειδί (όπως τα external search trees)

3. Μία μέθοδο **int nextChild(int key)**, η οποία επιστρέφει σε ποιό παιδί ενός κόμβου πρέπει να συνεχίσει η αναζήτηση για το δοσμένο κλειδί

4. Μία μέθοδο **int nextChild(NodeType* target)**, η οποία επιστρέφει σε ποιό παιδί ενός κόμβου πρέπει να συνεχίσει η αναζήτηση για το δοσμένο κόμβο

2.4 Βασικές Έννοιες

2.4.1 Πράξεις

Ως πράξη ορίζουμε ένα σύνολο ενεργειών που πράττονται πάνω σε μία δομή. Πράξη αποτελεί πχ. η εισαγωγή ενός ζεύγους κλειδιού τιμής σε μία δομή χάρτη(map).

2.4.2 Πράξεις Ανάγνωσης (Read Only)

Οι παράλληλες δομές που υλοποιούνται μέσω της βιβλιοθήκης επιτρέπουν την υλοποίηση πράξεων ανάγνωσης *ολόϊδιων* με τις αντίστοιχες της σειριακής τους υλοποίησης, επομένως σε αυτές δε θα γίνει άμεση αναφορά. Μονάχα ένας περιορισμός υπάρχει, οι πράξεις αυτές πρέπει να έχουν κατεύθυνση αποκλειστικά από τη ρίζα προς τα φύλλα.

Στην παρούσα υλοποίηση δεν παρέχεται μηχανισμός διαχείρισης μνήμης, επομένως δεν απαιτείται η χρήση κάποιου RCU Read κλειδώματος. Σε μελλοντικές υλοποιήσεις, όπου θα προστεθεί, θα απαιτείται απλά η αρχικοποίηση ενός αντικειμένου κλειδώματος που θα παρέχεται από τη βιβλιοθήκη.

2.4.3 Το αρχικό δέντρο

Ως αρχικό δέντρο αναφερόμαστε σε ένα σειριακό δέντρο που ορίζει ο χρήστης και ικανοποιεί τις απαιτήσεις δομής που προαναφέρθηκαν για τη χρήση της βιβλιοθήκης.

2.4.4 Σημείο Σύνδεσης

Το σημείο σύνδεσης είναι είτε η ρίζα του δέντρου είτε κάποιος κόμβος στον οποίο θα συνδεθεί ως παιδί του το δέντρο των αντιγράφων. Για κάποια συγκεκριμένη πράξη είναι ο κόμβος "κάτω" απο τον οποίο θα συμβούν οι αλλαγές και πάνω από τον οποίο δε θα μεταβληθεί το δέντρο. Ορίζεται συνήθως ταυτόχρονα ποιο παιδί του θα αλλάξει από την πράξη ατομικά.

2.4.5 Το δέντρο των αντιγράφων (*ΔΤΑ*)

Κάθε πράξη που τροποποιεί την παράλληλη δομή, υλοποιείται από την βιβλιοθήκη με την δημιουργία ενός **νέου υποδέντρου** που περιέχει αντίγραφα των αρχικών κόμβων με τις αλλαγές που θα γίνουν και νέους κόμβους, αλλά και αναφορές στους αρχικούς κόμβους. Αυτά περιγράφονται με μία νέα δενδρική δομή που αποτελείται από αντικείμενα που ονομάζουμε ασφαλείς κόμβους. Αυτή η νέα δενδρική δομή φροντίζει να **διατηρεί εσωτερικά αυτό το δέντρο των αντιγράφων, χωρίς αυτό να συνδέεται στο αρχικό δέντρο**.Αν ολοκληρωθεί επιτυχώς μία πράξη, **το δέντρο θα συνδεθεί στην θέση ενός παλιού υποδέντρου** με αντικατάσταση του **δείκτη παιδιού** του **σημείου σύνδεσης**.

Εφ'όσον στην παρούσα εργασία το ΔΤΑ δημιουργείται μονάχα μέσω των ασφαλών κόμβων, θα αναφερόμαστε στην δομή των ασφαλών κόμβων ως ΔΤΑ.

Η δημιουργία αυτού του δέντρου εξασφαλίζει δύο πράγματα,

1. πάντοτε θα υπάρχει κάποιο αποδεκτό υποδέντρο το οποίο μπορεί να προσπελάσει ένας αναγνώστης του δέντρου (κάθε αντίγραφο αντικαθίσταται από κάποιο άλλο αποδεκτό αντίγραφο)

2. μπορεί να προσομοιωθεί η πράξη ως μετάλλαξη ενός δέντρου αφού οι αλλαγές στο δέντρο αποθηκεύονται στο αντίγραφο.

Μία σημαντική ιδιότητα για την κατανόηση του δέντρου των αντιγράφων είναι το γεγονός ότι οι κόμβοι του ενδέχεται να έχουν ως παιδιά κόμβους του αρχικού δέντρου οι οποίοι δεν ανήκουν σε αυτό, εφόσον αυτοί δεν θα αλλαχτούν από κάποια πράξη. Με αυτό τον τρόπο είναι δυνατόν να μην αντιγραφούν αυτοί οι κόμβοι εφόσον δεν αλλάξουν αλλά και να μπορεί να επεκταθεί το δέντρο ώστε να περιέχει τα αντίγραφα τους όταν πρέπει να αλλαχθούν.

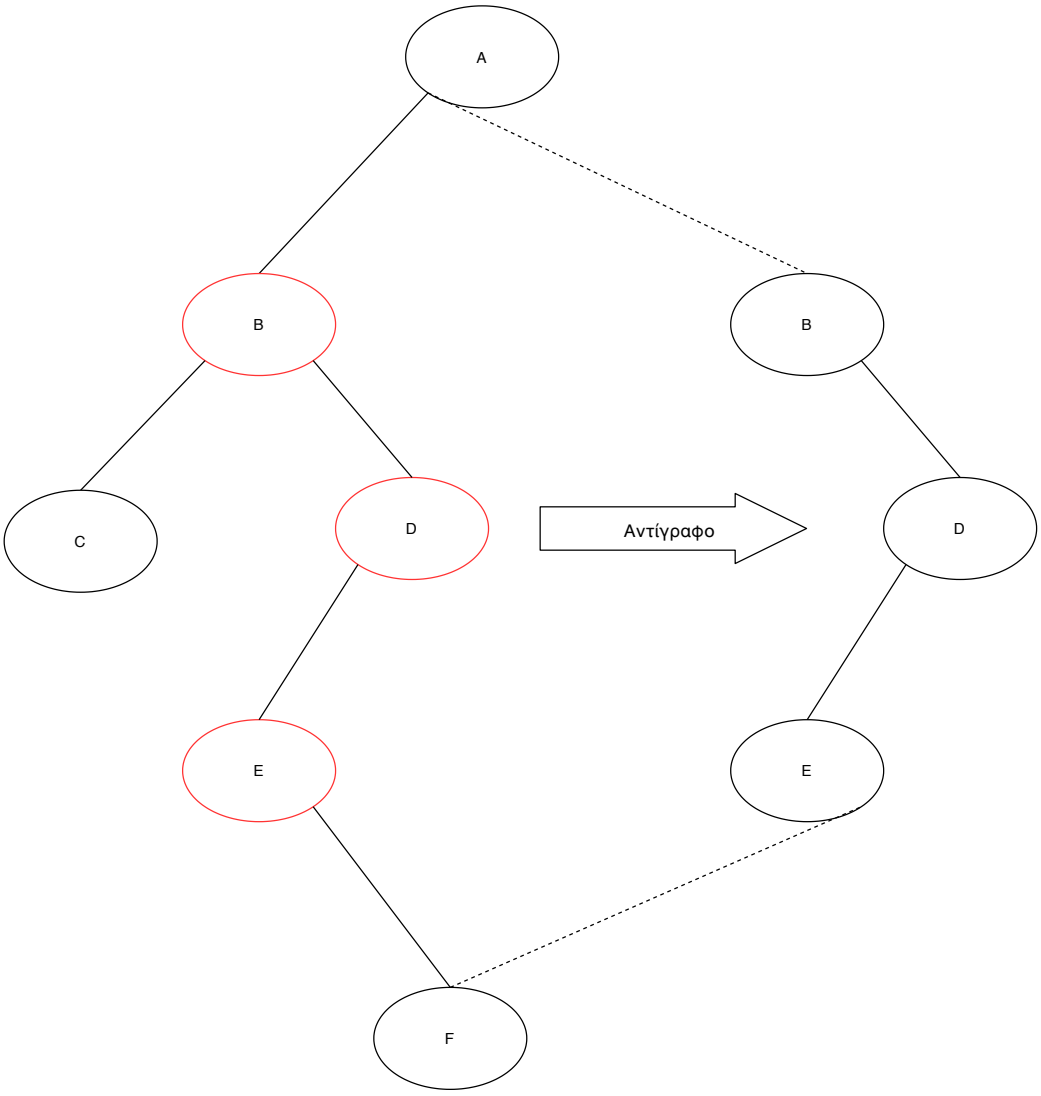
Ακολουθεί παράσταση του εσωτερικού ΔΤΑ που παράγεται για την αλλαγή ενός υποδέντρου, με κόκκινο παρουσιάζονται οι κόμβοι που θα αντικατασταθούν από το αντίγραφο τους. Προσοχή στο ότι δύναται και το αντίγραφο και ο αρχικός κόμβος E συνδέονται με τον κόμβο F.

2.4.6 Συναλλαγή

Ως συναλλαγή ορίζουμε ένα τμήμα κώδικα που θα εκτελεστεί ατομικά.

2.4.7 Επιβεβαίωση

Ως επιβεβαίωση αναφερόμαστε στη διαδικασία που συμβαίνει εντός συναλλαγής και εξασφαλίζει ότι η πράξη θα ολοκληρωθεί ατομικά.



Σχήμα 2.1: Το Δέντρο των Αντιγράφων

2.5 Βασικά Στοιχεία Βιβλιοθήκης

2.5.1 Block συναλλαγής

Κάθε πράξη μετάλλαξης της παράλληλης δομής πρέπει να περικλείεται εντός ενός block συναλλαγής. Το block συναλλαγής είναι ουσιαστικά ένα macro το οποίο διαχειρίζεται τη λογική της επανεκκίνησης της πράξης αν αποτύχει η επιβεβαίωση της. Κάθε πράξη που θέλουμε να παραλληλοποιηθεί από τη βιβλιοθήκη οφείλει να περικλείεται σε ένα τέτοιο block. Ορίζεται από τις δύο εντολές `TM_SAFE_OPERATION_START` και `TM_SAFE_OPERATION_END`.

Επιπλέον πρέπει να ρυθμιστούν οι παράμετροι της συναλλαγής, δηλ. ο αριθμός επαναπροσπάθειων(retries) της συναλλαγής προτού χρησιμοποιηθεί το κεντρικό κλείδωμα και το αντικείμενο των στατιστικών, με την αρχικοποίηση ενός αντικειμένου συναλλαγής `Transaction`.

Παραδείγμα:

```
// binary search tree insert method
bool insert(int key, int value) {

    int retries = 20;

    TM_SAFE_OPERATION_START {
        Transaction trans(retries, this->global_lock, this->stats);

        ...operation code...

    } TM_SAFE_OPERATION_END

    return true;
}
```

2.5.2 Αναζήτηση Σημείου Σύνδεσης - Αυτόματη και PathTracker

Κάθε πράξη της βιβλιοθήκης ξεκινάει με την εύρεση του σημείου σύνδεσης και του παιδιού του που θα αλλάξει. Το σημείο αυτό πρέπει να βρίσκεται εντός ενός αντικειμένου *ConnPointData*, το οποίο περιέχει διάφορα δεδομένα τα οποία απαιτεί για τη λειτουργία της η βιβλιοθήκη. Αυτά τα δεδομένα θα συζητηθούν στην τεχνική ανάλυση ωστόσο πρέπει να αναφερθούμε στο βασικότερο, *τη στοίβα του μονοπατιού*. Η διάσχιση του δέντρου μέχρι το σημείο σύνδεσης αποθηκεύεται σε μία στοίβα και υπάρχει διαθέσιμη μέσω αυτού του αντικειμένου. Για την παραγωγή αυτού του αντικειμένου υπάρχουν δύο ξεχωριστοί μηχανισμοί ανάλογα με το εαν το δέντρο είναι δέντρο αναζήτησης ή γενική δεντρική δομή.

2.5.2.1 Δέντρο Αναζήτησης

Στο δέντρο αναζήτησης, η συνάρτηση εύρεσης του σημείου σύνδεσης για μία πράξη παρέχεται αυτόματα από τη βιβλιοθήκη, χρησιμοποιώντας τις παρεχόμενες μεθόδους του κόμβου. Συγκεκριμένα παρέχεται η συνάρτηση

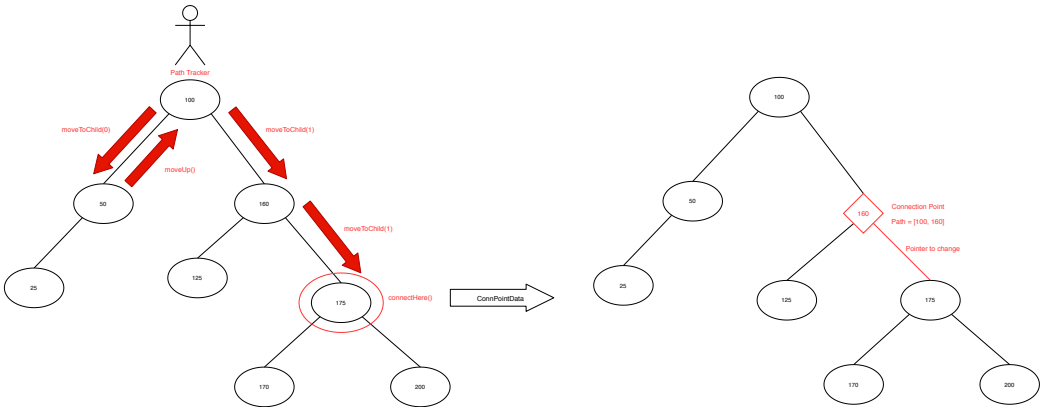
```
ConnPointData<NodeType>
find_conn_point<NodeType>(int key, NodeType** root)
```

η οποία δέχεται σαν παράμετρο τύπου τον τύπο του κόμβου και σαν όρισμα το κλειδί του επιθυμητού κόμβου και χρησιμοποιώντας τις μεθόδους **traversalDone(int key)** , **nextChild(int key)** εντοπίζει το σημείο σύνδεσης και το τοποθετεί σε ένα αντικείμενο `ConnPointData`. *Τέλος είναι σημαντικό να τονίσουμε πως για δέντρα αναζήτησης, η χρήση των αυτόματων μεθόδων ενεργοποιεί βελτιστοποιήσεις απόδοσης και ελαχιστοποιεί τα πιθανά λάθη στον προγραμματισμό επομένως προτείνεται.*

2.5.2.2 Γενική Δεντρική Δομή

Για την συλλογή των απαραίτητων δεδομένων αλλά και την ευκολία προγραμματισμού, προσφέρεται στο χρήστη σαν αφαιρετικός μηχανισμός το αντικείμενο *PathTracker*. Το αντικείμενο αυτό λειτουργεί σαν μία μηχανή που καταγράφει τη διαδρομή που έχει ακολουθηθεί σε ένα δέντρο και τη χρησιμοποιεί για την παραγωγή του *ConnPointData*. Μπορεί να παρομοιαστεί με ένα *iterator* ο οποίος επιτρέπει τη διάσχιση του δέντρου και δείχνει σε κάποιο κόμβο του δέντρου.Παρέχει τις μεθόδους:

1. `NodeType* getNode()`, η οποία επιστρέφει τον δείκτη στον κόμβο στον οποίο δείχνει το αντικείμενο
2. `void moveToChild(int index)`, η οποία μετακινεί τον PathTracker στο παιδί `index`του τωρινού κόμβου
3. `void moveUp(int n_steps)`, η οποία μετακινεί τον PathTracker `n_steps`φορές προς την κατεύθυνση του γονέα
4. `ConnPointData connectHere()`, η οποία θέτει τον τωρινό κόμβο ως σημείο σύνδεσης και επιστρέφει το ConnPointData



Σχήμα 2.2: PathTracker

2.5.3 Αναζήτηση Σημείου Σύνδεσης - Παραδείγματα Κώδικα

2.5.3.1 Αναζήτηση σε γενικό δέντρο

Θα δείξουμε πως μπορεί να υλοποιηθεί η συνάρτηση `find_conn_point` στο ίδιο δέντρο αναζήτησης με βάση το αντικείμενο `PathTracker`.

```
ConnPointData<TreeNode> find_conn_point_custom(int k, bool& found) {
    // initialize PathTracker with pointer to root
    PathTracker<TreeNode> tracker(&root);

    TreeNode* curr_node;
    int curr_key;

    // while tracker does not point to null
    while ((curr_node = tracker.getNode()))
    {
        // get the key
        // of the current node
        curr_key = curr_node->getKey();

         // if key is same as target
        if (curr_key == k) {

            found = true;   // declare it as found

            // return ConnPointData
            //with node as Connection Point
            return tracker.connectHere();

        } else if (k < curr_key) {
            tracker.moveToChild(0); // move left
        } else {
            tracker.moveToChild(1); // move right
        }
    }
    found = false;
    return tracker.connectHere();
}
```

Αρχικοποιούμε το αντικείμενο με τη διεύθυνση της διεύθυνσης της ρίζας και μετακινούμαστε μέχρι το σημείο που θέλουμε να γίνει το σημείο σύνδεσης, όπου καλούμε τη μέθοδο `connectHere()`,η οποία θα επιστρέψει το αντικείμενο `ConnPointData` με την στοίβα διαδρομής και το σημείο σύνδεσης.

2.5.3.2 Αναζήτηση σε δέντρο αναζήτησης

Εδώ τα πράγματα όπως αναφέρθηκε είναι απλά:

```
// in the insert method of a bst

// find the connection point for inserting with key k
auto conn_point_snapshot = find_conn_point<TreeNode>(k,&root);

// if not found return false
if (conn_point_snapshot.found()) {
    return false;
}
```

2.5.4 SafeNode (Ασφαλής Κόμβος)

Οι ασφαλείς κόμβοι αποτελούν το βασικότερο δομικό στοιχείο για την κατασκευή thread safe δεντρικών δομών με τη βιβλιοθήκη. Είναι αντικείμενα που ορίζονται με βάση κάποιο αντικείμενο κόμβου με τη χρήση C++ templates. Ο τύπος τους δηλαδή ορίζεται ως `SafeNode<TreeNode>`, όπου `TreeNode`,το αντικείμενο κόμβου.

Έχουν τον ρόλο των κόμβων του δέντρου των αντιγράφων και εξασφαλίζουν ότι με τη χρήση τους, **κάθε τροποποίηση στην παράλληλη δομή θα συμβαίνει ατομικά**. Περισσότερες λεπτομέρειες για το πως συμβαίνει αυτό θα αναφερθούν στην τεχνική ανάλυση.

Οι ασφαλείς κόμβοι χρησιμοποιούνται ώστε να περιγράψουν την αλλαγή που θέλουμε να συμβεί σε ένα υποδέντρο. Είναι σχεδιασμένοι ώστε να παρέχουν παρόμοιες μεθόδους με τους αρχικούς κόμβους του δέντρου. Η **δημιουργία ενός δέντρου αντιγράφων** μέσω αυτών των κόμβων θα έχει ως τελικό αποτέλεσμα **τη δημιουργία ενός υποδέντρου αποτελούμενου από τους user-defined κόμβους μετά το τέλος της πράξης** αλλά χωρίς τους κινδύνους του ταυτοχρονισμού.

Έμφαση πρέπει να δοθεί στο γεγονός ότι οι κόμβοι αυτοί *δεν γίνονται οι κόμβοι της παράλληλης δομής*. Δημιουργούνται κατα την διάρκεια μίας πράξης για τη διαχείριση των αλλαγών και στην επιτυχή ολοκλήρωση της ή σε κάποια αποτυχία αποσύρονται. Οι ασφαλείς κόμβοι δρουν ως **ένας wrapper γύρω από έναν αρχικό κόμβο του δέντρου** και προσφέρουν ένα σύντομο και εύχρηστο API,με τη μορφή μεθόδων, για την **περιγραφή αλλαγών στο δέντρο**. Η ανάγνωση και τροποποίηση παιδιών ενός SafeNode μέσω αυτού εγγυάται ότι **τα δεδομένα προέρχονται από αποδεκτή κατάσταση του δέντρου** και ότι **οι τροποποιήσεις θα γίνουν ατομικά**. Ταυτόχρονα ο απαιτούμενος κώδικας απαιτεί μικρές αλλαγές συγκριτικά με τον σειριακό.

Αρμοδιότητες του ασφαλούς κόμβου

1. Διατήρηση αναφορών στον αρχικό κόμβο του δέντρου και στο αντίγραφο του
2. Διαχείριση διασύνδεσης αντιγράφων μεταξύ τους αλλά και με το αρχικό δέντρο
3. Διατήρηση αλλαγών στο δέντρο
4. Δυναμική δημιουργία αντιγράφων μόνο κατά την προσπέλαση για μετάλλαξη (παρόμοια με τεχνική COW)
5. Διατήρηση διευθύνσεων των παιδιών του αρχικού κόμβου τη στιγμή της αντιγραφής για την διαδικασία της *επιβεβαίωσης*

2.5.4.1 Υποστηριζόμενες μέθοδοι

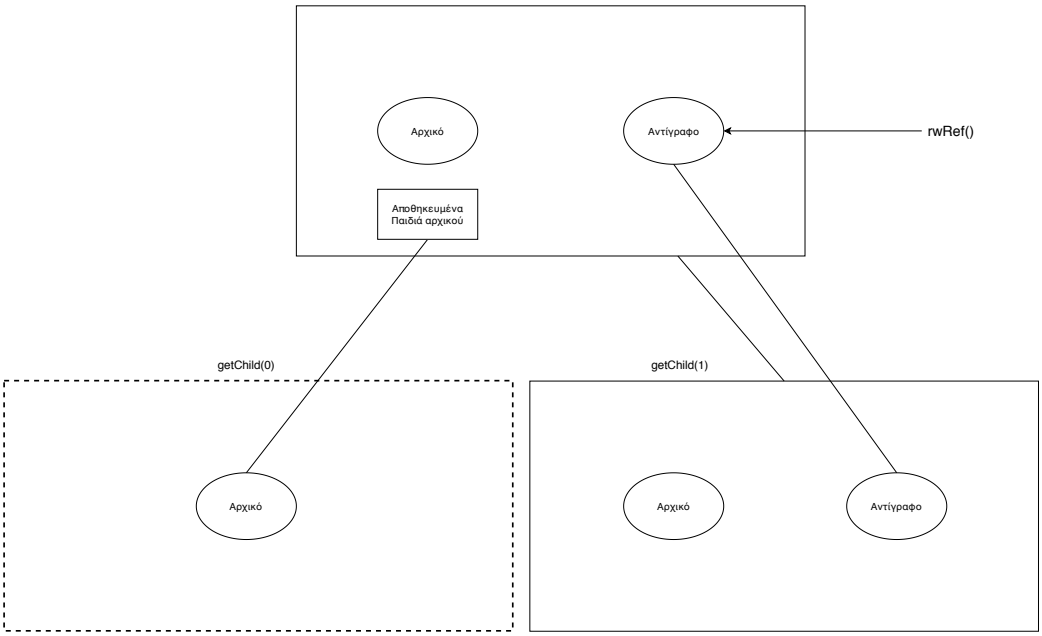
1. **SafeNode<NodeType>* getChild(const int index)**, επιστρέφει το επιθυμητό παιδί του κόμβου που περιέχεται στον ασφαλή κόμβο, επίσης τοποθετημένο εντός ενός ασφαλούς κόμβου. Κάνει δηλαδή έμμεσο τύλιγμα (βλ. `ConnPoint`). Αν έχει προσπελαστεί ξανά, επιστρέφεται άμεσα ο ασφαλής κόμβος που έχει αποθηκευτεί, αλλιώς δημιουργείται ένας νέος ασφαλής κόμβος που περιέχει το παιδί του αρχικού κόμβου. Επιστρέφει nullptr αν δεν υπάρχει παιδί.

2. **SafeNode<NodeType>* peekChild(const int index)**, επιστρέφει αναφορά στο παιδί του αρχικού κόμβου με δείκτη index χωρίς να το περικλείσει σε ασφαλή κόμβο. Χρησιμεύει στην περίπτωση που απλά θέλουμε να ξέρουμε τη διεύθυνση του κόμβου παιδιού. Η βιβλιοθήκη εξασφαλίζει ότι αν επιτύχει η επιβεβαίωση ο κόμβος έχει όντως αυτόν τον κόμβο ως παιδί ως το πέρας της συναλλαγής αλλά δεν εξασφαλίζει τίποτα για τα παιδιά αυτού του κόμβου.

3. **SafeNode<NodeType>* setChild(const int index, SafeNode<NodeType>* newChild)**, θέτει τον ασφαλή κόμβο newChild ως παιδί με δείκτη index, Ο κόμβος αυτός θα είναι διαθέσιμος σε επόμενη προσπέλαση από την μέθοδο getChild αλλά επιπλέον θα συνδεθούν και οι αρχικοί κόμβοι στο *ΔΤΑ*

4. **NodeType* rwRef()**, επιστρέφει μία ασφαλή αναφορά, δηλ. ένα δείκτη στο αντίγραφο του περιεχόμενου κόμβου (δημιουργώντας το αν δεν έχει δημιουργηθεί), ο οποίος επιτρέπει την πραγματοποίηση αλλαγών στα περιεχόμενα δεδομένα με thread-safe τρόπο. Οι αλλαγές αυτές θα αναιρεθούν αν αποτύχει η επιβεβαίωση της συναλλαγής

5. **NodeType* peekOriginal()**, επιστρέφει μία μη ασφαλή αναφορά στον αρχικό κόμβο



Σχήμα 2.3: Ο Ασφαλής Κόμβος - Πρώτη (Αριστερά) και Επανειλλημένη Πρόσβαση σε Παιδί

2.5.4.2 Πως χρησιμοποιείται

Όταν θέλουμε να κάνουμε κάποια αλλαγή σε κάποιον κόμβο, αρχικά Τον προσπελαύνουμε ξεκινώντας από την ρίζα των δέντρων των αντιγράφων καλώντας την μέθοδο getChild. Η ρίζα του δέντρου των αντιγράφων, που είναι ασφαλής κόμβος, παρέχεται αυτόματα από τη μέθοδο getRoot() του αντικειμένου ConnPoint αλλά οι υπόλοιποι κόμβοι δημιουργούνται δυναμικά κατά τη διάσχιση. Στη συνέχεια υπάρχουν δύο περιπτώσεις:

1. Θέλουμε να αλλάξουμε **τα παιδιά του κόμβου**, πχ να θέσουμε ένα νέο παιδί, οπότε περικλέιουμε το νέο παιδί σε ένα ασφαλή κόμβο (wrapSafe(), createSafe() κτλπ.) και το θέτουμε με την μέθοδο setChild

2. Θέλουμε να αλλάξουμε **τα περιεχόμενα δεδομένα του κόμβου**, οπότε λαμβάνουμε μία ασφαλή αναφορά με την μέθοδο rwRef() και αλλάζουμε τα δεδομένα ακριβώς όπως και στο σειριακό κώδικα

2.5.5 SafeNode - Παραδείγματα Κώδικα

Θα δούμε παραδείγματα των μεθόδων του ασφαλούς κόμβου μέσα από μερικές χαρακτηριστικές χρήσεις.

2.5.5.1 Πρώτη χρήση - Ανταλλαγή αριστερού και δεξιού παιδιού ενός κόμβου δυαδικού δέντρου

```
void swapLeftRight(SafeNode<MyNode>* sample_node) {
    auto left_child = sample_node->getChild(0); // read left child
    auto right_child = sample_node->getChild(1); // read right child

    sample_node->setChild(0, right_child); // set right as left
    sample_node->setChild(1, left_child);  // set left as right
}
```

Ο κώδικας μοιάζει εξαιρετικά με τον σειριακό και είναι αυτοεξηγούμενος αλλά η βιβλιοθήκη εξασφαλίζει ότι είναι thread-safe. Προσοχή στο γεγονός ότι χρησιμοποιούνται αποκλειστικά ασφαλείς κόμβοι για τη συνδεσμολογία.

2.5.5.2 Δεύτερη χρήση - Ενημέρωση τιμής ενός κόμβου δέντρου με ζεύγη κλειδιού-τιμής

```
void updateValue(SafeNode<MyNode>* sample_node, ValueType new_value) {
    MyNode* sample_node_ref = sample_node->rwRef(); // get a rw reference
    sample_node_ref->value = new_value;
}
```

Ξανά ο κώδικας μοιάζει εξαιρετικά με τον σειριακό και βλέπουμε πως η μετάλλαξη συμβαίνει με τη χρήση ασφαλούς αναφοράς.

2.5.5.3 Τρίτη χρήση - Ανταλλαγή κόμβων παιδιών εφ'όσον το άθροισμα των κλειδιών τους είναι ζυγό

```
void swapSumEven(SafeNode<MyNode>* target_node) {
    auto left_key = target_node->peekChild(0)->key;
    auto right_key = target_node->peekChild(0)->key;

    if ((left_key + right_key) % 2 == 0) {
        swapLeftRight(target_node);
    }
}
```

Επειδή δεν γνωρίζουμε αν χρειάζεται όντως να κάνουμε κάποια αλλαγή στο δέντρο, χρησιμοποιούμε την peekChild για να επιβεβαιώσουμε εάν ισχύει η συνθήκη και εφ'όσον ισχύει, καλούμε την μέθοδο swapLeftRight η οποία θα οδηγήσει στη δημιουργία αντιγράφων εφ'όσον καλεί την μέθοδο getChild.

2.5.6 ConnPoint (Σημείο σύνδεσης)

Το σημείο σύνδεσης (`ConnPoint`)είναι το αντικείμενο το οποίο περιέχει την ρίζα του δέντρου των αντιγράφων και τον κόμβο στον οποίο θα εγκατασταθεί (σημείο σύνδεσης). Αναλαμβάνει την επιβεβαίωση και την τελική εγκατάσταση του δέντρου των αντιγράφων στο αρχικό δέντρο. Δημιουργείται πάντοτε εντός block συναλλαγής και χρησιμοποιώντας ως ορίσματα το αντικείμενο `ConnPointData`και το αντικείμενο συναλλαγής `Transaction`.

Η πρώτη και βασική του χρήση είναι **η διατήρηση της ρίζας του δέντρου των αντιγράφων**. Η ρίζα αυτή θα συνδεθεί ως το παιδί του σημείου σύνδεσης (ή ως νέα ρίζα του αρχικού δέντρου). Για αυτό το λόγο παρέχονται τρεις μέθοδοι:

1. Η μέθοδος `SafeNode<NodeType>* getRoot()`, η οποία έχει δύο πιθανές συμπεριφορές. Στην περίπτωση που *δεν έχει οριστεί ρίζα*, περικλείει το παιδί του σημείου σύνδεσης που θα αλλάξει μέσα σε ένα ασφαλή κόμβο, το θέτει ως ρίζα του δέντρου των αντιγράφων και το επιστρέφει. Στην περίπτωση που *έχει οριστεί ήδη η ρίζα*, την επιστρέφει

2. Η μέθοδος `SafeNode<NodeType>* setRoot(SafeNode<NodeType>* new_root)`, η οποία θέτει ένα ασφαλή κόμβο ως νέα ρίζα του δέντρου των αντιγράφων. Επιστρέφει τη νέα ρίζα για ευκολία χρήσης

3. Η μέθοδος `SafeNode<NodeType>* newTree(NodeType* new_root)`, που συνδέει ένα καινούριο κόμβο ως ρίζα του δέντρου αντιγράφων αφού τον τοποθετήσει σε ένα ασφαλή κόμβο

Με τη χρήση αυτών των μεθόδων αρχικοποιείται το *ΔΤΑ* και μπορεί να τροποποιηθεί μέσω αυτού το όλο το αντίστοιχο υποδέντρο του σημείου σύνδεσης.

Πότε εμφανίζεται η αλλαγή στο αρχικό δέντρο Η επιβεβαιώση και εγκατάσταση του νέου υποδέντρου (αποτελούμενο από τους κόμβους που έχουν αποθηκευτεί εσωτερικά στους ασφαλείς κόμβους του δέντρου των αντιγράφων) συμβαίνει με την κλήση του destructor του αντικειμένου ConnPoint.

2.5.6.1 Δημιουργία Ασφαλών Κόμβων

Η δεύτερη και βασική χρήση του αντικειμένου ConnPoint είναι η δημιουργία ασφαλών κόμβων από αντικείμενα κόμβου. Τη διαδικασία αυτή θα την ονομάσουμε "**τύλιγμα**". Τυλίγοντας ένα αντικείμενο κόμβου σε έναν ασφαλή κόμβο, τοποθετούμε αυτόν τον κόμβο στο **σύνολο επιβεβαίωσης** του `ConnPoint`. Το σύνολο επιβεβαίωσης εξασφαλίζει ότι **η πράξη δεν θα ολοκληρωθεί εάν κατά τη διάρκεια της αλλάξουν οι διευθύνσεις των παιδιών των κόμβων που περιέχει**. Αυτή η ιδιότητα μας εξασφαλίζει το thread-safety άρα πρέπει να ισχύει για κάθε αλλαγή. Για αυτό, κάθε αλλαγή στις διευθύνσεις των παιδιών ενός κόμβου πρέπει να γίνεται μέσω μεθόδων ενός ασφαλούς κόμβου.

Για το τύλιγμα των αντικειμένων κόμβου λοιπόν έχουμε δύο επιλογές, η μία είναι το έμμεσο τύλιγμα μέσω των μεθόδων του ασφαλή κόμβου αλλά και το *άμεσο τύλιγμα* μέσω των μεθόδων του `ConnPoint`.Για αυτό παρέχονται τρεις μέθοδοι:

1. Η μέθοδος `SafeNode<NodeType>* wrap_safe(NodeType* some_node)`, η οποία τυλίγει ένα κόμβο και τον **προσθέτει στο σύνολο επιβεβαίωσης** του `ConnPoint`.Κάθε κόμβος που θα *τυλιχθεί έμμεσα* μέσω της μεθόδου `getChild` αυτού του κόμβου **θα προστεθεί επίσης στο σύνολο επιβεβαίωσης**.

2. Η μέθοδος `SafeNode<NodeType>* create_safe(NodeType* some_node)`, η οποία τυλίγει ένα κόμβο **χωρίς να τον προσθέτει στο σύνολο επιβεβαίωσης** του `ConnPoint`.Χρησιμοποιείται για την σύνδεση νεοδημιούργητων κόμβων που δεν ανήκουν ήδη στο δέντρο άρα δεν απαιτούν επιβεβαίωση. Κάθε κόμβος που θα *τυλιχθεί έμμεσα* μέσω της μεθόδου `getChild` αυτού του κόμβου **δεν θα προστεθεί στο σύνολο επιβεβαίωσης**.

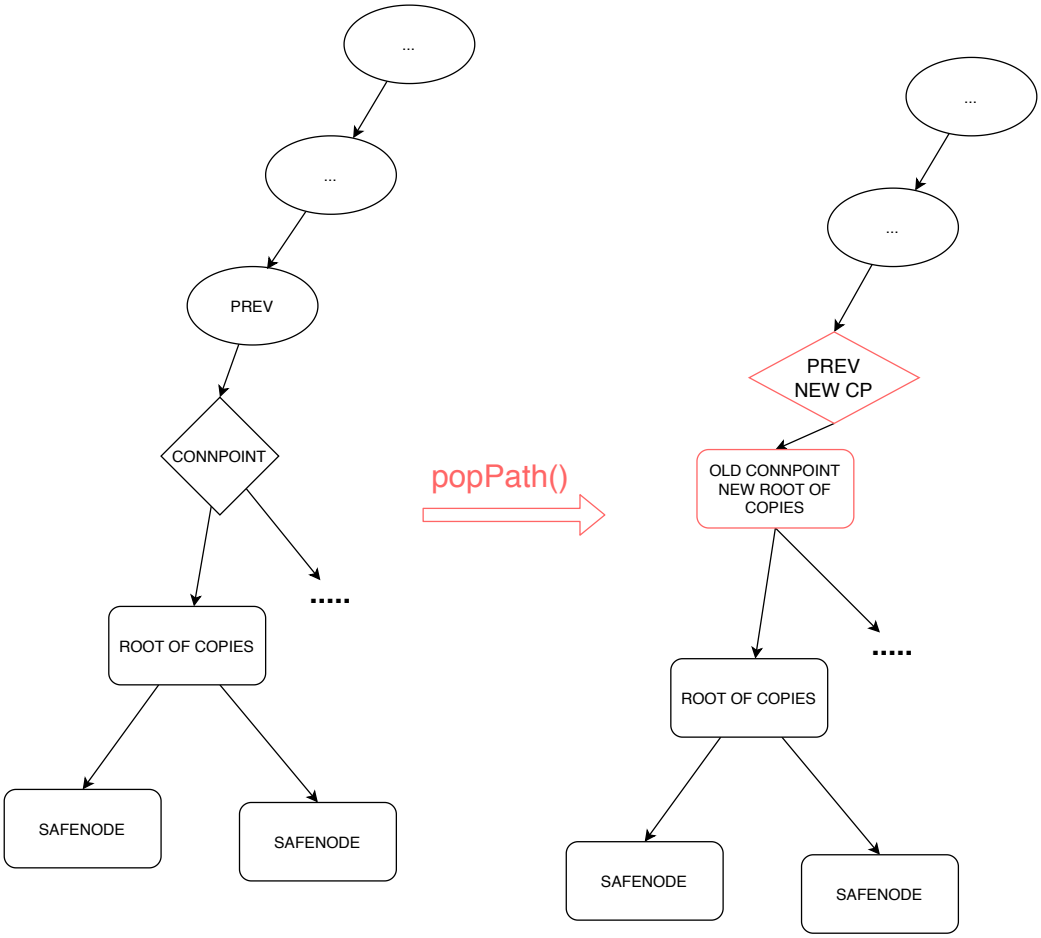
3. Η μέθοδος `SafeNode<NodeType>* wrap_no_validate`, η οποία λειτουργεί πανομοιότυπα με την `create_safe` αλλά έχει ως βασική διαφορά ότι **σηματοδοτεί πως οι κόμβοι που περιέχονται στους ασφαλείς κόμβους δεν είναι νεοδημιούργητοι αλλά ανήκουν στο αρχικό δέντρο** (άρα δεν είναι ασφαλές να σβηστούν σε περίπτωση που αποτύχει η επιβεβαίωση)

2.5.6.2 Μετακίνηση προς ρίζα

Σε κάποιες περιπτώσεις **δεν είναι δυνατός ο προσδιορισμός του κόμβου που θα πρέπει να είναι το σημείο σύνδεσης**. Σε μία εισαγωγή ενός ισοζυγισμένου δέντρου ,ως παράδειγμα, πιθανόν να χρειάζονται μετατροπές στο δέντρο ξεκινώντας από ένα κόμβο που βρίσκεται χαμηλά και ανεβαίνοντας προς τη ρίζα. Για την κάλυψη αυτής της βασικής λογικής, το αντικείμενο `ConnPoint` προσφέρει τη μέθοδο `SafeNode<NodeType>* popPath()`.

Η μέθοδος αυτή αξιοποιεί τη **στοίβα μονοπατιού** που δημιουργήθηκε στην αναζήτηση του αρχικού σημείου σύνδεσης ώστε να **θέσει το γονικό κόμβο του τωρινού σημείου σύνδεσης ως το νέο σημείο σύνδεσης**, ενώ ταυτόχρονα **τοποθετεί το παλιό σημείο σύνδεσης ως ρίζα του δέντρου των αντιγράφων** (εντός ασφαλούς κόμβου) και **το συνδέει και με το προϋπάρχον δέντρο**. Έτσι ουσιαστικά είναι σαν η αρχική αναζήτηση να είχε θέσει ως σημείο σύνδεσης το γονικό κόμβο.

Με αυτό τον τρόπο καθίσταται δυνατή η τροποποίηση κόμβων που βρίσκονται σε **υψηλότερο επίπεδο από το σημείο σύνδεσης**, θέτοντας υψηλότερα το σημείο σύνδεσης, επομένως ικανοποιώντας την απαίτηση ότι όλες οι αλλαγές συμβαίνουν κάτω από το σημείο σύνδεσης. Η μέθοδος επιστρέφει τη νέα ρίζα του δέντρου των αντιγράφων (δηλ. το παλιό σημείο σύνδεσης) μέσα σε ασφαλή κόμβο ή `nullptr` εάν δεν υπάρχει γονικός κόμβος, άρα το δέντρο θα εγκατασταθεί ως ρίζα του αρχικού δέντρου.



Σχήμα 2.4: Μετακίνηση Προς Ρίζα

2.5.7 ConnPoint - Παραδείγματα Κώδικα

2.5.7.1 Αρχικοποίηση σημείου σύνδεσης

```
// skeleton of every operation
TM_SAFE_OPERATION_START {

    // init transaction
    TSX::Transaction t(retries,_lock, stats[t_id]);

    // find connection point
    auto conn_point_snapshot = find_conn_point<TreeNode>(k,&root);

    // initialize ConnPoint with required objects
    ConnPoint<TreeNode> conn(conn_point_snapshot, t);

    // do operations using ConnPoint
    ..............
}
```

Κάθε πράξη που υλοποιείται μέσω της βιβλιοθήκης έχει αυτό το σκελετό.

2.5.7.2 Διάβασμα και εγκατάσταση νέας ρίζας

```
// in transaction block
{
    // skeleton here
    ...

    // conn is the ConnPoint object

    SafeNode<NodeType>* curr_root = conn.getRoot();

    // SafeNode new_root will be installed if there's no root

    if (!curr_root) {
        conn.setRoot(new_root);
    }
}
```

Στο τέλος αυτής της πράξης θα κληθεί ο καταστροφέας του ConnPoint αντικειμένου και αν το σημείο σύνδεσης δεν είχε παιδί, θα τεθεί ο εσωτερικός κόμβος του `new_root` αλλιώς δε θα συμβεί τίποτα.

2.5.7.3 Εξισορρόπηση δέντρου - popPath

Όπως αναφέρθηκε η εξισορρόπηση είναι καλό παράδειγμα χρήσης της μεθόδου `popPath()`. Αρχικά βρίσκουμε το σημείο σύνδεσης και συνδέουμε το νέο κόμβο ως παιδί του κάνοντας το ρίζα (και μοναδικό κόμβο) του δέντρου των αντιγράφων. Τώρα, όμως πιθανώς χαλάσαμε την ισορροπία του δέντρου.Επομένως:

- Επεκτείνουμε το *ΔΤΑ* θέτοντας το σημείο σύνδεσης ως νέα ρίζα και θέτουμε ως νέο σημείο σύνδεσης το γονέα του
- Εφαρμόζουμε μία περιστροφή ώστε να εξισορροπήσουμε το δέντρο ως αυτό το σημείο - η `rebalance_ins` χρησιμοποιεί ασφαλείς κόμβους και πιθανόν προεκτείνει επίσης το *ΔΤΑ* με ό,τι αλλαγές συμβούν
- Θέτουμε το αποτέλεσμα του rotation ως τη νέα ρίζα.

Κάνοντας αυτό επαναληπτικά ωσπού να φτάσουμε στη ρίζα του δέντρου ή να ισορροπήσει το δέντρο, κατασκευάζουμε ένα *ΔΤΑ* με όλες τις απαιτούμενες αλλαγές και καταλήγουμε σε ένα σημείο σύνδεσης στο οποίο πρέπει να συνδεθεί.

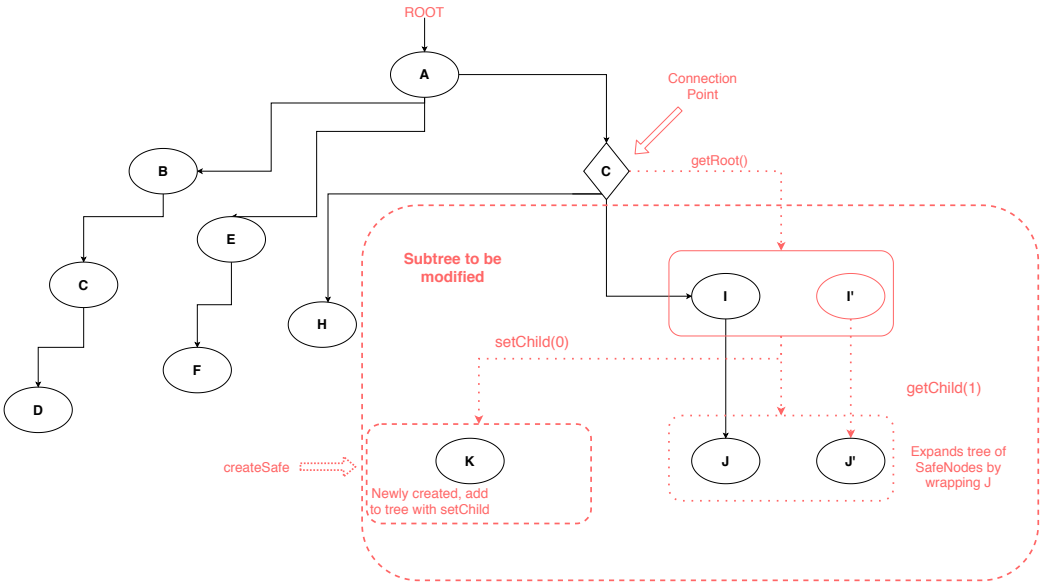
2.5.7.4 Παράδειγμα Κώδικα

```
// inserting in AVL Tree

// in skeleton
...

// conn is the ConnPoint object

// new node will be added below connection point
conn.setRoot(conn.wrap_safe(new AVLNode(...));

// but tree could be unbalanced now
// let's rebalance it

// keep setting parent as new connection point as long as it is
//unbalanced

for (SafeNode<TreeNode>* n = conn.popPath(); n != nullptr; n = conn.popPath()) {
    // get a reference to read data
    auto n_data = n->rwRef();
    int height_old = n_data->height;

    // do a rebalance of the current node
    n = rebalance_insert(n, k , rotation_happened);

    // set the result of the rebalancing as the new root
    // (to be inserted below the current connection point)
    conn.setRoot(n);    // change the root of the
                        // tree of copies to make the change visible

    // if balanced, break
    if (height_old == n_data->height && !rotation_happened) {
        break;
    }
    ...
```

2.6 Σύνοψη

Για την παραλληλοποίηση μίας πράξης μίας δεντρικής δομής:

1. Δημιουργείται το μπλοκ συναλλαγής
2. Εντοπίζεται και εγκαθίσταται το σημείο σύνδεσης με αυτόματο ή μη τρόπο, αρχικοποιείται το αντικείμενο ConnPoint
3. Από το αντικείμενο ConnPoint, ξεκινάει το ΔΤΑ με την πρόσβαση στη ρίζα του με τη μέθοδο *getRoot()*
4. Οι μέθοδοι και συναρτήσεις μετατρέπονται ώστε να χρησιμοποιούν ασφαλείς κόμβους μέσω αλλαγής των τύπων τους. Επεκτείνεται το δέντρο των ασφαλών κόμβων είτε προς τα παιδιά είτε προς την ρίζα του αρχικού δέντρου αυτόματα με τις προσβάσεις. Προσθήκες κόμβων γίνονται τυλίγοντας τους σε ασφαλείς κόμβους και θέτοντας τους με τη χρήση της μεθόδου. `setChild`
5. Το αυτόματα δημιουργημένο ΔΤΑ συνδέεται αυτόματα όταν επιτύχει η συναλλαγή



Σχήμα 2.5: Η βιβλιοθήκη παραλληλοποίησης

2.7 Παραλληλοποιώντας ένα map βασισμένο σε AVL δέντρο

2.7.1 Εισαγωγή

Στην παρούσα ενότητα θα δείξουμε τη διαδικασία παραλληλοποίησης ενός AVL δέντρου αναζήτησης, παρουσιάζοντας τις βασικότερες του πράξεις. Το AVL δέντρο αποτελεί ιδανικό παράδειγμα αφού αξιοποιεί όλα τα εργαλεία της βιβλιοθήκης μας. Μικρές λεπτομέρειες που αφορούν βοηθητικές μεθόδους θα προσπεραστούν ωστόσο η ολοκληρωμένη υλοποίηση μπορεί να βρεθεί από τις αναφορές στο τέλος της εργασίας.

2.7.2 Απαιτούμενα includes

Όλες οι απαραίτητοι μέθοδοι και αντικείμενα της βιβλιοθήκης υπάρχουν στο header αρχείο `SafeTree,hpp` ,υπό του χώρου ονομάτων `TSX`.

2.7.3 Το αντικείμενο Κόμβου - AVLNode

Παρακάτω δίνεται μία ελάχιστη υλοποίηση ενός κόμβου AVL δέντρου, το οποίο είναι και δέντρο. Οι μέθοδοι είναι απλές και αυτοεξηγούμενες.
Ιδιαίτερη μνεία θα δωθεί μονάχα στις μεθόδους `traversalDone` και `nextChild`. Η `traversalDone` όπως παρατηρούμε επιστρέφει `true` όταν το κλειδί της αναζήτησης ισούται με το κλειδί του κόμβου. Αυτό δεν επιτρέπει την ύπαρξη διπλότυπων κλειδιών στο δέντρο, αφού η αναζήτηση θα τερματίσει στο πρώτο αποτέλεσμα. Αυτός είναι και ο σκοπός της μεθόδου, να σταματάει την αναζήτηση εφ'όσον πρέπει να τερματίσει. Εαν επιθυμούσαμε να υποστηρίζονται και αντίγραφα, η συνθήκη θα έπρεπε να είναι διαφορετική, πχ. εδώ να επιστρέφει πάντοτε `false`.

Ο χρήστης της βιβλιοθήκης πρέπει να έχει επιβεβαιώσει ότι οι τρεις αυτές μέθοδοι λειτουργούν σωστά στο σειριακό δέντρο προτού προσπαθήσει να παραλληλοποιήσει το δέντρο αφού η μη σωστή λειτουργία τους μπορεί να επηρεάσει την ορθότητα του αλγορίθμου επιβεβαίωσης και να οδηγήσει σε λάθη ταυτοχρονισμού. Επιπλέον σημείωση ότι όλες οι μέθοδοι της βιβλιοθήκης βρίσκονται στο namespace TSX το οποίο έχει παραλειφθεί για χάρην χώρου (ισοδύναμο με χρήση using namespace TSX).

```
class AVLNode {
    friend class AVLTree<ValueType>;
    private:
        int key;
        ValueType value;
        AVLNode* children[2];
        int height;
    public:
        AVLNode(int key, ValueType val, AVLNode* left_child, AVLNode* right_child)
            children[0] = left_child;
            children[1] = right_child;
        }

        // to satisfy search tree interface
        bool hasKey(int key_requested) {
            return key == key_requested;
        }

        int nextChild(int desired_key) const {
            return desired_key < key ? 0 : 1;
        }

        bool traversalDone(int desired_key) const {
            return key == desired_key;
        }

        int nextChild(const AVLNode* target) const {
            return target->key < key ? 0 : 1;
        }

        // and the basic required methods
         static constexpr int maxChildren() {
            return 2;
        }

        AVLNode** getChildPointer(int i) {
            assert(i >= 0 && i < 2);
            return &children[i];
        }

        AVLNode* getChild(int i) {
            assert(i >= 0 && i < 2);
            return children[i];
        }
```



```
        void setChild(int i, AVLNode* node) {
            assert(i >= 0 && i < 2);
            children[i] = node;
        }
```

2.7.4 Το δέντρο - AVLTree

Τα δέντρο μας αρκεί να έχει μία ρίζα, ένα αντικείμενο SpinLock της βιβλιοθήκης και τα αντικείμενα των στατιστικών ανά νήμα. Τα αντικείμενα στατιστικών παρέχουν χρήσιμες πληροφορίες για τις αποτυχημένες συναλλαγές και μπορούν να χρησιμοποιηθούν για την διάγνωση προβλημάτων απόδοσης και για την ρύθμιση του αριθμού προσπαθειών. Ορίζουμε και τη συντόμευση `TreeNode`αντί για `AVLNode<ValueType>`για περιβαλλοντικούς λόγους, μιας και η παρούσα εργασία θα τυπωθεί.

```
template <class ValueType>
class AVLTree {
    private:
        AVLNode<ValueType>* root;

        // tree global lock
        TSX::SpinLock &_lock;

        // transactional retries
        const int trans_retries = 30;

        // transactional stats objects
        constexpr int THREAD_AMOUNT_MAX = 100;
        TSX::TSXStats stats[THREAD_AMOUNT_MAX];

        // alias for AVLNode
        using TreeNode = AVLNode<ValueType>;
}
```

2.7.5 Αναζήτηση

Η αναζήτηση για κάποιον κόμβο στο δέντρο μας παρέχεται αυτόματα απο τη βιβλιοθήκη μέσω της μεθόδου `find`:

```
TreeNode lookup(int desired_key) {
    return find<TreeNode>(root,desired_key);
}
```

Η μέθοδος `find` θα επιστρέψει είτε τον κόμβο με το επιθυμητό κλειδί είτε `nullptr` εφόσον αυτός δεν υπάρχει. Εσωτερικά θα χρησιμοποιήσει τις παρεχόμενες μεθόδους του κόμβου. Επιπλέον, η αναζήτηση αυτή θα γίνει χωρίς κλειδώματα και θα έχει εγγυημένη πρόοδο αφού συμβαίνει πανομοιότυπα με μία σειριακή διάσχιση.

2.7.6 Εισαγωγή - Εισαγωγή Κόμβου

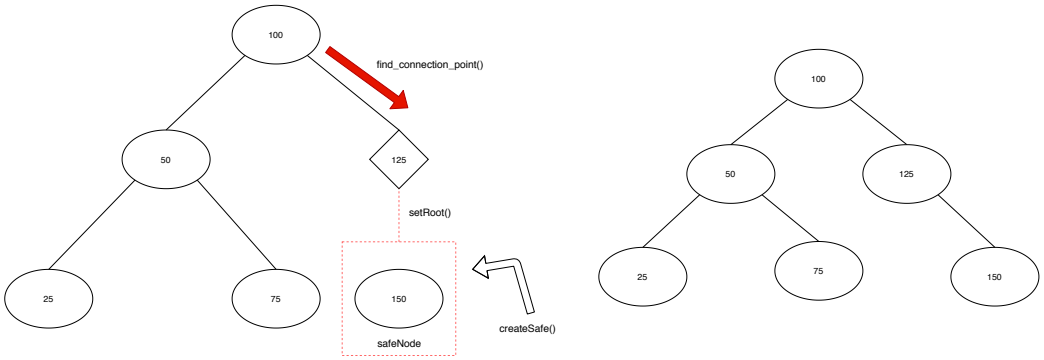
Πρώτο βήμα για την υλοποίηση της μεθόδου εισαγωγής είναι η χρήση του σκελετού. Προσοχή στο ότι παίρνουμε σαν παράμετρο τον αύξοντα αριθμό του νήματος ώστε να γνωρίζουμε ποιο αντικείμενο στατιστικών του αντιστοιχεί.
Απαιτείται να ορίσουμε μία μεταβλητή η οποία θα περιέχει αρχικά τον μέγιστο αριθμό των δοκιμών με HTM συναλλαγή που θέλουμε να αποπειραθούν πριν χρησιμοποιηθεί το κλείδωμα.

```
bool insert(const int k, ValueType val, int t_id) {
    // needs to be mutable
    int retries = trans_retries;

    TM_SAFE_OPERATION_START {

        TSX::Transaction t(retries,_lock, stats[t_id]);

        /* FIND PHASE */

        auto conn_point_snapshot = find_conn_point<TreeNode>(k,&root);

        if (conn_point_snapshot.found()) {
            return false;
        }
            /* FIND PHASE END */

        ConnPoint<TreeNode> conn(conn_point_snapshot, t);

        ...
        //insertion code goes here
        ...
    } TM_SAFE_OPERATION_END

    return true;
}
```

Η μέθοδος μας έχει μέχρι στιγμής, είτε εντοπίσει κόμβο με κλειδί k, επομένως επιστρέφει *false*, είτε το σημείο σύνδεσης, δηλ. τον κόμβο στον οποίο πρέπει να εισαχθεί ο νέος κόμβος ως παιδί και έχουμε αρχικοποιήσει το ΔTA με το αντικείμενο *ConnPoint*. Η αλλαγή που πρέπει να περιγραφεί για αρχή από το ΔTA είναι απλά η προσάρτηση ενός νέου κόμβου ως παιδί του σημείου σύνδεσης. Αυτό συμβαίνει απλά θέτοντας ως ρίζα του δέντρου των αντιγράφων το νέο κόμβο, αφού τον τυλίξουμε σε ένα ασφαλή κόμβο!

```
...
ConnPoint<TreeNode> conn(conn_point_snapshot, t);

// wrap newly inserted node
auto node_to_be_inserted = conn.create_safe(
                            new AVLNode<ValueType>(k,val,nullptr,nullptr)
                            );
// insert it
conn.setRoot(node_to_be_inserted);

...
//balancing code
...

} TM_SAFE_OPERATION_END
```

Άμα κατασκευάζαμε ένα μη εξισορροπημένο δυαδικό δέντρο αναζήτησης εδώ θα ολοκληρωνόταν ο κώδικας. Η βιβλιοθήκη θα φροντίσει η εισαγωγή να γίνει ατομικά. Ωστόσο το δέντρο μπορεί να έχει χάσει την ιδιότητα της ισορροπίας του εξαιτίας της εισαγωγής.



Σχήμα 2.6: Εισαγωγή Κόμβου

2.7.7 Εισαγωγή - Εξισορρόπηση Σειριακού Δέντρου

Στην προηγούμενη ενότητα φτάσαμε στο σημείο όπου έχουμε ένα σημείο σύνδεσης και ένα δέντρο αντιγράφων που περιέχει ένα καινούριο κόμβο στη ρίζα του.
Προτού χρησιμοποιήσουμε την βιβλιοθήκη για να υλοποιήσουμε την εξισορρόπηση, ας δούμε πως συμβαίνει σειριακά.

Υποθέτουμε πως έχουμε κρατήσει το μονοπάτι που ακολουθήσαμε από τη ρίζα προς τον γονέα στον οποίο εισάγουμε το νέο παιδί σε μία στοίβα `insertStack`. Πρέπει για κάθε επίπεδο του δέντρου, άμα απαιτείται να κάνουμε τις απαραίτητες *περιστροφές* και να το συνδέσουμε στον κόμβο του αμέσως ανώτερου επιπέδου. Ελέγχουμε ταυτόχρονα τη συνθήκη που μας υποδεικνύει ότι το δέντρο δεν αλλάζει πια, ώστε να μην χρειαστεί να ανέβουμε ως τη ρίζα, αν το δέντρο έχει εξισορροπηθεί σε χαμηλότερο επίπεδο.

```
auto rotationHappened = false;

AVLNode *prev, *n;
unsigned int heightOld = 0;

for (n = insertStack.pop() , prev = new_node; n; prev = n, n = insertStack.pop()) {
    //insert new node or rebalanced node
    if (rotationHappened) {
        if (k < n->key) {
            n->setL(prev);
            rotationHappened = false;
        } else {
            n->setR(prev);
            rotationHappened = false;
        }
    }

    n = rebalance_ins(n,k,rotation_happened);

    if (height_old == n->height && !rotation_happened) {
        break;
    }
}
```

2.7.7.1 Η Σειριακή Μέθοδος Εξισορρόπησης

Η μέθοδος `rebalance_insert` ελέγχει που απαιτείται κάποια περιστροφή και το εφαρμόζει επιστρέφοντας τη νέα ρίζα του επιπέδου στο οποίο καλέστηκε. Ακολουθεί μία τυπική υλοποίηση αυτής της συνάρτησης για το AVL δέντρο.

```
SafeNode<NodeType> rebalance_insert(SafeNode<NodeType> node) {
    heightOld = n->height;

    // calculate node height
    n->height = maxheight(n->getChild(0), n->getChild(1)) + 1;

    // calculate node's balance
    auto balance = nodeBalance(n);

    rotation_happened = true;

    if (balance > 1 && k < n->getChild(0)->key) { // right rotate
        n = rightRotate(n);
    } else if (balance < -1 && k > n->getChild(1)->key) { //left rotate
        n = leftRotate(n);
    } else if (balance > 1 && k > n->getChild(0)->key) { // left right rotate
        n->setL(leftRotate(n->getChild(0)));
        n->height = maxheight(n->getChild(0), n->getChild(1)) + 1;
        n = rightRotate(n);
    } else if (balance < -1 && k < n->getChild(1)->key) { // right Left Rotate
        n->setR(rightRotate(n->getChild(1)));
        n->height = maxheight(n->getChild(0), n->getChild(1)) + 1;
        n = leftRotate(n);
    } else {
        rotation_happened = false;
    }

    n->height = maxheight(n->getChild(0), n->getChild(1)) + 1;
}
```

Σε αυτό το σημείο πρέπει να παραθέσουμε και την υλοποίηση μίας από τις περιστροφές για να περιγράψουμε ακολούθως την μετατροπή της στην παράλληλη έκδοση. Παρουσιάζουμε την δεξιά περιστροφή.

```
static AVLNode* rightRotate(AVLNode *z) {
        // left becomes root
        auto newRoot = z->getChild(0);

        // left's right will be moved to old root's left
        auto T3 = newRoot->getChild(1);

        // connect old root right of new root
        newRoot->setR(z);

        // connect left's right
        z->setL(T3);

        z->height = maxheight(z->getChild(0), z->getChild(1)) + 1;

        newRoot->height = maxheight((newRoot->getChild(0)), newRoot->getChild(1)) +

        return newRoot;
    };
```

2.7.8 Εισαγωγή - Συνάρτηση Εξισορρόπησης Παράλληλου Δέντρου

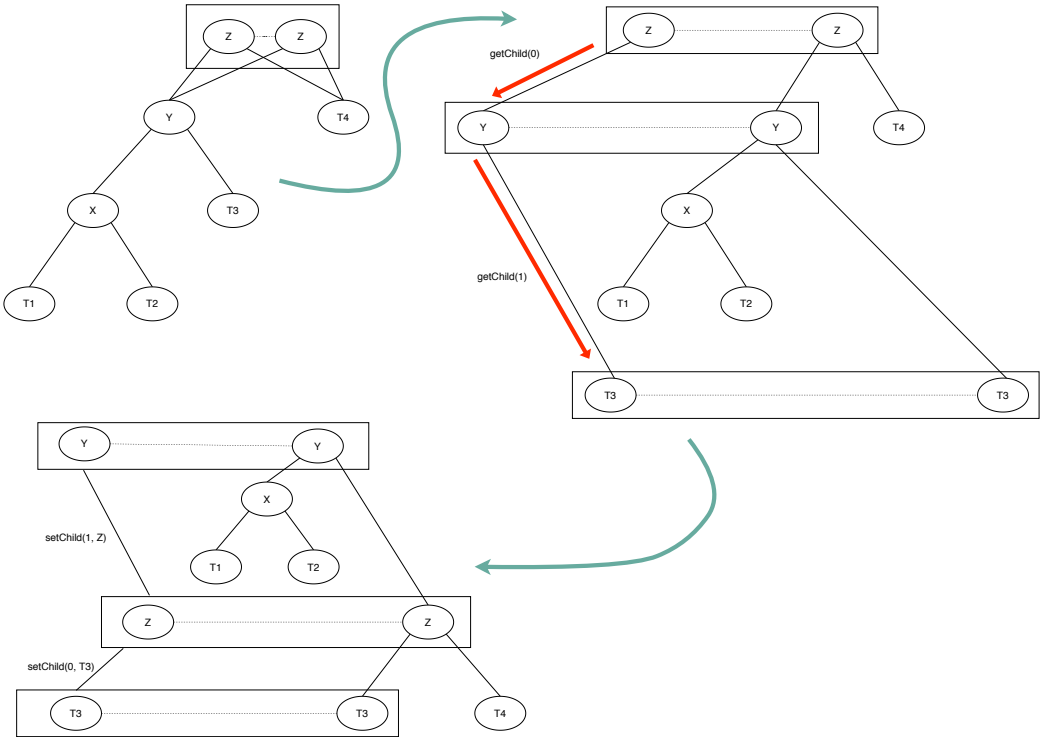
Τι αλλαγές πρέπει να γίνουν για να μπορεί αυτή η εξισορρόπηση να γίνει ως μέρος της παράλληλης πράξης εισαγωγής; Αρχικά ακολουθούμε τον κανόνα:

- Για αλλαγές συνδέσεων των κόμβων, χρησιμοποιούμε ασφαλείς κόμβους
- Για αλλαγές δεδομένων των κόμβων, χρησιμοποιούμε ασφαλείς αναφορές

Εδώ θα δουλέψουμε ανάποδα. Ας μετατρέψουμε με βάση τον κανόνα την δεξιά περιστροφή!

```
static SafeNode<AVLNodeType>* right_rotate(SafeNode<AVLNodeType>* z) {
    /*  T1, T2, T3 and T4 are subtrees.
        z                                      y
        / \                                   /   \
    y   T4      Right Rotate (z)          x      z
    /   \          - - - - - - - - ->    /  \    /  \
    x    T3                             T1  T2  T3  T4
    / \
    T1   T2 */

    // left becomes root
    auto newRoot = z->getChild(0);  // use the SafeNode to get children

    // left's right will be moved to old root's left
    auto T2 = newRoot->getChild(1);

    // connect old root right of new root
    newRoot->setChild(1,z);

    // connect left's right
    z->setChild(0,T2);

    // copy to change
    auto z_safe = z->rwRef();   // for values get a reference
    auto newRoot_safe = newRoot->rwRef();

    // set height with references
    z_safe->height = max_height(z_safe->getChild(0), z_safe->->getChild(1)) + 1;

    // max height returns the max height of the two trees and zero for empty trees
    newRoot_safe->height =
        max_height(newRoot_safe->getChild(0),
            newRoot_safe->getChild(1)) + 1;

    return newRoot;
}
```

Ο κώδικας όπως βλέπουμε μοιάζει σε πολύ μεγάλο βαθμό. Οι αλλαγές που έχουν γίνει είναι ότι η συνάρτηση χρησιμοποιεί και παράγει ασφαλείς κόμβους, επομένως επιδρά μόνο στο *ΔΤΑ* (άρα οι αλλαγές της είναι thread-safe). Χρησιμοποιούμε τις μεθόδους `getChild` και `setChild` για τις αλλαγές συνδέσεων (κρατώντας τον κώδικα ίδιο με το σειριακό) αλλά χρησιμοποιούμε ασφαλείς αναφορές για να αλλάξουμε την τιμή του ύψους των κόμβων. Ο



Σχήμα 2.7: Δεξιά Περιστροφή Βιβλιοθήκης

χρήστης δεν χρειάζεται να ασχοληθεί με το πως μεταβάλλεται το *ΔΤΑ* γιατί αυτό θα επεκταθεί και θα αλλάξει αυτόματα. Αυτό είναι και η επιτυχία της βιβλιοθήκης στην ευκολία χρήσης.

Ακριβώς την ίδια μετατροπή κάνουμε και στην μέθοδο εξισορρόπησης ώστε να δρα πάνω στο *ΔΤΑ*.

```
// apply rebalancing for an insertion operation
SafeNode<TreeNode>* rebalance_ins(SafeNode<TreeNode>* n,
    int k, bool& rotation_happened) {
     // reads and writes are safe
    auto n_data = n->rwRef();
    // calculate node height
    n_data->height =
        max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
    // calculate node's balance
    auto balance = node_balance(n_data);
    rotation_happened = true;
    // for tree modifications (change children etc) use SafeNode
    if (balance > 1 && k < n_data->getChild(0)->key) {
        // right rotate
        n = right_rotate(n);
    } else if (balance < -1 && k > n_data->getChild(1)->key) {
        //left rotate
        n = left_rotate(n);
    } else if (balance > 1 && k > n_data->getChild(0)->key) {
        // left right rotate
        n->setChild(0 ,left_rotate(n->getChild(0)));
        n_data->height =
            max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
        n = right_rotate(n);
    } else if (balance < -1 && k < n_data->getChild(1)->key) {
        // right Left Rotate
        n->setChild(1,right_rotate(n->getChild(1)));
        n_data->height =
            max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
        n = left_rotate(n);
    } else {
        rotation_happened = false;
    }
    // calculate new height
    n_data->height =
        max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
    return n;
}
```

2.7.9 Εισαγωγή - Εξισορρόπηση Παράλληλου Δέντρου

Έχοντας λοιπόν την συνάρτηση που εξισορροπεί κάθε επίπεδο του δέντρου των αντιγράφων, αρκεί να της παρέχουμε τα κατάλληλα επίπεδα και τελικά να φροντίσουμε ώστε αυτά να συνδεθούν κατάλληλα. Εδώ απλοποιεί τον κώδικα αρκετά το γεγονός ότι η βιβλιοθήκη διατηρεί μία στοίβα μονοπατιού. Ουσιαστικά η εξισορρόπηση που περιγράψαμε προηγουμένως ισοδυναμεί με τα εξής απλά βήματα:

1. Θέσε το σημείο σύνδεσης στο αμέσως ανώτερο επίπεδο, κάνοντας το τωρινό σημείο σύνδεσης τη νέα ρίζα του δέντρου των αντιγράφων
2. Εξισορρόπησε το ΔTA, υπολογίζοντας τη νέα ρίζα του
3. Θέσε τη νέα ρίζα ως ρίζα του δέντρου των αντιγράφων

Με το πρώτο βήμα εισάγουμε τους ανώτερους κόμβους στο ΔTA ώστε να μπορούμε να εφαρμόσουμε εξισορρόπηση. Όμως σε ποιο παιδί του νέου σημείου σύνδεσης θα συνδεθεί πια το ΔTA με την ολοκλήρωση της εισαγωγής; Κατά τη διάρκεια της διάσχισης, στη στοίβα του μονοπατιού αποθηκεύεται ο κάθε κόμβος του μονοπατιού μαζί με τον αύξοντα αριθμό του επόμενου παιδιού του μονοπατιού. Έτσι,αυτόματα, το ΔTA θα συνδεθεί σε αυτή τη θέση. Στο συγκεκριμένο δέντρο, αυτός ο αριθμός ορίζεται από τη μεθοδο `next_child` του κόμβου άρα δεν απαιτείται και ο αρχικός έλεγχος. Απλά συνδέουμε τη ρίζα. Τι θα συνέβαινε εάν δεν θέλαμε να συνδεθεί εκεί το ΔTA; Θα ανεβαίναμε ένα επίπεδο υψηλότερα, προσθέτοντας το σημείο σύνδεσης στο ΔTA και θα χρησιμοποιούσαμε την `setChild` ώστε να θέσουμε το σωστό παιδί.

```
// go up path and rebalance
for (SafeNode<TreeNode>* n = conn.popPath(); n != nullptr; n = conn.popPath()) {
    auto n_data = n->rwRef();
    int height_old = n_data->height;

    n = rebalance_ins(n, k , rotation_happened);
    // apply rebalancing to all
    // required nodes
    // rotation returns the new root to place
    // below the connection point

    conn.setRoot(n);     // change the root of the
                         // tree of copies to make the change visible
    if (height_old == n_data->height && !rotation_happened) {
        break;
    }

}
```

Ο παράλληλος κώδικας είναι **σχεδόν πανομοιότυπος** με τον σειριακό, κάνοντας την μετατροπή πανεύκολη. Αρκεί **η επέκταση του δέντρου των ασφαλών κόμβων σε κάθε κόμβο που θέλουμε να έχουμε πρόσβαση** και **η χρήση ασφαλών κόμβων και αναφορών για την περιγραφή των αλλαγών**.

Εδώ μάλιστα, μονάχα η **αλλαγή των τύπων των κόμβων σε ασφαλείς κόμβους**, αντιμετώπισε τη μετατροπή περίπλοκων μεθόδων όπως οι περιστροφές.

Επιπλέον, ο σχεδιασμός της βιβλιοθήκης για δέντρα αναζήτησης σημαίνει ότι η βιβλιοθήκη μπορεί να **αξιοποιήσει την παρεχόμενη διάσχιση για να διευκολύνει τον προγραμματιστή**, π.χ. αρκεί να θέσουμε τη νέα ρίζα του ΔΤΑ με τη μέθοδο `setRoot` και αυτόματα συνδέεται στο σωστό σημείο, εξαιτίας του ότι γνωρίζουμε το σημείο σύνδεσης και ποιο παιδί του πρέπει να αλλάξει.

2.8 Αποσφαλμάτωση (Debugging)

Η διαδικασία της αποσφαλμάτωσης είναι βασική για την κατασκευή μίας παράλληλης δομής δεδομένων αφού πρέπει να επιβεβαιώνεται η σωστή και αποδοτική λειτουργία της δομής και να αντιμετωπίζονται τυχόν λάθη λογικής. Ας δούμε πως αυτό επιτυγχάνεται μέσω της βιβλιοθήκης.

2.8.1 Πλεονεκτήματα Βιβλιοθήκης

1. **Μέθοδοι του χρήστη** Ο χρήστης γράφει αποκλειστικά σειριακές μεθόδους, άσχετα της χρήσης τους από τη βιβλιοθήκη. Εφόσον λειτουργούν σειριακά, λειτουργούν και στον παράλληλο κώδικα, άρα μεγάλο κομμάτι της μεθόδου αποτελεί σειριακή αποσφαλμάτωση

2. **Αλλαγές σε τοπικό αντίγραφο** Όλος ο κώδικας που γράφει ο χρήστης προκαλεί αλλαγές σε ιδιωτικά για κάθε νήμα αντίγραφα του υποδέντρου που αλλάζει. Αυτό σημαίνει ότι ο χρήστης δύναται να εκτυπώσει τις αλλαγές που γίνονται σε κάποιο από αυτά τα αντίγραφα και την κατάσταση του αντιγράφου γενικά, χωρίς το φόβο ότι κάποιο άλλο νήμα προκαλεί αλλαγές ταυτόχρονα σε αυτό. Αρκεί να χρησιμοποιήσει ασφαλείς αναφορές.

3. **Κεντρικό σημείο λαθών συγχρονισμού** Προφανώς δύναται να υπάρχουν λάθη στην υλοποιήση της βιβλιοθήκης που οδηγούν σε λάθος υλοποιήσεις παράλληλων δομών. Ο εντοπισμός τέτοιων λαθών θα χρησιμοποιείται για την διόρθωση της βιβλιοθήκης, άρα κάθε υλοποίηση δομής θα βελτιώνει ή θα επιβαιώνει την ορθότητα όλων των υπολοίπων υλοποιήσεων

2.8.2 Συχνά Λάθη

Σε αυτή την ενότητα θα αναφέρουμε μερικά συχνά λάθη υλοποιήσεων δομών με τη χρήση της βιβλιοθήκης και θα προτείνουμε λύσεις.

Πρόσβαση σε παιδιά κόμβου, χωρίς τη χρήση ασφαλούς κόμβου Ο μοναδικός τρόπος να διατηρηθεί η ιδιότητα του thread-safety είναι να επιβεβαιώνεται ότι δεν έχουν αλλάξει τα παιδιά κάθε κόμβου που συμμετέχει στο ΔΤΑ από την αρχή μέχρι το τέλος της συναλλαγής. Αυτό επιτυγχάνεται μονάχα με το τύλιγμα με ασφαλείς κόμβους. Κάθε πρόσβαση σε παιδιά πρέπει να γίνεται αποκλειστικά μέσω ασφαλών κόμβων και των μεθόδων τους.

Αλλαγή δεδομένων κόμβου χωρίς ασφαλή αναφορά Χωρίς τη χρήση ασφαλούς αναφοράς, η αλλαγή δεδομένων χρησιμοποιώντας τον αρχικό κόμβο οδηγεί σε απροσδιόριστη συμπεριφορά

Λάθη συγχρονισμού από σειριακές μεθόδους Πέραν των δυο παραπάνω λόγων, ειδικά στις υλοποιήσεις των δέντρων αναζήτησης, είναι αναγκαίο οι σειριακές μέθοδοι που παρέχει ο χρήστης να είναι απολύτως ορθές. Εφ'όσον δεν ισχύει αυτό, μπορεί να αποτύχει και η παραλληλοποίηση του δέντρου (πχ. μία λάθος υλοποιήση της μεθόδου `nextChild` μπορεί να εμποδίσει τη σωστή επιβεβαίωση του δέντρου)

Χρήση μεθόδων `getChild,setChild|` Οι μέθοδοι των ασφαλών κόμβων ονομάστηκαν με τον ίδιο τρόπο στους ασφαλείς κόμβους για την εύκολη μετατροπή κώδικα. Προτείνεται η αλλαγή των τύπων των μεταβλητών σε ασφαλείς κόμβους για τις ασφαλείς μεθόδους ώστε να μην χρησιμοποιείται από λάθος η σειριακή έκδοση των αρχικών κόμβων.

Κεφάλαιο 3

Τεχνική Ανάλυση Βιβλιοθήκης

3.1 Γενική Περιγραφή - Απαιτούμενες Μέθοδοι

Η βιβλιοθήκη βασίζεται σε πολύ μεγάλο βαθμό στην υλοποίηση του χρήστη ώστε να είναι όσο το δυνατόν πιο γενικού σκοπού. Χρησιμοποιεί C++ templates ώστε όλα τα αντικείμενα που παρέχει να δέχονται ως παράμετρο ένα αντικείμενο κόμβου του χρήστη και παράγει *μονομορφικό* κώδικα αξιοποιώντας το.

Έτσι ολες οι κλάσεις μπορούν να κάνουν inline τις μεθόδους του αντικειμένου κόμβου και για αυτό το λόγο, η έξυπνη χρήση της βιβλιοθήκης μπορεί να θεωρηθεί zero-cost abstraction συγκριτικά με την χειροκίνητη εφαρμογή της τεχνικής RCU-HTM.

Απαιτείται η παροχή των μεθόδων που περιγράφηκαν στο προηγούμενο κεφάλαιο ώστε η βιβλιοθήκη να δύναται να διαχειριστεί τις συνδέσεις των αρχικών κόμβων για τη σύνδεση των αντιγράφων, για την πρόσβαση στα παιδιά των κόμβων και για τον ορισμό του τρόπου διάσχισης του δέντρου στα δέντρα αναζήτησης.

3.2 Συγχρονισμός

3.2.1 SpinLock

Για την βιβλιοθήκη αυτή αναπτύχθηκε ένα SpinLock από την αρχή, ώστε να έχουμε πρόσβαση στην εσωτερική του κατάσταση εντός HTM συναλλαγής, μιας και οι υλοποιήσεις της standard library αποκρύπτουν αυτήν την πληροφορία αφού δεν μπορεί να προσπελαστεί με thread-safe τρόπο. Η υλοποίηση βασίζεται στις ατομικές μεταβλητές της C++.

```
    private:
        enum {
            UNLOCKED = false,
            LOCKED = true
        };
        std::atomic<bool> spin_lock_;
    public:
        SpinLock(): spin_lock_(false) {}
        void lock() noexcept{
            for (;;) {
                // test
                while (spin_lock_.load(std::memory_order_relaxed) == LOCKED) _mm_pause();
                if (!spin_lock_.exchange(LOCKED, std::memory_order_acquire)) {
                    break;
                }
            }
        }
        void unlock() noexcept{
            spin_lock_.store(false, std::memory_order_release);
        }
        bool isLocked() noexcept {
            return spin_lock_.load(std::memory_order_relaxed);
        }
};
```

Η υλοποίηση είναι σύνηθης, με την ατομική εντολή `exchange` να εξασφαλίζει το ατομικό κλείδωμα. Αξιοποιήθηκαν οι τεχνικές TTAS(Test and Test and Set) για την αποφυγή αχρείαστου polling με ατομικές εντολές και η intrinsic εντολή `__mm_pause()` η οποία ελαχιστοποιεί το αχρείαστο spinning προσθέτοντας μικρές καθυστερήσεις. Η τελευταία βελτιστοποίηση αν και φαίνεται πως θα είχε το ανάποδο αποτέλεσμα, βελτιώνει την απόδοση αφήνοντας χρόνο στην θέση μνήμης να αλλάξει κατάσταση μέχρι τον επόμενο έλεγχο της μέσω polling.

3.2.2 TSXGuard

Το βασικό τμήμα της βιβλιοθήκης που κάνει δυνατή την thread-safe εκτέλεση του κώδικα της βιβλιοθήκης, ονομάστηκε *TSXGuard*.
Πρόκειται για ένα αντικείμενο το οποίο με την αρχικοποίηση του μέσω του constructor του ξεκινάει μία συναλλαγή hardware transactional memory, ενώ διαχειρίζεται και τα aborts αλλά και τον τερματισμό της συναλλαγής.
Η πιο απλή του έκδοση που παρέχεται σαν υλοποίηση ειναι το αντικείμενο `TSXGuardWithStats`.Το αντικείμενο αυτό αρχικοποιείται εντός ενός loop που ελέγχει μία boolean μεταβλητή.
Το αντικείμενο αρχικοποιείται ως εξής:

```
TSXGuardWithStats(const int max_tx_retries,
            SpinLock &mutex,
            unsigned char &err_status,
            TSXStats &stats,
            bool disabled = false,
            RETRY_STRATEGY strat = STUBBORN)
```

όπου:

- `max_tx_retries`, ο μέγιστος αριθμός aborts προτού χρησιμοποιηθεί το κλείδωμα αντί για HTM συναλλαγή,
- `mutex`,αναφορά σε ένα κλείδωμα
- `&err_status`, αναφορά στην συνθήκη του loop,ώστε να τερματίσει αν επιτύχει η συναλλαγή
- `&stats`, αναφορά σε ένα αντικείμενο στατιστικών,
- `disabled`, flag που ακυρώνει το μηχανισμό
- `strat`, στρατηγική χρήσης των retries όπου με την επιλογή STUBBORN κάθε abort κοστίζει ένα retry ενώ με την επιλογή HALF, κάθε abort υποδιπλασιάζει τα retries

Εσωτερικά, ο constructor του αντικειμένου ξεκινάει μία HTM συναλλαγή ή κλειδώνει το SpinLock, αν έχουν εξαντληθεί τα retries. Μετά το τέλος του constructor, κάθε εντολή που ακολουθεί εκτελείται λοιπόν είτε εντός συναλλαγής είτε εντός κλειδώματος, άρα ατομικά.
Πότε ολοκληρώνεται η συναλλαγή; Όταν το αντικείμενο βρεθεί εκτός scope και κληθεί ο destructor του, που είτε θα τερματίσει τη συναλλαγή ή θα ελευθερώσει το κλείδωμα. Τι συμβαίνει σε περίπτωση abort; Η ροή του προγράμματος επιστρέφει εντός του constructor, όπου το abort καταγράφεται στα στατιστικά, μειώνεται ο αριθμός των retries και επαναλαμβάνεται η διαδικασία.

3.2.2.1 Παράδειγμα Χρήσης

```
TSXStats stats;

bool transaction_complete = false;

const int retries = 20;

SpinLock lock;

while (!transaction_complete) {
    TSXGuardWithStats guard(retries, lock, transaction_complete);

    // any code in this block will run atomically

}
```

Με την χρήση αυτού του αντικειμένου, λοιπόν, μπορούμε να δημιουργήσουμε μία περιοχή mutual exclusion στον κώδικα, η εκτέλεση της οποίας γίνεται με thread-safe τρόπο.

3.2.3 TSXTransOnlyGuard

Για την υλοποίηση της βιβλιοθήκης ωστόσο χρησιμοποιείται το ελαφρώς διαφορετικό αντικείμενο `TSXTransOnlyGuard`. Το οποίο έχει ίδια χρήση και λειτουργικότητα, αλλά δεν διαχειρίζεται το κλείδωμα. Η διαχείριση του κλειδώματος, δίνεται στο αντικείμενο `Transaction`το οποίο είναι μέρος του API της βιβλιοθήκης μας, ώστε ο χρήστης να μπορεί να παρέχει το δικό του αντικείμενο κλειδώματος. Το αντικείμενο αυτό αντίθετα χρησιμοποιείται εσωτερικά.

3.2.4 Transaction

Το αντικείμενο `Transaction` έχει την απλούστατη αρμοδιότητα να κλειδώνει το `SpinLock`όταν εξαντληθούν τα retries και να το ξεκλειδώνει στον destructor του, εφ'όσον κλειδώθηκε.

3.2.5 Το Block Συναλλαγής

Το block συναλλαγής ορίζεται μέσω των δύο macro `TM_SAFE_OPERATION_START` και `TM_SAFE_OPERATION_END`. Το macro `TM_SAFE_OPERATION_START` ουσιαστικά κρύβει την ύπαρξη ενός loop το οποίο υλοποιεί την λογική των retries. Το macro επεκτείνεται στην εξής μορφή:

```
__internal__thread_transaction_success_flag__ = false;
while(!__internal__thread_transaction_success_flag__)
```

Όπως φαίνεται υπάρχει μία εσωτερική μεταβλητή-συνθήκη η οποία παίζει τον ρόλο του `err_status` για το αντικείμενο `TSXTransOnlyGuard` και η αρχή ενός loop που ελέγχει αυτή τη συνθήκη. Γι' αυτόν τον λόγο πρέπει πάντοτε να ακολουθείται από ένα block με άγκιστρα.
Το macro `TM_SAFE_OPERATION_END` υπάρχει για αισθητικούς λόγους και δεν επεκτείνεται ως κάτι στην σύνηθη λειτουργία, ωστόσο επιτρέπει την υλοποίηση ενός έξτρα τρόπου λειτουργίας της βιβλιοθήκης, την λειτουργία *EARLY ABORT*.

3.3 Αναζήτηση

3.3.1 Το αντικείμενο ConnPointData

Το αντικείμενο ConnPointData είναι το αποτέλεσμα μίας αναζήτησης για το σημείο σύνδεσης μίας πράξης και απαιτείται για την αρχικοποίηση του ΔΤΑ (επόμενη ενότητα).

Πέρα από το ίδιο το σημείο σύνδεσης(ως διευθ/ση) απαιτούνται:

1. Το μονοπάτι που ακολουθήθηκε ως το σημείο σύνδεσης
2. Η διεύθυνση της διεύθυνσης της ρίζας του αρχικού δέντρου (ώστε να μπορεί να αλλάξει και η ίδια η ρίζα)
3. Η θέση(αύξοντας αριθμός) και η διευθ/ση του παιδιού που θα αλλάξει
4. Η πληροφορία του αν βρέθηκε το παιδί που θα αλλάξει

Το αντικείμενο ConnPointData περιέχει όλες αυτές τις πληροφορίες και παράγεται αυτόματα με έναν από τους τρόπους που θα παρουσιαστούν στη συνέχεια.

3.3.2 Αυτόματες Συναρτήσεις Αναζήτησης

Όπως αναφέρθηκε, όταν η βιβλιοθήκη χρησιμοποιείται για τη δημιουργία ενός δέντρου αναζήτησης, προσφέρεται αυτόματα η συνάρτηση `find_conn_point`. Η συνάρτηση δέχεται ως template όρισμα τον τύπο του αντικειμένου κόμβου και ως ορίσματα το κλειδί αναζήτησης και τη διεύθυνση της διεύθυνσης της ρίζας του αρχικού δέντρου.

Ας αναλύσουμε την εσωτερική της υλοποίηση:

Αρχικά, διαβάζει τη ρίζα από τον δείκτη που δόθηκε ως όρισμα. Αυτό το κάνει μία φορά, γιατί μπορεί να αλλάξει κατά τη διάρκεια της εκτέλεσης της μεθόδου.

```
// find_conn_point: traverse tree with given root
// and return a ConnPointData object for the
// node with the given key. The node will be determined
// by the traversalDone method
//and the path taken by the nextChild method.

template <class NodeType>
ConnPointData<NodeType> find_conn_point(int key, NodeType** root) {
    // everything we need to create a conn point
    ConnPointData<NodeType> result;

    result.root_of_structure = root;

    NodeType* orig_head = *root;
    // first step, get snapshot of head
    //(can change during runtime, use only copy)

    // will keep the previous node
    // which will be the connection point

    ...
}
```

Στη συνέχεια αξιοποιεί τις μεθόδους `traversalDone` και `nextChild` του κόμβου για να βρει το σημείο σύνδεσης αλλά και να αποθηκεύσει το μονοπάτι ως αυτό.

```
...
NodeType* curr = orig_head;

// search for node with key
for (; curr && !curr->traversalDone(key);) {

    auto next_child = curr->nextChild(key);

    // search for node with key, adding path to stack and keeping the prev
    result.path.push(curr, next_child);

    curr = curr->getChild(next_child);
};
...
```

Τέλος αξιοποιεί το μονοπάτι για να θέσει όλες τις πληροφορίες που απαιτεί το αντικείμενο ConnPointData.

```
    int prev_next_child_index = INSERT_POSITIONS::AT_ROOT;

    NodeType* prev_s = nullptr;

    if (!result.path.Empty()) {
        auto stack_top =  result.path.pop();
        prev_s = stack_top.node;
        prev_next_child_index = stack_top.next_child;
    }


    // was a node found
    result.found_ = curr && curr->hasKey(key);

    // set prev as the connection point
    // will be null if operation happens at head
    result.connection_point_ = prev_s;

    // the child to which the pointer to change corresponds
    // should be AT_ROOT if there's no connection_point
    result.con_ptr.child_index = prev_next_child_index;
```

```
    // the original value of the pointer to change
    result.con_ptr.snapshot = result.connection_point_ ? curr : orig_head;

    return result;
}
```

3.3.2.1 Συναρτήσεις Αναζήτησης

Αντίστοιχα χωρίς την παραγωγή του αντικειμένου ConnPointData και του μονοπατιού παρέχονται οι συναρτήσεις:

- `NodeType* find<NodeType>(NodeType* root, int desired_key)`, που πραγματοποιεί την αναζήτηση για δεδομένο κλειδί
- `bool node_exists(NodeType* target_node, NodeType* root)` που επιστρέφει αν ο δοσμένος κόμβος ανήκει στο δέντρο

3.3.3 Το αντικείμενο PathTracker

Για την αναζήτηση σε οποιαδήποτε δενδρική δομή παρέχεται το αντικείμενο PathTracker. Το αντικείμενο PathTracker ορίζεται με βάση το αντικείμενο κόμβου (template παράμετρος) και είναι ουσιαστικά μία μηχανή καταστάσεων που παριστάνει τη διάσχιση ενός δέντρου.

3.3.3.1 Αρχικοποίηση

Το αντικείμενο αρχικοποιείται με παράμετρο την διεύθυνση της διεύθυνσης του δέντρου. Απαιτείται αυτή η έμμεση αναφορά για την δημιουργία του αντικειμένου ConnPointData.

3.3.3.2 Δομή

Το αντικείμενο παρακολουθεί 4 παραμέτρους:

1. Την ρίζα
2. Τον κόμβο στον οποίο δείχνει το αντικείμενο, *την τρέχουσα θέση του*
3. Το επίπεδο του δέντρου, δηλ. σε τι ύψος βρίσκεται η θέση του
4. Το μονοπάτι, που αποθηκεύει το μονοπάτι από την ρίζα προς την τρέχουσα θέση

3.3.3.3 Τρέχουσα θέση - Σημείο Σύνδεσης

Η τρέχουσα θέση χρησιμοποιείται ως το σημείο στο οποίο θα τοποθετηθεί το δέντρο των αντιγράφων. Χρησιμοποιώντας το μονοπάτι από τη ρίζα προς τη θέση, μπορεί να οριστεί ως σημείο σύνδεσης ο γονέας της τρέχουσας θέσης και ως παιδί που θα αλλάξει η τρέχουσα θέση.

3.3.3.4 Μετακίνηση

Κατά τον ορισμό του αντικειμένου, η τρέχουσα θέση είναι ο κόμβος της ρίζας του αρχικού δέντρου. Υποστηρίζονται κινήσεις προς δύο κατευθύνσεις:

1. Προς την κατεύθυνση των παιδιών με την μέθοδο `moveToChild(int pos)`, που μετακινεί την τρέχουσα θέση στο παιδί της με αριθμό `pos`

```
NodeType* moveToChild(int pos) {
    ++at_level_;

    path_.push(current_pos_, pos);
    current_pos_ = current_pos_->getChild(pos);

    return current_pos_;
}
```

   Η μετακίνηση αυξάνει το επίπεδο κατά 1, καταγράφει το μονοπάτι και επιστρέφει τη νέα θέση.

2. Προς την κατεύθυνση της ρίζας με την μέθοδο `moveUp(int n_levels = 1)`, που δέχεται ως προαιρετική παράμετρο πόσα επίπεδα προς τη ρίζα θα είναι η μετακίνηση.

```
void moveUp(int n_levels = 1) {
    at_level_ = at_level_ >= n_levels ? at_level_ - n_levels: -1;

    for (int i = 0; at_level_ >= 0 && i < n_levels - 1; i++) {
        path_.pop();
    }

    if (!path_.Empty()) {
        current_pos_ =  path_.pop().node;
    }
}
```

   Η μετακίνηση μειώνει το επίπεδο κατά `n_levels` με το τερματικό επίπεδο (στη ρίζα) να υποδηλώνεται με -1.

Ορισμός Σημείου Σύνδεσης Τελικός σκοπός του PathTracker είναι ο ορισμός του σημείου σύνδεσης. Για αυτό χρησιμοποιείται η μέθοδος `connectHere()`. Όταν κληθεί, αφαιρεί τον προηγούμενο κόμβο της τρέχουσας θέσης από το μονοπάτι, θέτοντας τον ως σημείο σύνδεσης και επιστρέφει το κατάλληλο αντικείμενο ConnPointData, κατασκευάζοντας το όμοια με την αυτόματη μέθοδο `find_conn_point`.

```
ConnPointData<NodeType> connectHere() {
    // create a ConnPointData object
    ConnPointData<NodeType> conn_point_snapshot;

    // conn point is at root level
    bool at_root = at_level_ == -1;

    // get prev node from path
    NodeAndNextPointer<NodeType> conn_point_and_next_child = path_.pop();

    // set it
    conn_point_snapshot.connection_point_ = at_root ? nullptr: conn_point_and_next_child.n

    // record path in ConnPointData
    path_.copy_to(conn_point_snapshot.path);


    // fill ConnPointData
    conn_point_snapshot.root_of_structure = root_;

    conn_point_snapshot.con_ptr.child_index = at_root?
            INSERT_POSITIONS::AT_ROOT :
                conn_point_and_next_child.next_child;

    conn_point_snapshot.con_ptr.snapshot = current_pos_;

    // restore path
    path_.push(
        conn_point_and_next_child.node,
        conn_point_and_next_child.next_child
    );

    // return ConnPointData
    return conn_point_snapshot;
}
```

3.3.4 PathTracker Thread-Safety

Το αντικείμενο PathTracker είναι κατασκευασμένο ώστε να χρησιμοποιείται εκτός συναλλαγής ώστε να πραγματοποιείται η αναζήτηση του σημείου σύνδεσης. Η αναζήτηση αυτή *εγγυάται πως θα επιστρέψει σωστά δεδομένα στη βιβλιοθήκη για την εκτέλεση μιας thread-safe πράξης*. **Δεν εγγυάται ωστόσο ότι οι κόμβοι που συναντήθηκαν στη διαδρομή αποτελούν αποδεκτή κατάσταση του δέντρου**.

Η τελευταία πρόταση αφορά πχ. περιπτώσεις όπου ο χρήστης θα έμπαινε σε πειρασμό να χρησιμοποιήσει το αντικείμενο για σύνθετες διασχίσεις, όπως μία in-order διάσχιση. Το αντικείμενο PathTracker δε θα εγγυόταν ότι η in-order διάσχιση θα ήταν αποδεκτή αλλά μονάχα ότι *το μονοπάτι από τη ρίζα μέχρι κάθε κόμβο της διάσχισης ήταν αποδεκτό τη στιγμή της επίσκεψης του κάθε κόμβου*.

3.4 Το δέντρο των αντιγράφων

3.4.1 Χτίζοντας την ιδέα

Το δέντρο των αντιγράφων προέκυψε σαν έμπνευση από την έννοια των Monads στις συναρτησιακές γλώσσες. Θέλουμε να περιγράψουμε μία διαδικασία με παρενέργειες, την αλλαγή ενός οποιουδήποτε δέντρου, θέλουμε ο κώδικας να μοιάζει με σειριακό κώδικα, αλλά ταυτόχρονα θέλουμε ο ίδιος ο κώδικας να μην έχει παρενέργειες ώστε να μπορεί να εκτελεστεί παράλληλα από πολλά νήματα. Φυσικά, στο τέλος μίας πράξης πάνω στο δέντρο, θέλουμε αυτή η αλλαγή να φαίνεται στο δέντρο.

3.4.1.1 Υπόθεση

Ας υποθέσουμε αρχικά ότι η πράξη του δέντρου μας αλλάζει ένα υποσύνολο του δέντρου, ένα υποδέντρο που έχει ως γονέα κάποιο κόμβο του δέντρου ή είναι η ρίζα και περιέχει κάποιους κόμβους. Οι κόμβοι αυτού του υποδέντρου θα έχουν παιδιά που θα ανήκουν σε αυτό αλλά και παιδιά που ανήκουν στο αρχικό δέντρο, αλλά δεν θα αλλάξουν από την πράξη. Έστω ότι αντιγράφαμε μονάχα αυτό το υποσύνολο, παράγοντας ένα νέο δέντρο και κάνοντας τις απαραίτητες αλλαγές, αλλά ταυτόχρονα "θυμόμαστε" ποιοι ήταν οι αρχικοί κόμβοι που το παρήγαγαν.

3.4.1.2 Ατομική Σύνδεση

Όπως δείχτηκε και στην τεχνική RCU-HTM, αν επιβεβαιώσουμε ατομικά πως:

1. τα αρχικά παιδιά των αλλαγμένων κόμβων δεν έχουν αλλάξει κατά τη διάρκεια της πράξης
2. το υποδέντρο των αρχικών κόμβων είναι ακόμα συνδεδεμένο στο αρχικό δέντρο

τότε αν ανταλλάξουμε ατομικά το υποδέντρο που παρήγαγε το αντίγραφο με το αντίγραφο, διατηρείται η συνοχή του δέντρου.

3.4.1.3 Το αντίγραφο

Αρχικά προκύπτει η ανάγκη να παράξουμε ένα αντίγραφο ενός υποδέντρου του αρχικού δέντρου αλλά όπου συμβαίνουν αλλαγές. Το αντίγραφο θέλουμε να περιέχει:

1. τα παιδιά των αρχικών κόμβων τη στιγμή που αντιγράφηκαν, ώστε να ελεγχθεί το (1)
2. αναφορές στους αρχικούς κόμβους, ώστε να ελεγχθεί το (1)
3. αναφορές στους νέους κόμβους - αντίγραφα
4. μία αναφορά στη ρίζα του αρχικού υποδέντρου, ώστε να ελεγχθεί το (2)
5. τις αλλαγές στις συνδέσεις αλλά και στις τιμές των νέων κόμβων

Δεδομένου ότι η δομή αυτή θα περιγράφει τις αλλαγές ενός δέντρου, μία πολύ λογική επιλογή είναι να είναι και αυτή ένα δέντρο. Κάθε νήμα θα παράγει και θα διαχειρίζεται ξεχωριστά μία τέτοια δομή ώστε να προκαλέσει αλλαγές στο δέντρο.

3.5 Ασφαλής Κόμβος

Έτσι φτάνουμε στους κόμβους του ΔΤΑ, τους *ασφαλείς κόμβους*. Ο σκοπός των κόμβων αυτών είναι να περιέχουν ό,τι αναφέραμε στην προηγούμενη ενότητα αλλά ταυτόχρονα να περιγράφουν αλλαγές σε οποιαδήποτε δεντρική δομή. Δημιουργούνται με βάση έναν κόμβο του αρχικού δέντρου.
Οι ασφαλείς κόμβοι περιέχουν λοιπόν τα εξής:

1. αναφορά σε ένα αντικείμενο ConnPoint (τη ρίζα)
2. αναφορά στον αρχικό κόμβο, από τον οποίο δημιουργήθηκαν
3. αναφορά στο αντίγραφο του κόμβου, που μπορεί να έχει αλλαγές
4. ως παιδιά, αναφορές σε άλλους ασφαλείς κόμβους
5. πίνακα ένδειξης ότι κάποιο παιδί του αρχικού κόμβου έχει ήδη προσπελαστεί σε αυτή την πράξη άρα υπάρχει ασφαλής κόμβος που του αντιστοιχεί
6. τα παιδιά των αρχικών κόμβων τη στιγμή που αντιγράφηκαν
7. ένα τύπο που υποδηλώνει αν ο αρχικός κόμβος ανήκε στο αρχικό δέντρο ,είναι νεοδημιούργητος για να συνδεθεί ή γενικά δεν πρέπει να επιβεβαιωθεί

3.5.1 Ορισμός

```
template <class NodeType>
class SafeNode
{
    enum SAFENODE_TYPE {
        ORIG_TREE_NODE,
        ORIG_TREE_NO_VALIDATION,
        NEW_NODE
    };

    private:
        ConnPoint<NodeType>& conn_point_;
        NodeType* const original_;
        NodeType* copy_;
        std::array<SafeNode<NodeType>*,NodeType::maxChildren()> children_;
        std::array<bool,NodeType::maxChildren()> modified_;
        std::array<NodeType*, NodeType::maxChildren()>
        children_pointers_snapshot_;

        bool deleted_; // for deletion

        SAFENODE_TYPE node_type_;
        ...
}
```

Οι ιδιότητες του ασφαλούς κόμβου

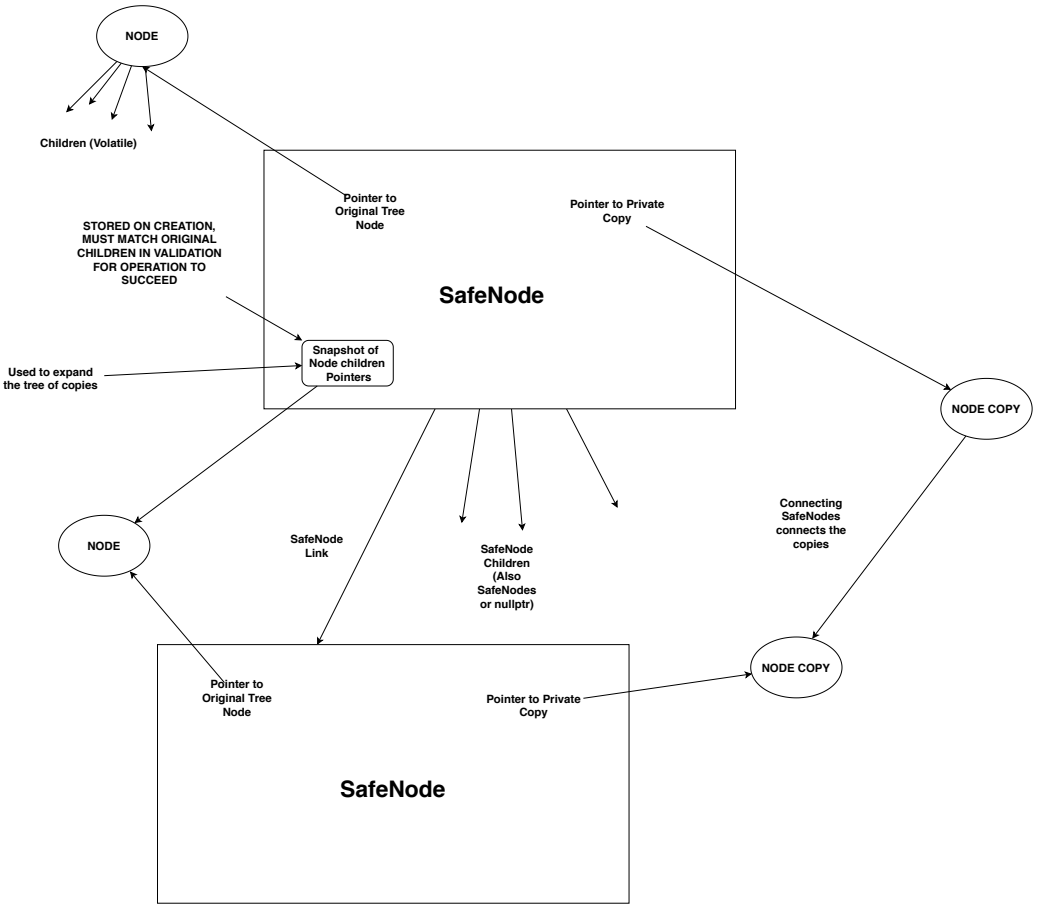
1. Προσδίδουν την αίσθηση στο χρήστη πως αλλάζει τους κόμβους στο αρχικό δέντρο, αλλά στην πραγματικότητα παράγουν το ΔΤΑ που θα συνδεθεί

2. Η πρόσβαση σε παιδιά του μέσω της μεθόδου `getChild` *επεκτείνει το ΔΤΑ*, δημιουργώντας αυτόματα ένα νέο ασφαλή κόμβο από το αντίστοιχο παιδί του αρχικού κόμβου και συνδέοντας τα εσωτερικά αντίγραφα μεταξύ τους

3. Προσφέρουν την *ασφαλή αναφορά* μέσω της μεθόδου `rwRef` , η οποία επιστρέφει δείκτη στο αντίγραφο (το οποίο μπορεί να αλλάξει χωρίς παρενέργειες στο αρχικό δέντρο), προσομοιώνοντας την σειριακή μετάλλαξη των τιμών του κόμβου

4. Προσομοιώνουν την σειριακή αλλαγή παιδιών με τη μέθοδο `setChild` η οποία σκόπιμα λειτουργεί μόνο με ασφαλείς κόμβους και επίσης ενημερώνει τις συνδέσεις των αντιγράφων

5. Το ΔΤΑ είναι **persistent**, δηλαδή οι επεκτάσεις του ΔΤΑ (`getChild` αυτόματα, `setChild` με πρωτοβουλία του χρήστη) αποθηκεύονται ως παιδιά των ασφαλών κόμβων και επόμενες προσβάσεις δε θα ξαναδημιουργήσουν νέους κόμβους και νέα αντίγραφα



Σχήμα 3.1: Υλοποίηση Ασφαλούς Κόμβου

3.5.2 Η μέθοδος getChild

Η περιγραφή της μέθοδου `getChild` είναι ιδανική για την κατανόηση του μηχανισμού αφού παράγει ουσιαστικά το ΔΤΑ.
Δέχεται ως παράμετρο τον αύξοντα αριθμό του παιδιού του ασφαλή κόμβου που θα προσπελαστεί.

```
//getChild: returns either an already linked SafeNode
//or creates a new SafeNode when accessing an unsafe node
SafeNode<NodeType>* getChild(const int child_pos) {
    /* process:
    A SafeNode without set children will have nullptr in
    its array. When getting the child of a SafeNode:
    if:
        1. has been set) return it
        2. has not been set) create SafeNode, assign to children and return it

    */
    ...
}
```

Αρχικά ελέγχει αν ο αριθμός αυτός είναι αποδεκτός και αν ο τύπος του ασφαλούς κόμβου επιτρέπει την ασφαλή προσπέλαση των παιδιών του. Αν έχει απενεργοποιηθεί η επιβεβαίωση, δεν επιτρέπεται η πρόσβαση στα παιδιά του αφού δε θα ήταν thread-safe.

Στη συνέχεια, αν ο κόμβος δεν έχει αντιγραφεί ήδη και ανήκει στο αρχικό δέντρο, αντιγράφεται (πριν τη δημιουργία του αντιγράφου ο δείκτης του αντιγράφου δείχνει στον αρχικό κόμβο). Γιατί ωστόσο να αντιγραφεί ο κόμβος επειδή θέλουμε να προσπελάσουμε το παιδί του; Επειδή οι ασφαλείς κόμβοι όπως είπαμε διαχειρίζονται τις συνδέσεις μόνο των αντιγράφων τους. Επομένως για να είναι συνεκτικό το ΔΤΑ, πρέπει κάθε ασφαλής κόμβος που έχει παιδιά άλλους ασφαλείς κόμβους να αντιγραφεί.

Η χρήση της μέθοδου getChild υποδεικνύει πως θέλουμε να κάνουμε κάποια αλλαγή στους κόμβους που προσπελαύνουμε. Αν απλά μας ενδιαφέρει ποια είναι τα παιδιά ενός ασφαλούς κόμβου ενδείκνυται η χρήση της μεθόδου `peekChild` η οποία δεν δημιουργεί αντίγραφα.

```
 // no oob errors
assert(child_pos >= 0 && child_pos < NodeType::maxChildren());
assert(node_type_ != ORIG_TREE_NO_VALIDATION && "Validation Error: Tried to access

// force parent copy
// getting a child
```

```
// explicitly
// means that you
// seek to modify it
if (copy_ == original_ && node_type_ == ORIG_TREE_NODE) {
    make_copy();
}
...
```

Στη συνέχεια, χρησιμοποιείται ο πίνακας `modified_` ώστε αν ήδη έχει δημιουργηθεί ασφαλής κόμβος από παλαιότερη προσπέλαση, να μην ξαναδημιουργηθεί και αντίθετα να επιστραφεί ο αντίστοιχος αποθηκευμένος κόμβος.

```
...

//child has already been copied
if (modified_[child_pos]) {
    return children_[child_pos];
}

...
```

Στην πρώτη προσπέλαση, θα δημιουργηθεί ένας νέος ασφαλής κόμβος από το αντίστοιχο παιδί του αρχικού κόμβου, θα τεθεί ως παιδί του ασφαλούς κόμβου ο οποίος καλεί τη μέθοδο και θα επιστραφεί.

Προσοχή στο γεγονός ότι ο τύπος του κόμβου που καλέι τη μέθοδο καθορίζει και τον τύπο των παιδιών. Συγκεκριμένα, αν ο ασφαλής κόμβος ανήκε στο αρχικό δέντρο (`ORIG_TREE_NODE`), οι ασφαλείς κόμβοι που θα παράξει η getChild θα έχουν επίσης τον ίδιο τύπο. Αντίθετα αν ο τύπος είναι αυτός του νεοδημιούργητου κόμβου, όλα τα παιδιά του θα θεωρούνται νεοδημιούργητα επίσης.

```
// child has been modified
modified_[child_pos] = true;

SafeNode<NodeType>* new_safe = nullptr;

// if node has been copied read from copy
// else read from original child pointers backup
const auto original_child = copy_->getChild(child_pos);

 //child not copied yet,
//copy and use the copy from now on
if (original_child) {

    new_safe = node_type_ == ORIG_TREE_NODE ?
    conn_point_.wrap_safe(original_child):
        conn_point_.create_safe(original_child);

    children_[child_pos] = new_safe;
    copy_->setChild(child_pos, new_safe->rwRef());
}

return new_safe;
```

3.5.3 Η μέθοδος setChild

Η μέθοδος setChild διατηρεί τη δομή που έχει ήδη περιγραφεί αλλά επιτρέπει την σύνδεση οποιουδήποτε ασφαλούς κόμβου ως παιδί του καλώντος. Μπορεί να χρησιμοποιηθεί για τη σύνδεση ενός νεοδημιούργητου κόμβου, αν συνδυαστεί με τη χρήση της μεθόδου `createSafe` του `ConnPoint`.

3.5.4 Σχέση με ConnPoint

Κάθε ΔΤΑ ορίζεται από ένα μοναδικό ConnPoint και τους ασφαλείς κόμβους του. Έτσι κάθε ασφαλής κόμβος έχει μία μοναδική αναφορά στο ConnPoint που τον δημιούργησε. Με αυτό τον τρόπο, το ConnPoint μπορεί να έχει πρόσβαση στους ασφαλείς κόμβους που το αποτελούν. Το ConnPoint ταυτόχρονα διατηρεί ένα σύνολο επιβεβαίωσης στο οποίο προστίθενται όλοι οι ασφαλείς κόμβοι που περιέχουν κόμβους που πρέπει να επιβεβαιωθούν.

Για αυτό δεν δίνεται η δυνατότητα άμεσης κατασκευής ενός ασφαλούς κόμβου (private constructor) ώστε να εξασφαλίζεται η σχέση, αφού αυτοί κατασκευάζονται είτε μέσω ενός ConnPoint είτε από άλλον ασφαλή κόμβο, "κληρονομώντας" την αναφορά του.

3.5.5 Τύποι Ασφαλούς Κόμβου

Οι τύποι ασφαλούς κόμβου που υπάρχουν είναι:

1. ο κόμβος που ανήκε στο αρχικό δέντρο, υποδεικνύει ότι ο αρχικός κόμβος ανήκε στο αρχικό δέντρο
2. ο νεοδημιούργητος κόμβος, ο οποίος δεν προστίθεται στο σύνολο επιβεβαίωσης
3. κόμβος χωρίς επιβεβαίωση, ο οποίος δεν προστίθεται στο σύνολο επιβεβαίωσης

Ο λόγος ύπαρξης των τύπων είναι η μελλοντική προσθήκη μηχανισμού memory-management όπου είναι σημαντικό να γνωρίζει η βιβλιοθήκη ποιοί κόμβοι μπορούν να απελευθερωθούν στη μνήμη στην αποτυχία μίας συναλλαγής και ποιοι όχι.

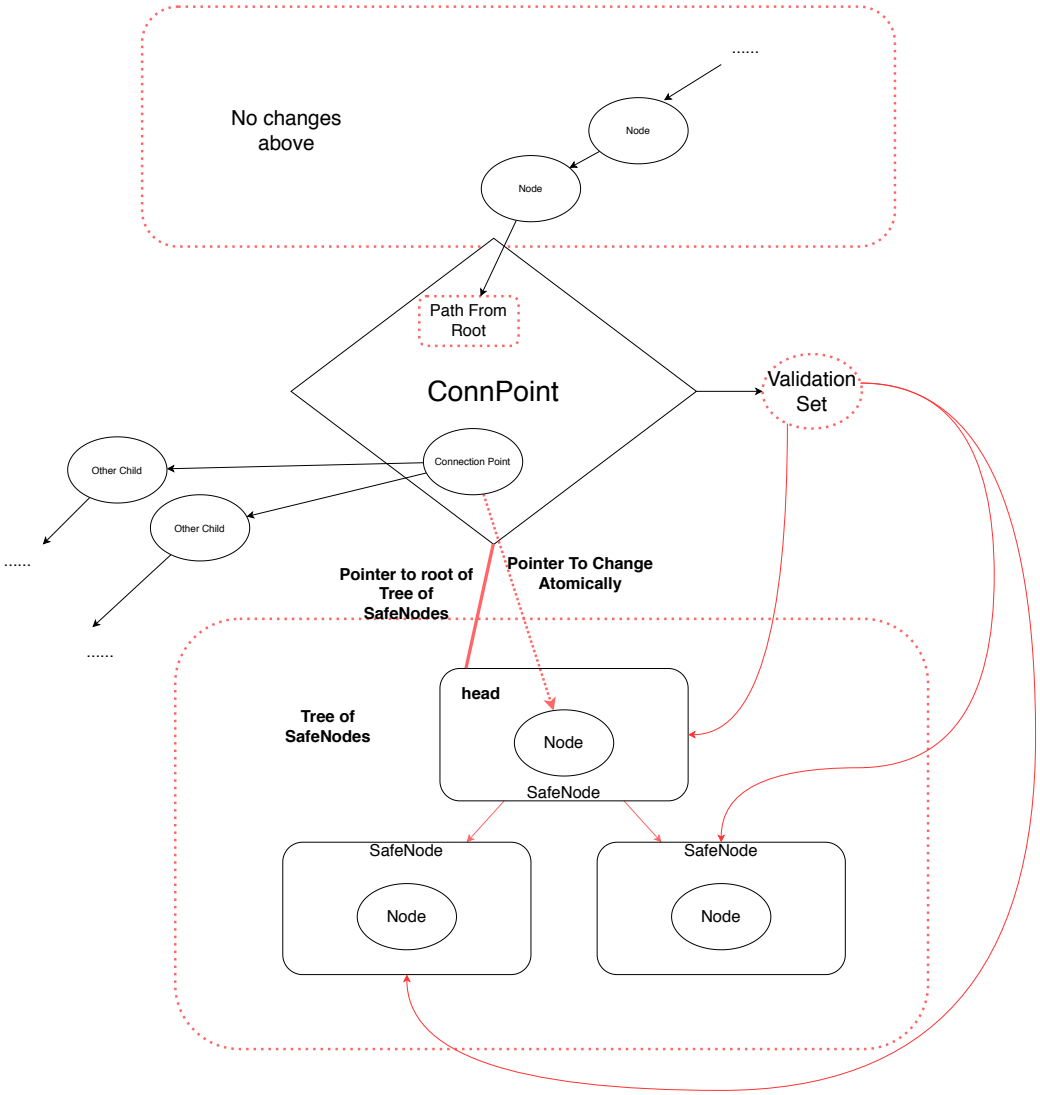
3.6 Σημείο Σύνδεσης (ConnPoint)

3.6.1 Δομή Σημείου Σύνδεσης

Το σημείο σύνδεσης είναι το αντικείμενο το οποίο ενώνει όλα τα κομμάτια της βιβλιοθήκης. Παραθέτεται ο κώδικας των εσωτερικών αναφορών και μεταβλητών του αντικειμένου για λόγους συντομίας αφού περιληπτικά περιλαμβάνει κάθε αναφορά και αντικείμενο που διαχειρίζεται το αντικείμενο ConnPoint.

```
// returns if copy connection was successful
    bool& connect_success_;

    // preallocated pool of SafeNodes
    thread_local static PreAllocVec<SafeNode<NodeType>*, 500> validation_set_;

    // the connection point is the node whose child pointer will
    // be modified to connect the tree of copies
    NodeType* connection_point_;

    // if null, the insertion should be done at
    // the root of the data structure

    // the point of insertion of the new tree
    NodeType** connection_pointer_;

    // the root of the data structure to modify
    NodeType** root_;

    // original connection pointer
    // should not have changed during transaction
    NodeType* conn_pointer_snapshot_;


    // if a node is popped from path,
    // to be set as the new connection point,
    // the old head will be its child at index
    // child_to_exchange_
    int child_to_exchange_;

    // path to connection point
    Stack& path_to_conn_point_;
```

```
// tree of copies was successfully connected
bool copy_connected_;


// The global lock
TSX::SpinLock& _lock;

// tsx stats
TSX::TSXStats &stats_;


// the root node of the tree of copies
SafeNode<NodeType>* head_;

// determines if any modification
// was done to the tree
bool tree_was_modified_;

// won't run the operation
// using transaction if lock is already locked
bool already_locked_;

// transactional retries before
// acquiring fallback lock
int& trans_retries_;

// set to true if operation
// throws a validation error
bool validation_aborted_;
```



Σχήμα 3.2: Υλοποίηση ConnPoint

3.6.2 Λειτουργίες

Το αντικείμενο ConnPoint είναι το βασικότερο αντικείμενο της βιβλιοθήκης διότι επιτελέι σχεδόν όλες τις απαραίτητες λειτουργίες της τεχνικής HTM, κρύβοντας τις ταυτόχρονα από τον χρήστη.

Δημιουργείται από ένα αντικείμενο ConnPointData, αποτέλεσμα της αναζήτησης του σημείου σύνδεσης.

Το σημείο σύνδεσης εκτελεί τις εξής λειτουργίες:

1. Δρα ως ρίζα των ΔΤΑ, παράγοντας τον πρώτο ασφαλή κόμβο (ρίζα)
2. Διατηρεί εσωτερικά το σημείο σύνδεσης (κόμβος αρχικού δέντρου) και σε ποιο παιδί του θα συνδεθεί το ΔΤΑ
3. Συνδέει το δέντρο των αντιγράφων στο αρχικό δέντρο, αν επιτύχει η συναλλαγή
4. Κατασκευάζει ασφαλείς κόμβους από κόμβους του αρχικού δέντρου ή από καινούριους κόμβους
5. Διαχειρίζεται την λογική της επιβεβαίωσης, έχοντας πρόσβαση σε όλους τους ασφαλείς κόμβους που το αποτελούν
6. Μετακινεί προς τη ρίζα το σημείο σύνδεσης, αν χρειάζεται

3.6.3 Ρίζα του ΔΤΑ

Το αντικείμενο ConnPoint εσωτερικά διατηρεί μία ξεχωριστή δεντρική δομή που αποτελείται από ασφαλείς κόμβους. Η δομή αυτή διατηρείται με τη χρήση ενός δείκτη στη ρίζα της. Κατά τη δημιουργία του αντικειμένου, η δομή είναι κενή, άρα ο δείκτης είναι `nullptr`.

Η κλήση της μεθόδου `getRoot` θα λειτουργήσει όπως η `getChild` του ασφαλούς κόμβου και θα τυλίξει το παιδί στη θέση στην οποία θα συνδεθεί το ΔΤΑ με ένα ασφαλή κόμβο, ξεκινώντας το ΔΤΑ. Αντίστοιχα η μέθοδος `setRoot` θέτει αυτή τη ρίζα σε έναν ασφαλή κόμβο της επιλογής του χρήστη.

3.6.4 Σύνδεση με Αρχικό Δέντρο

Έχοντας ως δεδομένο τον κόμβο στον οποίο πρέπει να συνδεθεί το ΔΤΑ και ως ποιο παιδί του (ή αντίστοιχα ότι πρέπει να συνδεθεί στη ρίζα), πρέπει να αναλυθεί το τι σημαίνει πως θα συνδεθεί το ΔΤΑ στο αρχικό δέντρο.

Αρχικά ας ξεκινήσουμε από το γεγονός ότι το ΔΤΑ αποτελείται από ασφαλείς κόμβους, ενώ το αρχικό δέντρο από κόμβους του χρήστη. Στην σύνδεση, το σημείο σύνδεσης (κόμβος αρχικού δέντρου ή η ρίζα), θα συνδεθεί με το αντίγραφο που βρίσκεται εσωτερικά στον ασφαλή κόμβο της ρίζας του ΔΤΑ ή με το `nullptr`.

Οι ασφαλείς κόμβοι εξασφαλίζουν ότι τα αντίγραφα που περιέχουν δημιουργούν ένα ορθώς συνδεδεμένο δέντρο που αντικατοπτρίζει τις αλλαγές που περιέγραψε ο χρήστης, άρα αρκεί η σύνδεση της ρίζας αυτού του δέντρου ως παιδί του σημείου σύνδεσης.

Η σύνδεση αυτή γίνεται εντός συναλλαγής HTM, μετά από την επιβεβαίωση της συνοχής του αρχικού δέντρου, άρα είναι και ατομική.

3.6.5 Κατασκευή ασφαλών κόμβων

Το αντικείμενο ConnPoint είναι το μοναδικό αντικείμενο με την αρμοδιότητα δημιουργίας ασφαλών κόμβων. Εντός των μεθόδων του `wrap_safe`,`create_safe` και `wrap_no_validate`, καλεί τον κατασκευαστή του ασφαλούς κόμβου προντίζοντας να δώσει την αυτόαναφορα του (`this`) ως όρισμα, ώστε να εξασφαλίζεται η 1-1 σχέση μεταξύ ασφαλών κόμβων και ConnPoint.

```
SafeNode<NodeType>* wrap_safe(NodeType* some_node) {
    if (!some_node) {
        return nullptr;
    }

    return new SafeNode<NodeType>(*this, some_node);
}
```

3.6.6 Επιβεβαίωση

Γενικά Η επιβεβαίωση του ΔΤΑ είναι η διαδικασία που εξασφαλίζει ότι το ΔΤΑ μπορεί να συνδεθεί χωρίς να καταστραφεί η συνοχή του αρχικού δέντρου. Συμβαίνει εντός συναλλαγής HTM (ώστε και η ίδια να συμβαίνει ατομικά) και αποτελείται από τα εξής βήματα:

1. Επιβεβαίωση ότι το σημείο σύνδεσης παραμένει συνδεδεμένο στο αρχικό δέντρο

2. Επιβεβαίωση ότι δεν έχει αλλάξει η διεύθυνση του παιδιού που θα συνδεθεί το ΔΤΑ και τα παιδιά κανενός από τους κόμβους που ανήκουν στο ΔΤΑ (ή γενικά έχουν τυλιχτεί σε ασφαλή κόμβο κατά τη διάρκεια της πράξης)

Ατομικότητα Το γεγονός ότι όλοι οι έλεγχοι γίνονται εντός συναλλαγής HTM εξασφαλίζει ότι οποιαδήποτε αλλαγή στους δείκτες που αναγνώστηκαν από άλλο νήμα θα αναγκάσει τη συναλλαγή να αποτύχει. Επομένως γίνονται ατομικά.

Το Σημείο Σύνδεσης παραμένει συνδεδεμένο στο δέντρο Ο έλεγχος αυτός γίνεται με δύο διαφορετικούς τρόπους ανάλογα αν πρόκειται για δέντρο αναζήτησης ή γενική δενδρική δομή.

- Για δέντρα αναζήτησης, αξιοποιείται η παραγόμενη συνάρτηση `find_target_node`. Η συνάρτηση αυτή αναζητά το σημείο σύνδεσης, διασχίζοντας το δέντρο ξεκινώντας από τη ρίζα αξιοποιώντας τη μέθοδο `next_child(NodeType* target_node)` των κόμβων για την επιλογή της πορείας. Ο τρόπος αυτός ενδείκνυται δίοτι αρκεί το σημείο σύνδεσης απλά να βρίσκεται στο δέντρο, ακόμα και με αλλαγμένο μονοπάτι προς αυτό και η πράξη θα επιτύχει.

- Για γενικές δενδρικές δομές, επιβεβαιώνεται η ύπαρξη του μονοπατιού που διασχίστηκε ώστε να βρεθεί αρχικά το σημείο σύνδεσης, το οποίο μονοπάτι διατηρείται εντός του αντικειμένου ConnPoint.

Έλεγχος δεικτών παιδιών Το αντικείμενο ConnPoint διατηρεί αντίγραφο της διεύθυνσης του παιδιού του σημείου σύνδεσης ώστε να επιβεβαιώσει ότι δεν έχει αλλάξει. Για την επιβεβαίωση των κόμβων του ΔΤΑ χρησιμοποιείται το σύνολο επιβεβαίωσης.

Σύνολο επιβεβαίωσης Κάθε ασφαλής κόμβος, την στιγμή της δημιουργίας του, δημιουργεί ένα "αρχείο" των διευθύνσεων των παιδιών του κόμβου που περιέχει. Έχει λοιπόν τη δυνατότητα, αφού έχει αναφορά στον αρχικό κόμβο και σε αυτό το αρχείο, να ελέγξει αν άλλαξαν τα παιδιά του.
Το σύνολο επιβεβαίωσης είναι ένας πίνακας που περιέχει αναφορές σε όλους τους ασφαλείς κόμβους που οφείλουν να επιβεβαιωθούν.
Κατά τη δημιουργία του, ο κάθε ασφαλής κόμβος τοποθετεί τον εαυτό του

(αν είναι τύπου NODE_FROM_ORIG_TREE) στο σύνολο επιβεβαίωσης του αντικειμένου ConnPoint στο οποίο αντιστοιχεί.

```
bool validate_copy() {
   if (!connection_point_) {
       // insertion at root

       // has root copy changed ?
       if (conn_pointer_snapshot_ != *root_) {
           TSX::TSXGuard::abort<VALIDATION_FAILED>();
           return false;
       }
   } else {
       // has the connection pointer to be modified
       // changed?
       if (conn_pointer_snapshot_ != connection_point_->getChild(child_to_exchange
           TSX::TSXGuard::abort<VALIDATION_FAILED>();
           return false;
       }

       // is the root of the tree of copies still connected?
       if (!connPointConnected()) {
           TSX::TSXGuard::abort<VALIDATION_FAILED>();
           return false;
       }
   }

   return validate_children(...);

   }
```

```
bool validate_children() {
    // have the children pointers of the original nodes changed?
    for (auto i = 0; i < validation_set_.size(); i++) {
            auto safe_node = validation_set_.get(i);
            // skip for new nodes
            if (safe_node->node_type_ == SafeNode<NodeType>::ORIG_TREE_NODE) {
                // the original node and its current children
                const auto original_node = safe_node->original_;

                // the saved children which shouldn't have changed
                const auto &saved_children = safe_node->children_pointers_snapshot_;

                for (int i = 0; i < NodeType::maxChildren(); i++) {
                    if (saved_children[i] != original_node->getChild(i)) {
                        TSX::TSXGuard::abort<VALIDATION_FAILED>();
                        return false;
            }
        }
    }
}
```

Αποτυχία επιβεβαίωσης Η επιβεβαίωση αποτυγχάνει σε δύο περιπτώσεις:

1. Αποτυγχάνει κάποιος από τους ελέγχους, οπότε η βιβλιοθήκη εκπέμπει ένα `abort` με αποτέλεσμα την επανεκκίνηση της συναλλαγής ή και ολόκληρης της πράξης
2. Έχουμε κάποια ταυτόχρονη πρόσβαση από νήμα που ακυρώνει τη συναλλαγή
3. Η συναλλαγή ακυρώθηκε για κάποιον άσχετο λόγο πχ. page fault

3.6.7 Μετακίνηση σημείου σύνδεσης

Έχουν γίνει πολλές αναφορές στην χρησιμότητα της μετακίνησης του σημείου σύνδεσης προς τη ρίζα. Εδώ θα αναφερθεί μονάχα ο μηχανισμός με τον οποίο αυτό επιτυγχάνεται.

Δεδομένου ενός σημείου σύνδεσης και του αποθηκευμένου μονοπατιού προς αυτό, για να μετακινηθεί το σημείο σύνδεσης του ΔΤΑ στο γονέα του αρκεί το παλιό σημείο σύνδεσης να γίνει η ρίζα του ΔΤΑ.

Ας δουμε αναλυτικά τη μέθοδο που αναλαμβάνει αυτή τη λειτουργία:

Αρχικά αφαιρείται ο προηγούμενος κόμβος του μονοπατιού και ετοιμάζεται να τεθεί ως νέο σημείο σύνδεσης. Βλέπουμε ότι έχουμε και την πληροφορία του σε ποιο παιδί αντιστοιχεί το παλιό σημείο σύνδεσης.

```
SafeNode<NodeType>* pop_path() {

    // is at root
    // return nullptr
    if (!connection_point_) {
        return nullptr;
    }


    NodeType* new_conn_point = nullptr;
    int new_conn_point_next_index = -1;

    if (!path_to_conn_point_.Empty()) {

        // pop node
        auto parent_of_conn_point = path_to_conn_point_.pop();

        new_conn_point = parent_of_conn_point.node;

        new_conn_point_next_index = parent_of_conn_point.next_child;

    }
...
```

Στη συνέχεια, η διευθ/ση του παλιού σημείου σύνδεσης (connpoint_) αποθηκεύεται ως η διεύθυνση του παιδιού που θα αλλάξει ατομικά και αποθηκεύεται και ο αύξοντας αριθμός του, για την επιβεβαίωση. Τίθεται και το νέο σημείο σύνδεσης.

```
// keep old conn pointer snapshot to force its validation
auto old_conn_pointer_snapshot_ = conn_pointer_snapshot_;

// for the new connection point
// the pointer to change
// should point to the previous
// connection point
conn_pointer_snapshot_ = connection_point_;

// calculate which child should be changed to insert the tree of copies
// by using the previous connection point's key
const int next_child_index = new_conn_point_next_index;


// change connection point to the new conn point
connection_point_ = new_conn_point;

// also update connection pointer
// if not at root
if (connection_point_) {
    connection_pointer_ = connection_point_->getChildPointer(next_child_index);
}
```

Στο τέλος επεκτείνεται προς τη ρίζα του αρχικού δέντρου το ΔΤΑ, τυλίγοντας το παλιό σημείο σύνδεσης,θέτοντας το ως ρίζα του ΔΤΑ και συνδέοντας το ήδη υπάρχον ΔΤΑ ως παιδί της. Ενδιαφέρον κομμάτι της υλοποίησης αποτελεί ο έλεγχος πως ο κόμβος της νέα ρίζας του ΔΤΑ έχει ακόμα ως παιδί του τον κόμβο της παλιάς ρίζας του ΔΤΑ, απαραίτητη προϋπόθεση ώστε να διατηρηθεί η συνοχή του δέντρου. Η βιβλιοθήκη μέσω compilation flag υποστηρίζει δύο τρόπους εφαρμογής αυτού του ελέγχου. Αν έχει τεθεί η σημαία TM_EARLY_ABORT, θα ελεγχθεί εκτός συναλλαγής επιτόπου αν ισχύει η συνθήκη και εφ'όσον αποτύχει ο έλεγχος, επανεκκινείται όλη η πράξη. Εναλλακτικά, η συνθήκη κωδικοποιείται στα αποθηκευμένα παιδιά της νέας ρίζας, ώστε να ελεγχθεί στην επιβεβαίωση.

```
    ....

    // using the previous connection point
    // make a safeNode to turn to new head
    SafeNode<NodeType>* newHead = wrap_safe(conn_pointer_snapshot_);


    // and connect it
    newHead->setChild(child_to_exchange_, head_);


    // the old connection point should necessarily continue
    // pointing to it's old value
    #ifdef TM_EARLY_ABORT
        if (newHead->original->getChild(child_to_exchange_)
                != old_conn_pointer_snapshot_) {
            validation_abort();
            throw ValidationAbortException();
        }
    #endif

    newHead->children_pointers_snapshot_[child_to_exchange_]
                                    = old_conn_pointer_snapshot_;


    // now head can be updated
    head_ = newHead;
    // as well which child to update for the next pop
    child_to_exchange_ = next_child_index;

    return head_;
}
```

3.6.8 Ολοκλήρωση Πράξης

Όταν ολοκληρωθεί ο κώδικας του μπλοκ συναλλαγής, θα κληθεί ο destructor του ConnPoint. Σε αυτό το σημείο θα κληθεί η μέθοδος που αναλαμβάνει την σύνδεση του ΔΤΑ.

```
bool connect_atomically() noexcept {

    if (validation_aborted_) {
        return false;
    }

    unsigned char err_status = 0;

    TSX::TSXTransOnlyGuard guard(
        trans_retries_,
        _lock,
        err_status,
        stats_,
        already_locked_,
        TSX::STUBBORN
    );

    if (err_status != VALIDATION_FAILED && validate_copy()) {
        connect_copy();
        return true;
    }

    return false;
}
```

Όπως φαίνεται στη μέθοδο, δημιουργούμε ένα ατομικό τμήμα κώδικα στο οποίο γίνεται προσπάθεια επιβεβαίωσης και σύνδεσης του ΔΤΑ. Εφόσον αποτύχει εξαιτίας κάποιου λάθους συνοχής που εντόπισε η επιβεβαίωση, όλο το δημιουργημένο δέντρο θεωρείται άχρηστο, άρα επανεκκινείται η πράξη με την **επιστροφή του false**. Αν αποτύχει για κάποιον άλλο λόγο, γίνονται και άλλες προσπάθειες της επιβεβαίωσης, σε περίπτωση που επιτύχει.

Τελικά, συνδέεται το ΔΤΑ και **επιστρέφεται true**.

3.7 Pools

Η χρήση της βιβλιοθήκης φανερώνει την αναγκαιότητα δημιουργίας πολλών αντικειμένων ασφαλών κόμβων (ως κόστος της βιβλιοθήκης) αλλά και κόμβων του αρχικού δέντρου (πχ. σε πράξεις εισαγωγής). Οι κόμβοι αυτοί αποθηκεύονται στο σωρό, κάτι που απαιτεί την κλήση συναρτήσεων διαχείρισης μνήμης που προκαλούν μεγάλη μείωση απόδοσης αφού και έχουν μεγαλό χρονικό κόστος εκτέλεσης, αλλά και επειδή χρησιμοποιούν εσωτερικά μηχανισμούς συγχρονισμού που δεν είναι αναγκαστικά αποδοτικοί, οδηγούν σε δραματική ελλάτωση της δυνατότητας κλιμάκωσης των παράλληλων δομών. Η χρήση ειδικών memory allocators για εφαρμογές πολλών νημάτων βελτιώνει την κατάσταση, αλλά υπάρχει μία πιο αποδοτική λύση.

Ειδικά για τους ασφαλείς κόμβους γνωρίζουμε τα εξής:

- Οι πράξεις στα δέντρα συνήθως αλλάζουν και προσθέτουν μικρό σχετικά αριθμό κόμβων
- Οι ασφαλείς κόμβοι αποσύρονται είτε μία συναλλαγή επιτύχει, είτε αποτύχει

Με αυτά ως δεδομένο παρατηρούμε ότι έχοντας δεσμεύσει μνήμη για ένα μικρό αριθμό ασφαλών κόμβων ξεχωριστά για κάθε νήμα, μπορούμε να επαναχρησιμοποιούμε αυτή τη μνήμη και ουσιαστικά να λειτουργεί ο μηχανισμός της βιβλιοθήκης με μηδενική ανάγκη για δέσμευση μνήμης.

Αντίστοιχα για τους αρχικούς κόμβους, για να απομονώσουμε την απόδοση της βιβλιοθήκης από την επιρροή του εκάστοτε memory allocator, χρησιμοποιούμε την ίδια τεχνική για την δέσμευση μεγάλου χώρου μνήμης στην αρχικοποίηση της βιβλιοθήκης και δημιουργούμε τους κόμβους χρησιμοποιώντας αυτή την μνήμη.

Υλοποίηση Η υλοποίηση βασίζεται στις thread-local μεταβλητές της C++, οι οποίες επιτρέπουν την ύπαρξη ξεχωριστών static μεταβλητών για κάθε νήμα αλλά την διαχείριση τους με κοινό κώδικα.

Για τους απλούς κόμβους,έχουμε δημιουργήσει την κλάση `memory_pool` η οποία μας παρέχει τη δυνατότητα πρόσβασης σε ένα δοσμένο αριθμό από preallocated αντικείμενα ασφαλών κόμβων. Έτσι η κλάση ConnPoint περιέχει ένα thread-local αντικείμενο `memory_pool` (ένα ανα νήμα), το οποίο επιτρέπει την επαναχρησιμοποίηση κόμβων.Ο χρήστης δεν αλληλεπιδρά άμεσα με αυτό το μηχανισμό.

Για τους αρχικούς κόμβους χρησιμοποιείται το αντικείμενο `memory_pool_tracked`, το οποίο λειτουργεί με παρόμοιο τρόπο , με τη διαφορά ότι παρέχει δυνατότητα deallocation της μνήμης που δεσμεύεται. Έτσι η κλάση ConnPoint παρέχει το δικό της memory allocator που υλοποιείται με ένα thread-local αντικείμενο `memory_pool_tracked` (ένα ανα νήμα) και τη μέθοδο `create_new_node`, που δέχεται τα ίδια ορίσματα με την συνάρτηση `new`

για την δημιουργία ενός νέου κόμβου. Πριν τη χρήση τους απαιτείται η κλήση της static μεθόδου `init_node_pool` για την αρχικοποίηση των pools, στον κατασκευαστή του αρχικού δέντρου και της `reset_node_pool` στον destructor για την επαναφορά τους στην αρχική κατάσταση.

Περιορισμοί Στην τωρινή υλοποίηση, δεν έχει υλοποιηθεί μηχανισμός διαχείρισης μνήμης επομένως, δεν απελευθερώνεται ποτέ η μνήμη των αρχικών κόμβων εκτός από όταν χρησιμοποιείται η μέθοδος `reset_node_pool`, η οποία απελευθερώνει κάθε κόμβο που έχει δημιουργηθεί σε ένα δεδομένο δέντρο. Επιπλέον, δεδομένου ότι ο μηχανισμός σχεδιάστηκε κυρίως για την εκτέλεση benchmarks, υποστηρίζει την χρήση των pools με μονάχα ένα δέντρο ανά εκτελέσιμο, δεδομένου ότι τα pools αποτελούν static μεταβλητές των αντικειμένων της βιβλιοθήκης.

3.8 Πρόωρη Ακύρωση Πράξης

Η βιβλιοθήκη υποστηρίζει μία προαιρετική έκδοση όπου αξιοποιούνται οι εξαιρέσεις της C++ για την πρόωρη ακύρωση μίας πράξης, αποφεύγοντας την αχρείαστη εκκίνηση κάποιας συναλλαγής HTM η οποία μπορεί να προκαλέσει συγκρούσεις σε άλλες συναλλαγές.

Συγκεκριμένα, σε αυτή την έκδοση, κατά τη διάρκεια δημιουργίας αντιγράφων των εσωτερικών κόμβων των ασφαλών κόμβων, πραγματοποιείται έλεγχος ότι δεν άλλαξαν τα παιδιά του αρχικού κόμβου από τη στιγμή δημιουργίας του ασφαλούς κόμβου, παρόμοια με την επιβεβαίωση εντός συναλλαγής.

```
for (int i = 0; i < NodeType::maxChildren(); i++) {
    #ifdef TM_EARLY_ABORT_ALL
        if (children_pointers_snapshot_[i] != original_->getChild(i)) {
            conn_point_.validation_abort();
            throw ValidationAbortException();
        }
    #endif

    copy_->setChild(i,children_pointers_snapshot_[i]);
}
```

Την εξαίρεση διαχειρίζεται κρυφά το block της συναλλαγής (εδώ είναι απαραίτητη η ετικέτα τέλους `TM_SAFE_OPERATION_END`, αφού κρύβει αυτή τη διαχείριση) και ουσιαστικά προκαλεί επανεκκίνηση της πράξης.

Κεφάλαιο 4

Αξιολόγηση

4.1 Μέγεθος Κώδικα - Απλότητα

Η πρώτη μετρική που θα αξιολογήσουμε είναι αυτή του μεγέθους κώδικα. Θα χρησιμοποιήσουμε καταχρηστικά τον αριθμό γραμμών κώδικα των υλοποιήσεων ενός AVL δέντρου υλοποιημένου με τη βοήθεια της βιβλιοθήκης εναντίον μίας χειροκίνητης υλοποιήσης της τεχνικής RCU-htTM από τον εισηγητή μου Δημήτρη Σιακαβάρα.

Εξαιτίας του γεγονότος ότι οι υλοποιήσεις είναι γραμμένες σε C++ και C, αντίστοιχα, θα συγκρίνουμε μονάχα τον κώδικα υλοποίησης των μεθόδων εισαγωγής, συμπεριλαμβάνοντας κάθε βοηθητική μέθοδο που υλοποιεί λειτουργίες της τεχνικής RCU-htTM. Θα συγκρίνουμε ταυτόχρονα και την απλότητα του κώδικα από άποψη κατανόησης.

4.1.0.1 Manual RCU-HTM

Ξεκινώντας από την εισαγωγή ενός κόμβου χειροκίνητα, η οποία δεν προσφέρεται για έυκολη κατανόηση:

```
1  int rbt_insert(void *rbt, void *thread_data, int key, void *value)
2  {
3      int ret = 0;
4      ret = _avl_insert_helper(rbt, key, value, thread_data);
5      return ret;
6  }
7
8  static int _avl_insert_helper(avl_t *avl, int key, void *value, tdata_t *tdata)
9  {
10     avl_node_t *nodes_to_free[MAX_htEIGhtT], *nodes_alloced[MAX_htEIGhtT];
11     int ntf_top, nalloced_top;
```

```
12      avl_node_t *node_stack[MAX_htEIGhtT];
13      int stack_top;
14      tm_begin_ret_t status;
15      int retries = -1;
16      int i;
17      avl_node_t *tree_copy_root, *connection_point;
18      int connection_point_stack_index;
19
20  try_from_scratch:
21      ht_reset(tdata->ht);
22
23
24      if (++retries >= TX_NUM_RETRIES) {
25          tdata->lacqs++;
26          pthread_spin_lock(&avl->avl_lock);
27          _traverse_with_stack(avl, key, node_stack, &stack_top);
28          if (stack_top >= 0 && node_stack[stack_top]->key == key) {
29              pthread_spin_unlock(&avl->avl_lock);
30              return 0;
31          }
32          connection_point_stack_index = -1;
33          connection_point = _insert_and_rebalance_with_copy(key, value,
34                                      node_stack, stack_top, tdata, &tree_copy_root,
35                                      &connection_point_stack_index,
36                                      nodes_to_free, &ntf_top,
37                                      nodes_alloced, &nalloced_top);
38          tdata->replaced_nodes += ntf_top + 1;
39          if (!connection_point) {
40              avl->root = tree_copy_root;
41          } else {
42              if (key <= connection_point->key)
43                      connection_point->left = tree_copy_root;
44              else
45                      connection_point->right = tree_copy_root;
46          }
47          pthread_spin_unlock(&avl->avl_lock);
48
49      /* Asynchronized traversal. If key is there we can safely return. */
50      _traverse_with_stack(avl, key, node_stack, &stack_top);
51      if (stack_top >= 0 && node_stack[stack_top]->key == key) {
52          return 0;
53      }
54
```

```
55      // For now let's ignore empty tree case and case with only one node in the tree.
56      assert(stack_top >= 2);
57
58      connection_point_stack_index = -1;
59      connection_point = _insert_and_rebalance_with_copy(key, value,
60                                      node_stack, stack_top, tdata, &tree_copy_root,
61                                      &connection_point_stack_index,
62                                      nodes_to_free, &ntf_top,
63                                      nodes_alloced, &nalloced_top);
64      tdata->replaced_nodes += ntf_top + 1;
65
66      int validation_retries = -1;
67  validate_and_connect_copy:
68
69      if (++validation_retries >= TX_NUM_RETRIES) {
70      goto try_from_scratch;
71      }
72
73      /* Transactional verification. */
74      while (avl->avl_lock != LOCK_FREE)
75          ;
76
77      tdata->tx_starts++;
78      status = TX_BEGIN(0);
79      if (status == TM_BEGIN_SUCCESS) {
80          if (avl->avl_lock != LOCK_FREE)
81              TX_ABORT(ABORT_GL_TAKEN);
82
83          // Validate copy
84          if (key < node_stack[stack_top]->key && node_stack[stack_top]->left != NULL)
85              TX_ABORT(ABORT_VALIDATION_FAILURE);
86          if (key > node_stack[stack_top]->key && node_stack[stack_top]->right != NULL)
87              TX_ABORT(ABORT_VALIDATION_FAILURE);
88          if (avl->root != node_stack[0])
89              TX_ABORT(ABORT_VALIDATION_FAILURE);
90
91          if (connection_point_stack_index <= 0) {
92              for (i=0; i < stack_top; i++) {
93                  if (key <= node_stack[i]->key) {
94                          if (node_stack[i]->left != node_stack[i+1])
95                              TX_ABORT(ABORT_VALIDATION_FAILURE);
96                  } else {
97                      if (node_stack[i]->right!= node_stack[i+1])
```

```
98                          TX_ABORT(ABORT_VALIDATION_FAILURE);
99                  }
100             }
101         } else {
102             avl_node_t *curr = avl->root;
103             while (curr && curr != connection_point)
104                 curr = (key <= curr->key) ? curr->left : curr->right;
105             if (curr != connection_point)
106                 TX_ABORT(ABORT_VALIDATION_FAILURE);
107             for (i=connection_point_stack_index; i < stack_top; i++) {
108                 if (key <= node_stack[i]->key) {
109                     if (node_stack[i]->left != node_stack[i+1])
110                         TX_ABORT(ABORT_VALIDATION_FAILURE);
111                 } else {
112                     if (node_stack[i]->right!= node_stack[i+1])
113                         TX_ABORT(ABORT_VALIDATION_FAILURE);
114                 }
115             }
116         }
117 
118         int j;
119         for (i=0; i < htT_LEN; i++) {
120             for (j=0; j < tdata->ht->bucket_next_index[i]; j+=2) {
121                 avl_node_t **np = tdata->ht->entries[i][j];
122                 avl_node_t  *n  = tdata->ht->entries[i][j+1];
123                 if (*np != n)
124                     TX_ABORT(ABORT_VALIDATION_FAILURE);
125             }
126         }
127 
128 
129         // Now let's 'commit' the tree copy onto the original tree.
130         if (!connection_point) {
131             avl->root = tree_copy_root;
132         } else {
133             if (key <= connection_point->key)
134                 connection_point->left = tree_copy_root;
135             else
136                 connection_point->right = tree_copy_root;
137         }
138 
139         TX_END(0);
140     } else {
```

```
141         tdata->tx_aborts++;
142         if (ABORT_IS_EXPLICIT(status) &&
143             ABORT_CODE(status) == ABORT_VALIDATION_FAILURE) {
144             tdata->tx_aborts_explicit_validation++;
145             goto try_from_scratch;
146         } else {
147             goto validate_and_connect_copy;
148         }
149 }
150
151
152 static inline void _traverse_with_stack(avl_t *avl, int key,
153                                         avl_node_t *node_stack[MAX_htEIGhtT],
154                                         int *stack_top)
155 {
156     avl_node_t *parent, *leaf;
157
158     parent = NULL;
159     leaf = avl->root;
160     *stack_top = -1;
161
162     while (leaf) {
163         node_stack[++(*stack_top)] = leaf;
164
165         int leaf_key = leaf->key;
166         if (leaf_key == key)
167             return;
168
169         parent = leaf;
170         leaf = (key < leaf_key) ? leaf->left : leaf->right;
171     }
172 }
173
174 static avl_node_t *_insert_and_rebalance_with_copy(int key, void *value,
175         avl_node_t *node_stack[MAX_htEIGhtT], int stack_top, tdata_t *tdata,
176         avl_node_t **tree_copy_root_ret, int *connection_point_stack_index,
177         avl_node_t *nodes_to_free[MAX_htEIGhtT], int *ntf_top,
178         avl_node_t *nodes_alloced[MAX_htEIGhtT], int *nalloced_top)
179 {
180     avl_node_t *tree_copy_root, *connection_point;
181
182     *ntf_top = -1;
183     *nalloced_top = -1;
```

```
184
185     //> Start the tree copying with the new node.
186     tree_copy_root = avl_node_new(key, value);
187     nodes_alloced[++(*nalloced_top)] = tree_copy_root;
188     *connection_point_stack_index = stack_top;
189     connection_point = stack_top >= 0 ? node_stack[stack_top--] : NULL;
190
191     //> Empty tree case
192     if (stack_top < 0) {
193         *tree_copy_root_ret = tree_copy_root;
194         return connection_point;
195     }
196
197     while (stack_top >= -1) {
198         //> If we've reached and passed root return.
199         if (!connection_point)
200             break;
201
202         //> If no height change occurs we can break.
203         if (tree_copy_root->height + 1 <= connection_point->height)
204             break;
205
206         //> Copy the current node and link it to the local copy.
207         avl_node_t *curr_cp = avl_node_new_copy(connection_point, tdata);
208         nodes_to_free[++(*ntf_top)] = connection_point;
209         nodes_alloced[++(*nalloced_top)] = curr_cp;
210         ht_insert(tdata->ht, &connection_point->left, curr_cp->left);
211         ht_insert(tdata->ht, &connection_point->right, curr_cp->right);
212
213         curr_cp->height = tree_copy_root->height + 1;
214         if (key < curr_cp->key) curr_cp->left = tree_copy_root;
215         else                    curr_cp->right = tree_copy_root;
216         tree_copy_root = curr_cp;
217
218         // Move one level up
219         *connection_point_stack_index = stack_top;
220         connection_point = stack_top >= 0 ? node_stack[stack_top--] : NULL;
221
222         // Get current node's balance
223         avl_node_t *sibling;
224         int curr_balance;
225         if (key < curr_cp->key) {
226             sibling = curr_cp->right;
```

```
227             curr_balance = node_height(curr_cp->left) - node_height(sibling);
228         } else {
229             sibling = curr_cp->left;
230             curr_balance = node_height(sibling) - node_height(curr_cp->right);
231         }
232 
233         if (curr_balance == 2) {
234             int balance2 = node_balance(tree_copy_root->left);
235 
236             if (balance2 == 1) { // LEFT-LEFT case
237                 tree_copy_root = rotate_right(tree_copy_root);
238             } else if (balance2 == -1) { // LEFT-RIGhtT case
239                 tree_copy_root->left = rotate_left(tree_copy_root->left);
240                 tree_copy_root = rotate_right(tree_copy_root);
241             } else {
242                 assert(0);
243             }
244 
245             break;
246         } else if (curr_balance == -2) {
247             int balance2 = node_balance(tree_copy_root->right);
248 
249             if (balance2 == -1) { // RIGhtT-RIGhtT case
250                 tree_copy_root = rotate_left(tree_copy_root);
251             } else if (balance2 == 1) { // RIGhtT-LEFT case
252                 tree_copy_root->right = rotate_right(tree_copy_root->right);
253                 tree_copy_root = rotate_left(tree_copy_root);
254             } else {
255                 assert(0);
256             }
257 
258             break;
259         }
260     }
261 
262     *tree_copy_root_ret = tree_copy_root;
263     return connection_point;
264 }
```

Η υλοποίηση αυτή είναι περίπου 264 γραμμές αρκετά δυσνόητου κώδικα που παρεκκλίνει αρκετά από μία σειριακή υλοποίηση. Περιέχει:

1. Κινδύνους συγχρονισμού για την υλοποιήση των συναλλαγών
2. Περίπλοκη λογική δημιουργίας αντιγράφων
3. Ανάγκη μεταφοράς πολλών παραμέτρων μεταξύ συναρτήσεων, πολλές από τις οποίες σχετίζονται μονάχα με το συγχρονισμό και άσχετες με την λογική της δομής
4. Πολλούς ελέγχους ορίων που θα μπορούσαν εύκολα να επηρεαστούν από ανθρώπινο λάθος

Επιπλέον, καθιστά το debugging εξαιρετικά δύσκολο λόγω των πολλών μεταβλητών και παρέχει περιορισμένες δυνατότητες επαναχρησιμοποίησης του κώδικα για άλλες δομές. Τέλος, μοιάζει ελάχιστα με μία σειριακή υλοποίηση. Αυτό συμβαίνει διότι πρέπει να υλοποιήσει ολόκληρη τη λογική της βιβλιοθήκης μας για κάθε μέθοδο ξεχωριστά, χρησιμοποιώντας δύσχρηστα primitives.

4.1.0.2 Βιβλιοθήκη

Ας το δούμε υλοποιημένο μέσω της βιβλιοθήκης:

```
 bool insert(const int k, ValueType val, int t_id) {
        return insert_impl(k,val, t_id);
}

// the insert operation
bool insert_impl(const int k, ValueType val, int t_id) {

    int retries = trans_retries;

    TM_SAFE_OPERATION_START {

        TSX::Transaction t(retries,_lock, stats[t_id]);

        /* FIND PhtASE */

        auto conn_point_snapshot = find_conn_point<TreeNode>(k,&root);


        if (conn_point_snapshot.found()) {
            return false;
        }


        /* FIND PhtASE END */

        ConnPoint<TreeNode> conn(conn_point_snapshot, t);

        /* INSERT */

        auto node_to_be_inserted = conn.create_safe(
        ConnPoint<TreeNode>::create_new_node(k,val,nullptr,nullptr));


        // insert it
        conn.setRoot(node_to_be_inserted);

        // identical to bst up to this point


        // if not inserting at root
```

```
41          if (conn_point_snapshot.connection_point()) {
42
43              bool rotation_happened = false;
44
45              /* REBALANCE */
46
47              // go up path and rebalance
48              for (SafeNode<TreeNode>* n = conn.pop_path();
49                                          n != nullptr; n = conn.pop_path()) {
50
51                  auto n_data = n->rwRef();
52                  int height_old = n_data->height;
53
54                  n = rebalance_ins(n, k , rotation_happened);
55                  // apply rebalancing to all
56                  // required nodes
57                  // rotation returns the new root to place
58                  // below the connection point
59
60                  conn.setRoot(n);
61                  // change the root of the
62                  // tree of copies to make the change visible
63
64                  if (height_old == n_data->height && !rotation_happened) {
65                      break;
66                  }
67
68              }
69          }
70      } TM_SAFE_OPERATION_END
71
72      return true;
73  }
74
75     // apply rebalancing for an insertion operation
76          SafeNode<TreeNode>* rebalance_ins(SafeNode<TreeNode>* n,
77                                      int k, bool& rotation_happened) {
78
79               // reads and writes are safe
80              auto n_data = n->rwRef();
81
82              // calculate node height
83              n_data->height =
```

```
84                  max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
85
86
87              // calculate node's balance
88              auto balance = node_balance(n_data);
89
90              rotation_happened = true;
91
92              // for tree modifications (change children etc) use SafeNode
93
94              if (balance > 1 && k < n_data->getChild(0)->key) {
95                  // right rotate
96                  n = right_rotate(n);
97              } else if (balance < -1 && k > n_data->getChild(1)->key) {
98                  //left rotate
99                  n = left_rotate(n);
100             } else if (balance > 1 && k > n_data->getChild(0)->key) {
101                 // left right rotate
102                 n->setChild(0, left_rotate(n->getChild(0)));
103
104                 n_data->height =
105                     max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
106
107                 n = right_rotate(n);
108             } else if (balance < -1 && k < n_data->getChild(1)->key) {
109                 // right Left Rotate
110                 n->setChild(1,right_rotate(n->getChild(1)));
111
112                 n_data->height =
113                     max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
114
115                 n = left_rotate(n);
116             } else {
117                 rotation_happened = false;
118             }
119
120             // calculate new height
121             n_data->height = max_height(n_data->getChild(0), n_data->getChild(1)) + 1;
122
123             return n;
124 }
```

4.1.0.3 Αξιολόγηση Παράλληλου Κώδικα

Η υλοποίηση της μεθόδου είναι περίπου 107 γραμμές κώδικα (60% μείωση) με αρκετά πιο ξεκάθαρη λειτουργία και πολλές ομοιότητες με σειριακό κώδικα. Οι ομοιότητες μάλιστα είναι τόσο εκτεταμένες που συγκεκριμένα σημεία δεν χρειάζονται καμία αλλαγή για την παραλληλοποίηση τους από τη βιβλιοθήκη. Ο κώδικας συγχρονισμού ελαχιστοποιείται και η συνάρτηση αποτελείται κυρίως από τη λογική της πράξης της δομής. Το αποτέλεσμα αυτό, το αξιολογούμε ως μεγάλη νίκη για την παραγωγικότητα του προγραμματιστή.

4.2 Πειραματική Μέθοδος

Μηχάνημα Δοκιμών Μετρήσαμε την απόδοση της βιβλιοθήκης σε έναν υπολογιστή εξοπλισμένο με δύο επεξεργαστές Intel(R) Xeon(R) E5-2697 v3, σε ξεχωριστά sockets, όπου ο καθένας έχει 14 επεξεργαστικούς πυρήνες και υποστηρίζει έως 28 νήματα εκτέλεσης βασισμένα σε υλικό μέσω hyperthreading. Περιοριστήκαμε στη χρήση ενός από τους δύο επεξεργαστές, λόγω της επίδρασης της NUMA πρόσβασης στη μνήμη. Το σύστημα ήταν εξοπλισμένο με 200GB μνήμης RAM.

Κατασκευάσαμε δύο δέντρα με βάση τη βιβλιοθήκη, ένα απλό δυαδικό δέντρο αναζήτησης και ένα AVL δέντρο αναζήτησης. Για τη μέτρηση της επίδοσης, δημιουργούμε σειριακά ένα στιγμιότυπο αυτών των δέντρων με συγκεκριμένο αριθμό κόμβων με τυχαία κλειδιά εντός ενός διαστήματος τιμών (πχ. 10000 κλειδιά στο διάστημα [0,20000]), με το διάστημα τιμών να είναι διπλάσιο σε μέγεθος από το πλήθος των κόμβων.

Στη συνέχεια δημιουργούμε νήματα που αντιστοιχίζονται με pinning στους διαθέσιμους πυρήνες του συστήματος και με δεδομένες πιθανότητες, κάνουν συνεχώς πράξεις του δέντρου (εισαγωγή, αναζήτηση, διαγραφή) για 5 δευτερόλεπτα. Κρατώντας ίσες τις πιθανότητες εισαγωγής και διαγραφής, το μέγεθος του δέντρου δεν αλλάζει σημαντικά από την αρχική κατάσταση. Μετράμε τον αριθμό πράξεων που ολοκληρώθηκαν ανα δευτερόλεπτο με τη μετρική MOPS(εκατομμύρια πράξεις ανα δευτερόλεπτο).

Σαν υλοποιήσεις χρησιμοποιούνται τα δύο δέντρα της βιβλιοθήκης (*avl-framework,unbalanced-framework*) και οι χειροκίνητες υλοποιήσεις (*avl-manual, unbalanced-manual*) του εισηγητή μου Δημήτρη Σιακαβάρα.

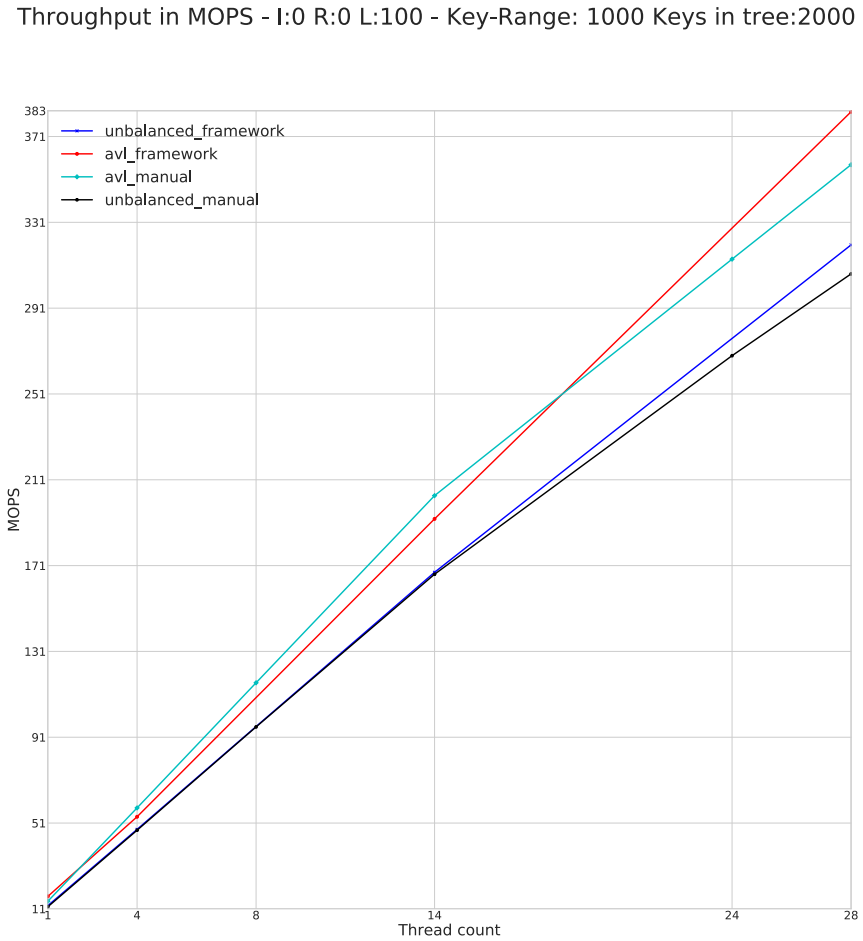
4.3 Αποτελέσματα

Στις μετρήσεις μας ενδιαφέρει η σύγκριση με την χειροκίνητη υλοποίηση της τεχνικής RCU-htTM και όχι η απόλυτη επίδοση, εφ'όσον η βιβλιοθήκη έχει ως σκοπό την αυτόματη εφαρμογή της τεχνικής. Μετράμε,λοιπόν,το κόστος χρήσης της βιβλιοθήκης.

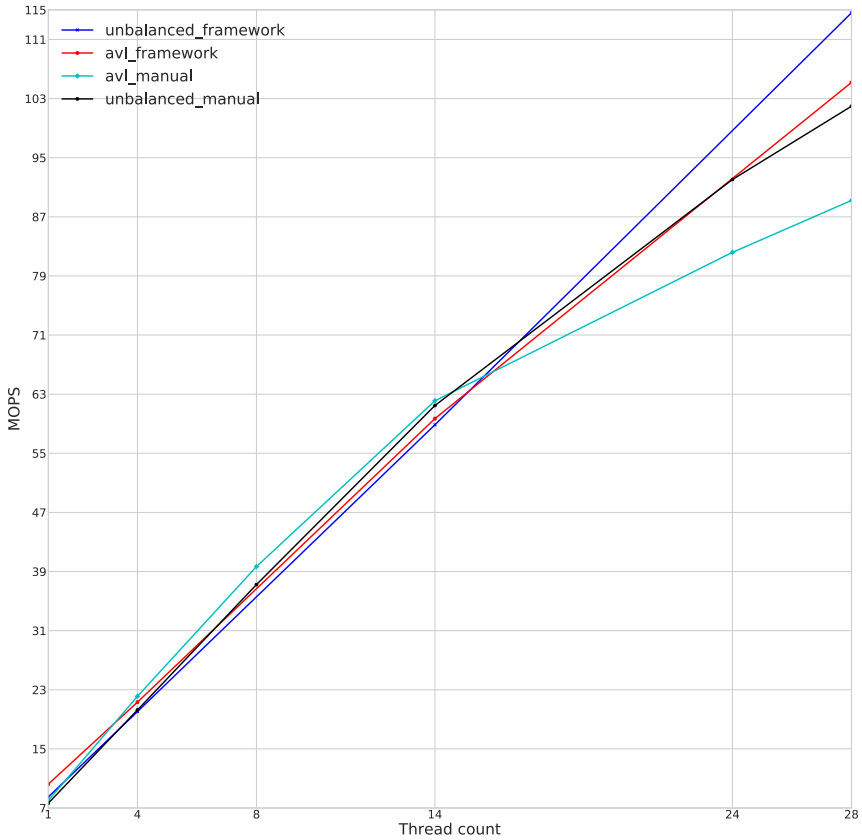
Ως *I,R,L* παριστάνονται οι πιθανότητες εισαγωγής, διαγραφής και αναζήτησης, ως *Key Range* το διάστημα τιμών των κλειδιών των κόμβων (με το 0 ως αριστερό άκρο) και ως *Keys in Tree*, ο αριθμός κόμβων με τον οποίο αρχικοποιήθηκαν τα δέντρα.

4.3.1 Εργασίες Αναζήτησης

4.3.1.1 Μικρά Δέντρα

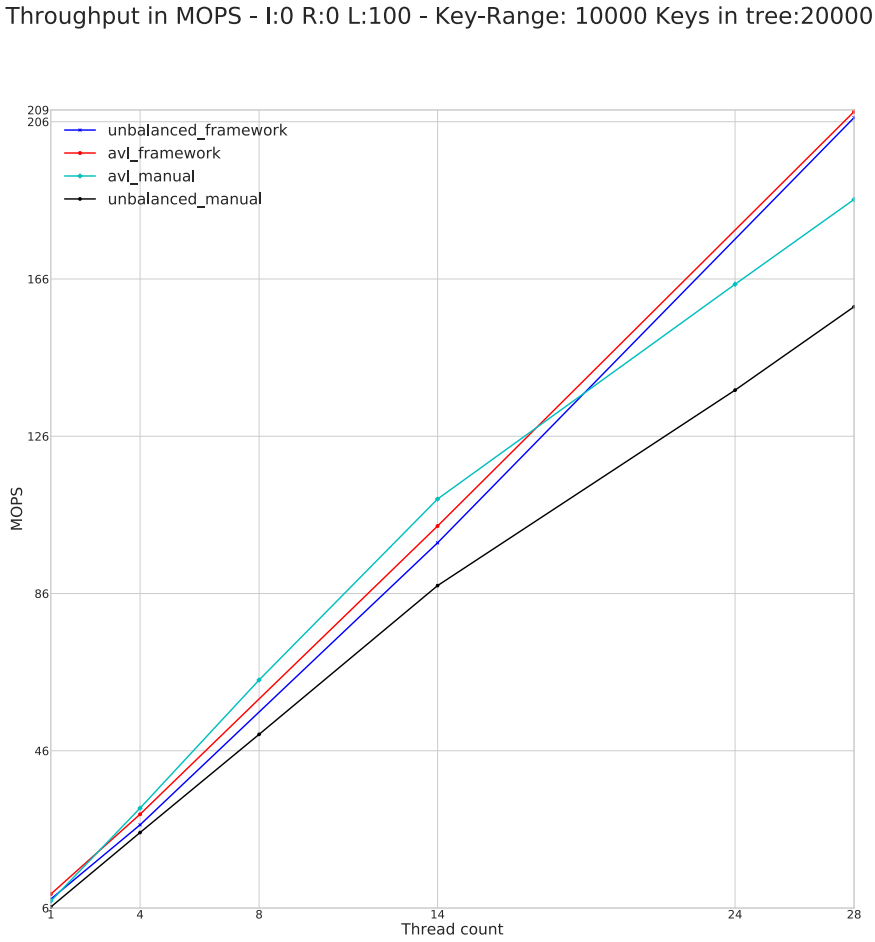


Σχήμα 4.1: 100% Αναζήτηση

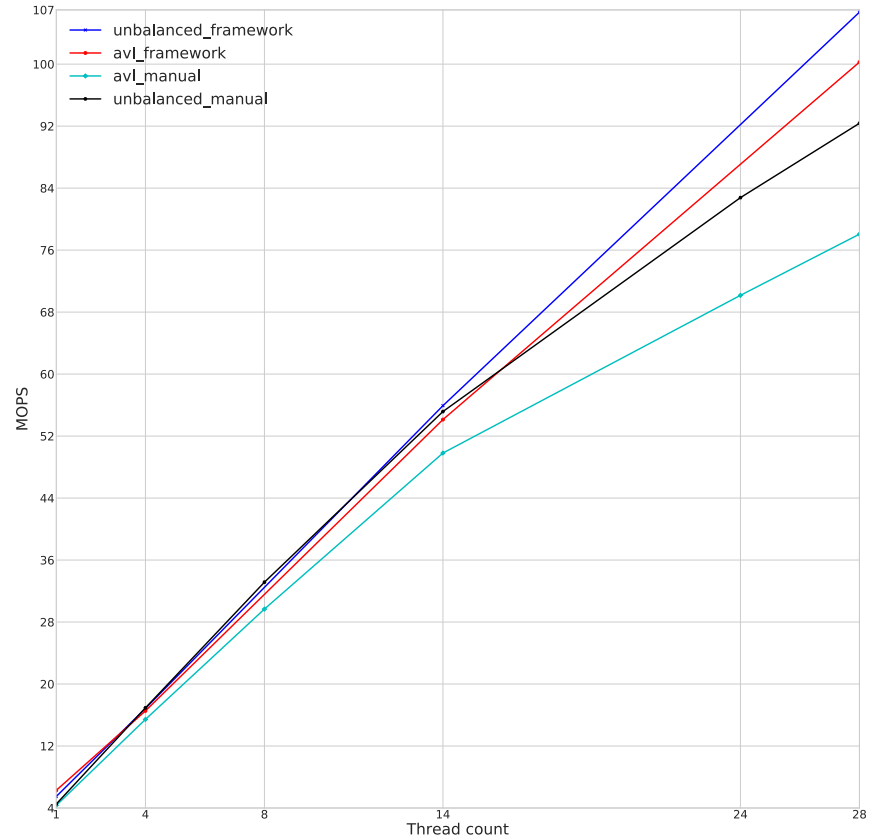


Σχήμα 4.2: 80% Αναζήτηση

4.3.1.2 Μεσσαία Δέντρα

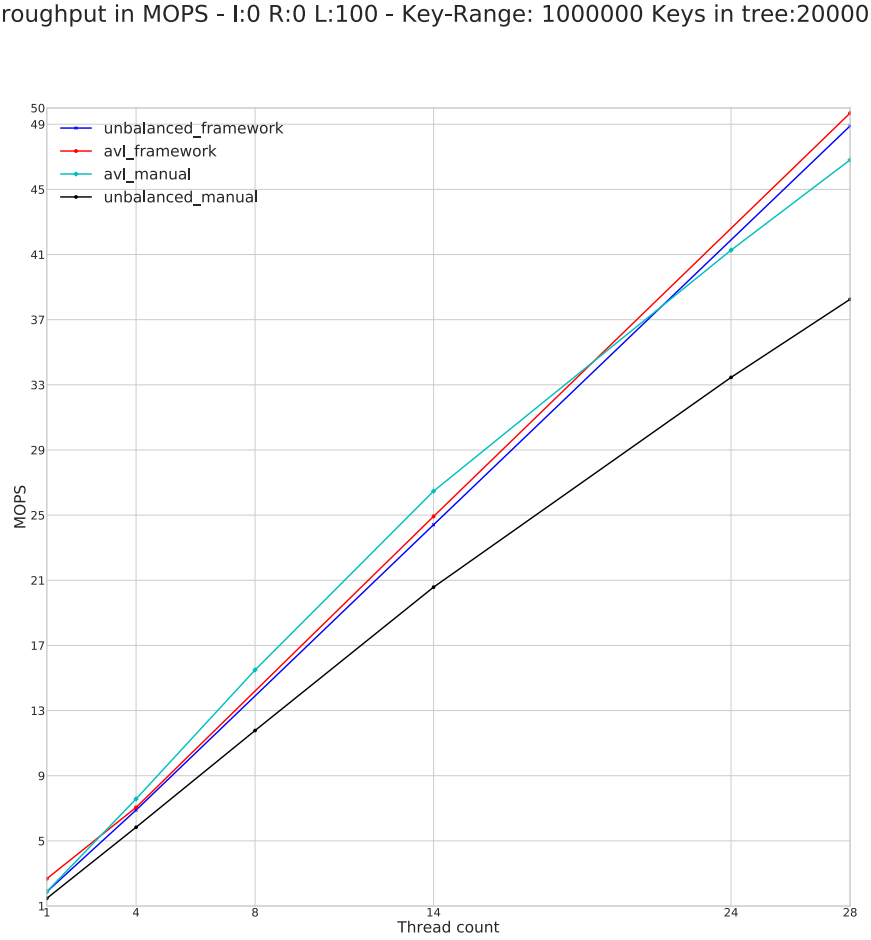


Σχήμα 4.3: 100% Αναζήτηση

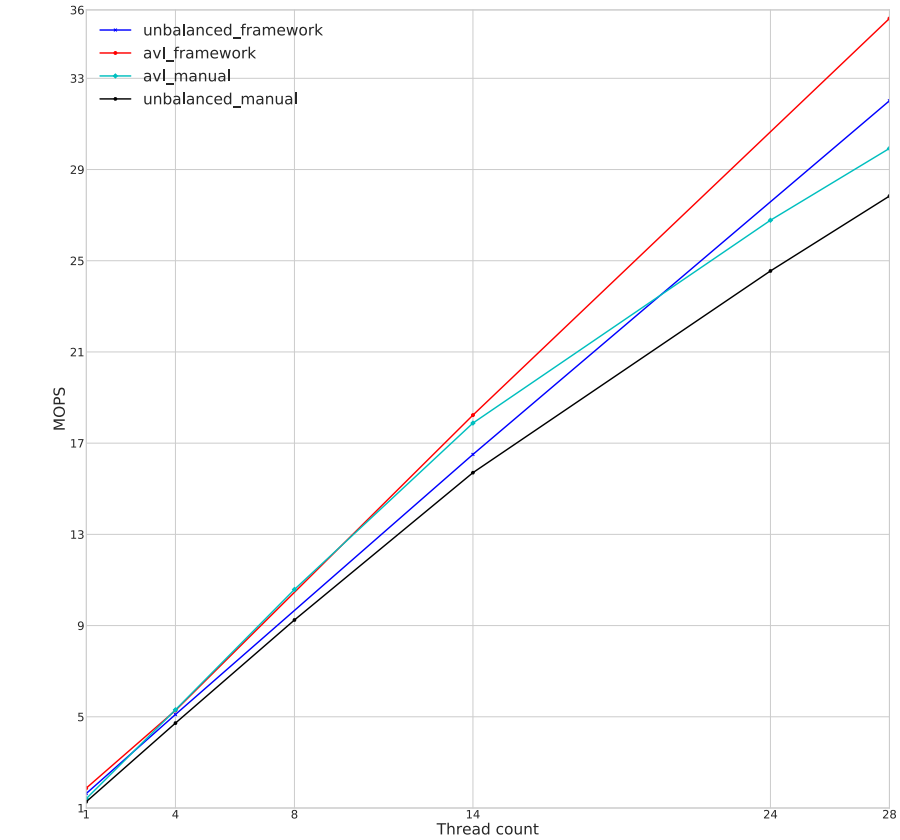


Σχήμα 4.4: 80% Αναζήτηση

4.3.1.3 Μεγάλα Δέντρα



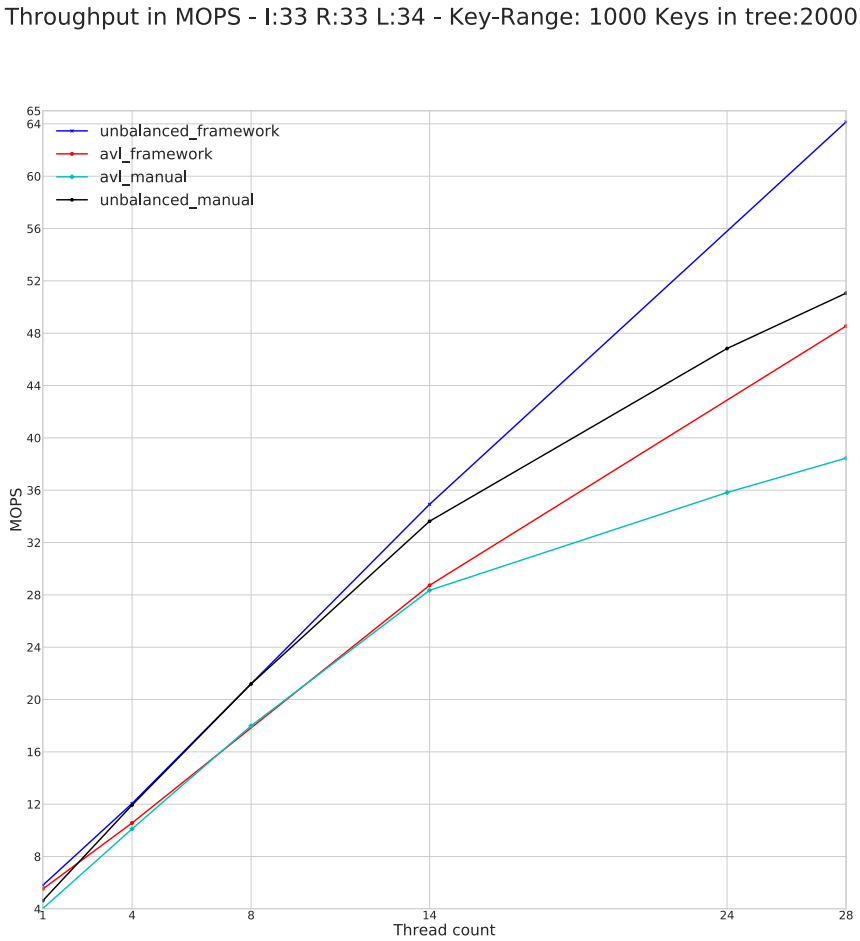
Σχήμα 4.5: 100% Αναζήτηση



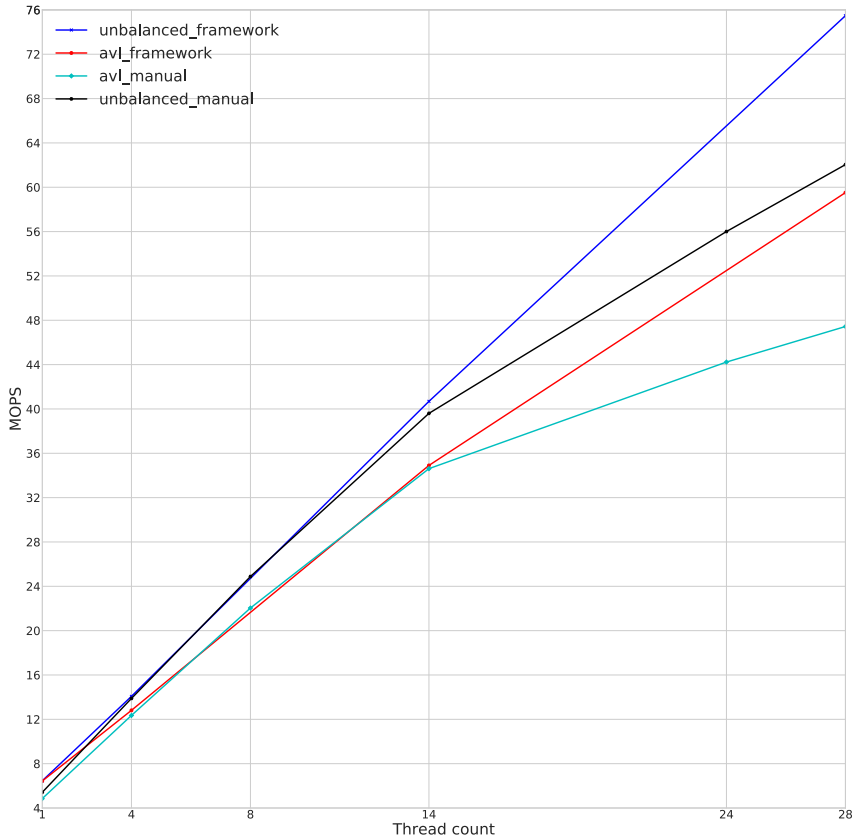
Σχήμα 4.6: 80% Αναζήτηση

4.3.2 Ισομοιρασμένες Εργασίες

4.3.2.1 Μικρά Δέντρα

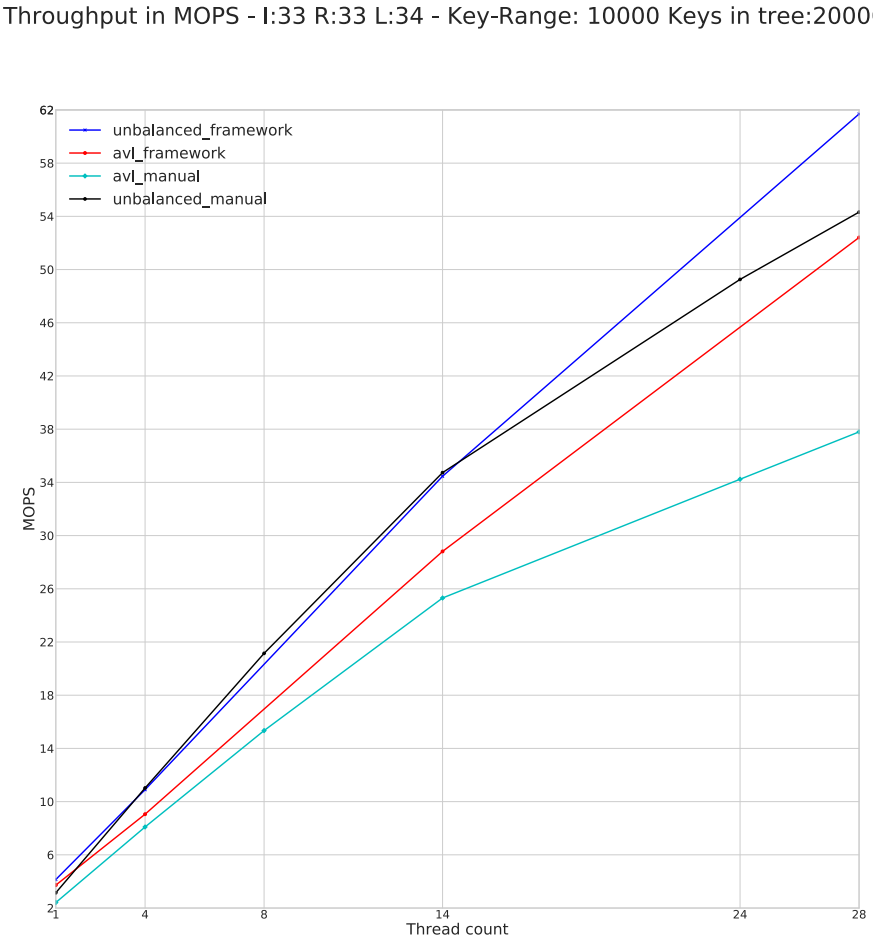


Σχήμα 4.7: Ισομοιρασμένες Πράξεις

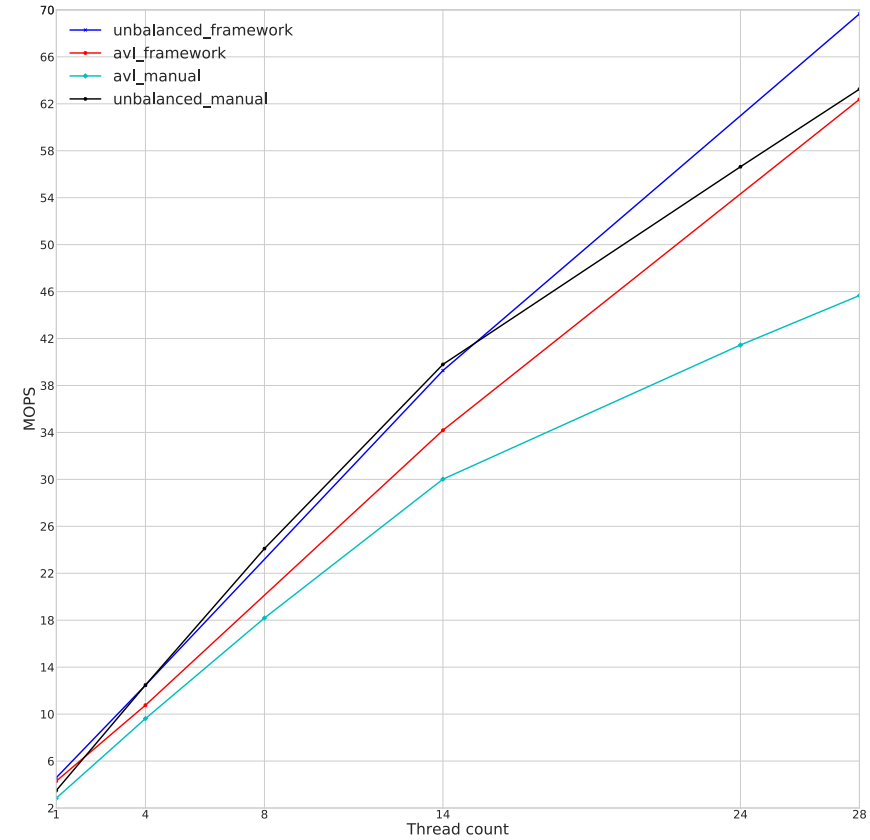


Σχήμα 4.8: 50% Αναγνώσεις

4.3.2.2 Μεσσαία Δέντρα

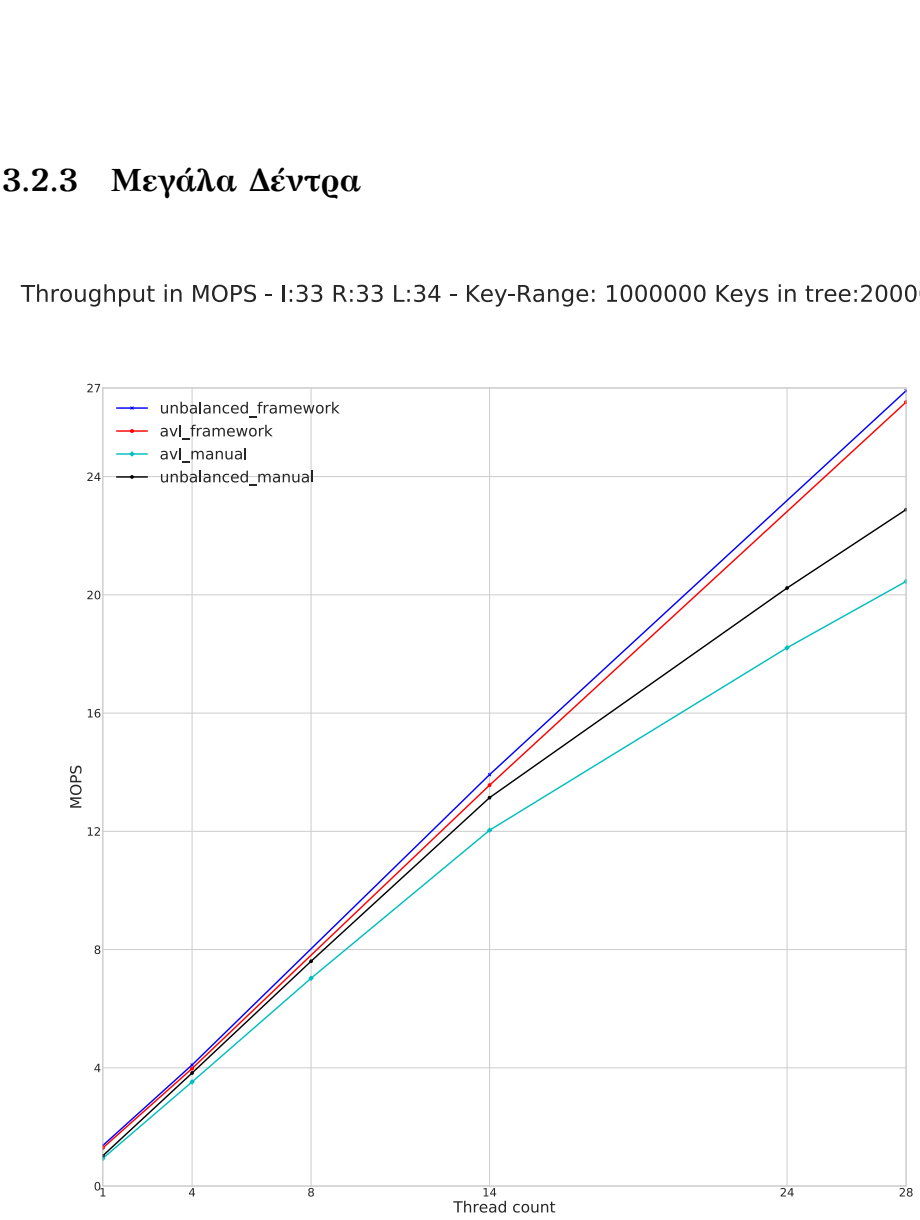


Σχήμα 4.9: Ισομοιρασμένες Πράξεις

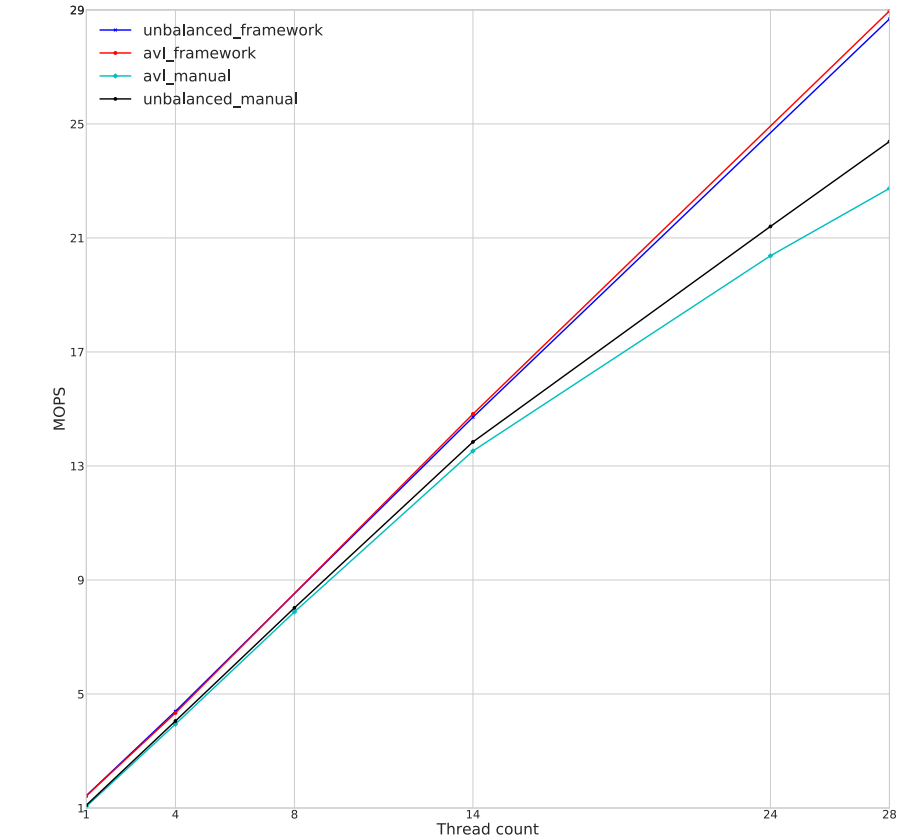


Σχήμα 4.10: Μισές Αναγνώσεις

4.3.2.3 Μεγάλα Δέντρα



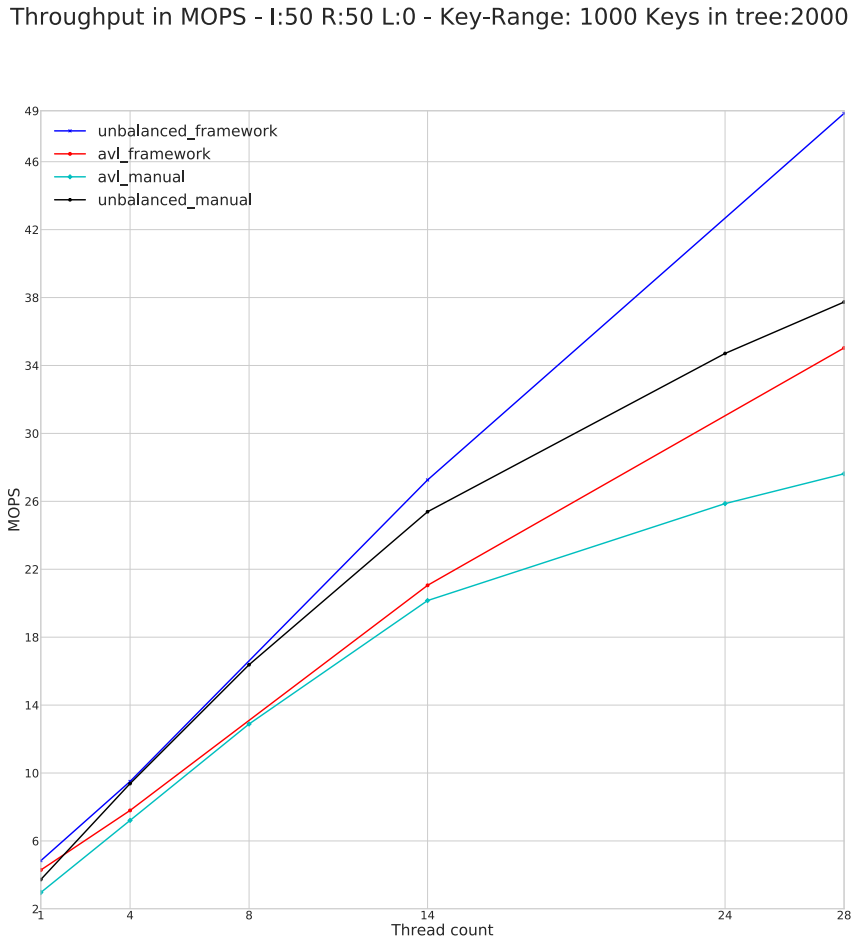
Σχήμα 4.11: Ισομοιρασμένες Πράξεις



Σχήμα 4.12: Μισές Αναγνώσεις

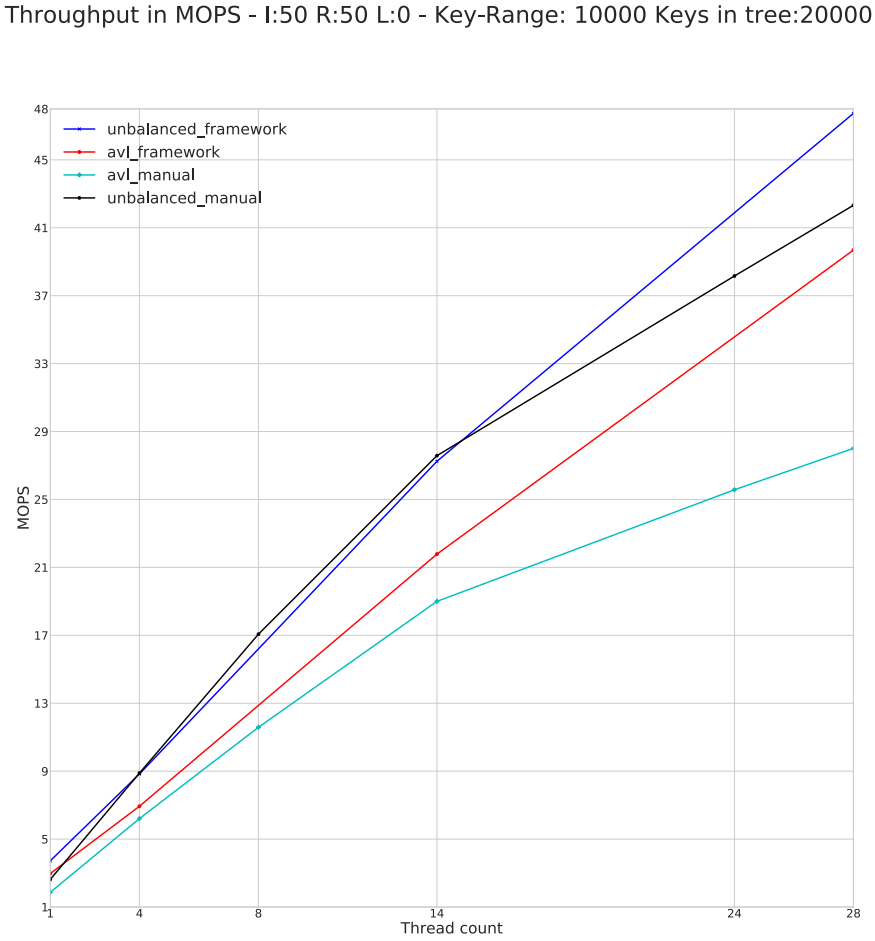
4.3.3 Μόνο Αλλαγές

4.3.3.1 Μικρά Δέντρα



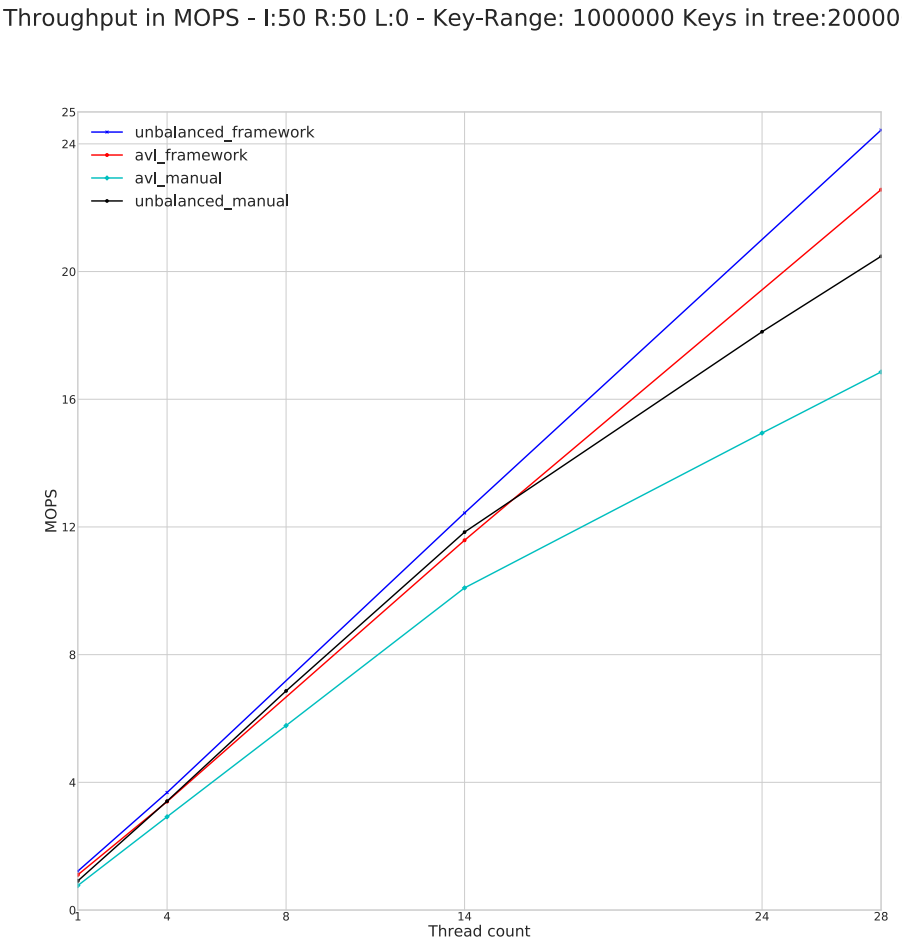
Σχήμα 4.13: Μόνο Αλλαγές

4.3.3.2 Μεσσαία Δέντρα



Σχήμα 4.14: Μόνο Αλλαγές

4.3.3.3 Μεγάλα Δέντρα



Σχήμα 4.15: Μόνο Αλλαγές

4.3.3.4 Ανάλυση απόδοσης

Σύγκριση με χειροκίνητη υλοποίηση Βλέπουμε ότι σε όλα τα μεγέθη δέντρων και για κάθε σύνολο εργασίας, οι **αυτόματα παραλληλοποιημένες υλοποιήσεις υπερτερούν σε απόδοση σε σχέση με τις χειροκίνητες υλοποιήσεις**. Τα ακριβή νούμερα δεν έχουν σημασία, διότι μπορούμε να θεωρήσουμε πως η χειροκίνητη υλοποίηση δεν είναι ιδανική (μία ιδανική υλοποίηση πιθανώς θα μπορούσε να φτάσει την απόδοση της αυτόματης), κάτι λογικό λόγω της πολυπλοκότητας της. Ωστόσο αυτό αποτελεί απόδειξη ότι η αυτοματοποίηση δεν επιφέρει κόστος συγκριτικά με τη χειροκίνητη υλοποίηση, άρα οι προκύπτουσες δομές είναι εξίσου ανταγωνιστικές με τις εναλλακτικές υλοποιήσεις παράλληλων δομών όσο η χειροκίνητες υλοποιήσεις RCU-htTM(βλ. [Sia+17]). Επιπλέον, η ευκολία χρήσης της βιβλιοθήκης οδηγεί σε πιο αποδοτικές βιβλιοθήκες κρύβοντας την δυσκολία της βελτιστοποίησης.

Γενικά περί απόδοσης Βλέπουμε πως οι υλοποιήσεις των παράλληλων δομών της βιβλιοθήκης παρουσιάζουν σχεδόν γραμμική αύξηση απόδοσης με την αύξηση των νημάτων. Αυτό παρατηρείται για κάθε μέγεθος δέντρου και σε κάθε σύνολο εργασίας.

4.4 Επίδραση NUMA στην απόδοση

Στις μετρήσεις μας, χρησιμοποιήσαμε και ένα σύστημα που αποτελούταν από δύο επεξεργαστές XEON σε ξεχωριστά sockets και χρησιμοποιούσε σύστημα μνήμης ετερογενούς πρόσβασης (NUMA). Στο συγκεκριμένο σύστημα, παρατηρήσαμε ότι η απόδοση σε συνάρτηση με τα χρησιμοποιούμενα νήματα έπεφτε απότομα όταν τα νήματα εκτελούνταν σε πυρήνες διαφορετικών sockets. Όσο αυξανόταν ο αριθμός των πυρήνων από διαφορετικά sockets, η απόδοση χειροτέρευε, τείνοντας σε αυτή που είχαμε με τη χρήση ενός η δύο νημάτων.

Ερευνώντας το ζήτημα ανακαλύψαμε ότι κατά την πρόσβαση στην μνήμη ετερογενούς πρόσβασης αυτού του συστήματος, δεδομένου ότι η δομή έπρεπε να βρίσκεται σε κοινή μνήμη, προκαλούνται πολλά page faults τα οποία προκαλούν συνεχή aborts των συναλλαγών HTM, καταστρέφοντας την απόδοση.

Το ζήτημα αυτό δεν αναλύθηκε στα πλαίσια της διπλ. εργασίας αλλά είναι αξιοπρόσεκτο σε περίπτωση που χρησιμοποιηθεί η βιβλιοθήκη με συστήματα μνήμης (NUMA).

4.5 Παράγοντες που επηρεάζουν την απόδοση της βιβλιοθήκης

Στην παρούσα ενότητα θα αναφέρουμε μερικούς σημαντικούς παράγοντες στην απόδοση των δομών που παραλληλοποιούνται με τη χρήση της δομής. Οι παρατηρήσεις αυτές έγιναν κατά τη διάρκεια της υλοποίησης των δύο δομών που αξιολογήθηκαν.

1. **Alignment των αρχικών κόμβων**, το μέγεθος της ευθυγράμμισης των αρχικών κόμβων παίζει σημαντικό ρόλο όταν τα δέντρα αποτελούνται από λίγους κόμβους (λιγότερους απο 100). Σε αυτές τις περιπτώσεις, τυχαίνει κόμβοι διαφορετικών υποδέντρων να τοποθετούνται στην ίδια γραμμή κρυφής μνήμης, επομένως λόγω false sharing, να προκαλούνται aborts στις συναλλαγές, καταστρέφοντας την απόδοση όταν χρησιμοποιούνται πολλά νήματα. Αυξάνοντας την ευθυγράμμιση των κόμβων σε μεγαλύτερα μεγέθη, ελαχιστοποιείται το φαινόμενο αλλά με τη μεγέθυνση των αντικειμένων αυξάνεται το κόστος αντιγραφής των κόμβων. Ενδείκνυται πειραματισμός σε περίπτωση προβλημάτων στην απόδοση

2. **Αριθμός κλήσεων `getChild|`** , η χρήση της μεθόδου `getChild` οδηγεί στη δημιουργία αντιγράφων για κάθε πρόσβαση σε κάποιο κόμβο ο οποίος δεν έχει τοποθετηθεί σε ασφαλή κόμβο. Για αυτό το λόγο, πρέπει να ελαχιστοποιούνται αυτές οι προσβάσεις.

3. **Δυναμική Διαχείριση Μνήμης `getChild|`** , η χρήση δυναμικής διαχείρισης μνήμης επηρεάζει αρνητικά την απόδοση αφού οδηγεί με μεγάλη συχνότητα σε aborts συναλλαγών είτε λόγω πρόκλησης page fault, είτε λόγω πρόσβασης σε κοινές κρυφές δομές του συστήματος διαχείρισης μνήμης.

Κεφάλαιο 5

Συμπεράσματα και Μελλοντικές Προσθήκες

5.1 Συμπεράσματα

Επισκεπτόμενοι ξανά τους στόχους που θέσαμε στο κεφάλαιο της βιβλιοθήκης, συμπεραίνουμε ότι το αποτέλεσμα επιτυγχάνει κάθε ένα ξεχωριστά.

1. Οι κίνδυνοι και οι δυσκολίες του παράλληλου κώδικα κρύβονται και αντιμετωπίζονται από την βιβλιοθήκη και όχι από τον χρήστη, ο οποίος καλείται να γράψει κώδικα παρόμοιο με σειριακό

2. Ο τελικός κώδικας έχει ελάχιστες διαφορές από το σειριακό και είναι εύκολος στην κατανόηση, δίνοντας αυτόματα έμφαση στη λογική λειτουργίας της δομής και όχι στον συγχρονισμό

3. Η απόδοση της βιβλιοθήκης αντιστοιχεί σε μία εξαιρετική χειροκίνητη εφαρμογή της τεχνικής RCU-HTM, άρα αποτελεί ένα zero-cost abstraction

4. Η αποσφαλμάτωση αποδεικνύεται ευκολότερη αφού ο χρήστης παρέχει αποκλειστικά σειριακές μεθόδους, τις οποίες πρέπει να ελέγξει, αρκεί να ακολουθεί τις συμβάσεις της βιβλιοθήκης

Επιπλεόν η βιβλιοθήκη, επεκτείνει την τεχνική RCU-HTM, επιτρέποντας την εύκολη παραλληλοποίηση της πλειονότητας των δενδρικών δομών που χρησιμοποιούνται είτε στην παραγωγή είτε ακαδημαϊκά. Συμπεραίνουμε ,λοιπόν, ότι η βιβλιοθήκη αποτελεί ένα καινοτόμο εργαλείο που μπορεί να διευκολύνει σε τεράστιο βαθμό την κατασκευή παράλληλων δεντρικών δομών δεδομένων, βελτιώνοντας κάθε στάδιο κατασκευής τους από το σχεδιασμό έως την επιβεβαίωση της συνοχής τους.

5.2 Μελλοντικές Προσθήκες

Δεδομένων των πολλών δυνατοτήτων της βιβλιοθήκης, ενδιαφέρον έχει να τονιστεί τι ελλείψεις έχει και ποιες μελλοντικές προσθήκες μπορούν να την βελτιώσουν σημαντικά.

1. Η βασική έλλειψη της βιβλιοθήκης είναι ένας μηχανισμός διαχείρισης δυναμικής μνήμης. Η διαχείριση δυναμικής μνήμης στις παράλληλες δομές δεδομένων αποδεικνύεται μεγάλη πρόκληση, ισάξια ίσως στην δημιουργία αυτής της βιβλιοθήκης. Η βιβλιοθήκη παρέχει μηχανισμούς δημιουργίας των ασφαλών κόμβων αλλά και των κόμβων χρήστη μέσω των pools, αλλά δεν υπάρχει τρόπος ασφαλούς ανάκτησης της μνήμης. Αυτό τίθεται ως βασική προτεραιότητα μελλοντικών προσθηκών στη βιβλιοθήκη

2. Εναλλακτικοί μηχανισμοί HTM, πέρα από την υλοποίηση της Intel (πχ. ARM, κάποιο software fallback), διότι η τωρινή υλοποίηση περιορίζεται σε Intel επεξεργαστές που υποστηρίζουν το σύνολο εντολών TSX-NI

3. Μεταφορά βιβλιοθήκης και σε άλλες γλώσσες προγραμματισμού πέραν της C++

Παράρτημα Α΄

Ρυθμίσεις

Compilation Flags

Η βιβλιοθήκη υποστηρίζει ένα αριθμό από ρυθμίσεις και λειτουργίες που ελέγχονται από compilation flags.

Αριθμητικές Παράμετροι

1. `RCU_HTM_MAX_THREADS`, μέγιστος αριθμός νημάτων, ρυθμίζεται από τη μεταβλητή (προεπιλογή: 100)

2. `PATH_MAX_LEN`, μέγιστο μήκος μονοπατιού από τη ρίζα μέχρι οποιοδήποτε κόμβο (προεπιλογή: 10000)

Μέθοδοι Λειτουργίας - MODES

1. `TREE_TYPE`, τύπος δέντρου, δέχεται τιμές `SEARCH_TREE` και `GENERAL_TREE` για την μέθοδο δέντρων αναζήτησης και γενικής δενδρικής δομής αντίστοιχα

2. `TM_EARLY_ABORT_POP`, ενεργοποιεί την πρόωρη ακύρωση πραξης αν κατά την μετακίνηση σημείου σύνδεσης αποτύχει η επιβεβαίωση

3. `TM_EARLY_ABORT_ALL`, ενεργοποιεί την ενεργοποιεί την πρόωρη ακύρωση πράξης αν κατά την διάρκεια μίας πράξης αποτύχει η εκτός συναλλαγής επιβεβαίωση (ενεργοποιεί και την σημαία `TM_EARLY_ABORT_POP`)

Βελτιστοποιήσεις

1. Ενεργοποίηση των pools για ασφαλείς κόμβους, με τον ορισμό της μεταβλητής `TSX_MEM_POOL`
2. Ενεργοποίηση των pools για αντικείμενα κόμβου, με τον ορισμό της μεταβλητής `USER_MEM_POOL`
3. Χρήση stack-allocated συνόλου επιβεβαίωσης με τον ορισμό της μεταβλητής `PREALLOC_VALIDATION_SET`

Βιβλιογραφία

[Bro+10] Nathan G. Bronson κ.ά. "A Practical Concurrent Binary Search Tree". Στο: *SIGPLAN Not.* 45.5 (Ιαν. 2010), σσ. 257–268. ISSN: 0362-1340. DOI: 10.1145/1837853.1693488. URL: https://doi.org/10.1145/1837853.1693488.

[Des+12] Mathieu Desnoyers κ.ά. "User-Level Implementations of Read-Copy Update". Στο: *IEEE Transactions on Parallel and Distributed Systems* 23 (2012), σσ. 375–382. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159.

[CGR13] Tyler Crain, Vincent Gramoli και Michel Raynal. "A Contention-Friendly Binary Search Tree". Στο: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, σσ. 229–240. ISBN: 9783642400469. DOI: 10.1007/978-3-642-40047-6_25. URL: https://doi.org/10.1007/978-3-642-40047-6_25.

[CNT14] Bapi Chatterjee, Nhan Nguyen και Philippas Tsigas. *Efficient Lock-free Binary Search Trees*. 2014. arXiv: 1404.3272 [cs.DC].

[RM15] Arunmoezhi Ramachandran και Neeraj Mittal. "A Fast Lock-Free Internal Binary Search Tree". Στο: *Proceedings of the 2015 International Conference on Distributed Computing and Networking*. ICDCN '15. Goa, India: Association for Computing Machinery, 2015. ISBN: 9781450329286. DOI: 10.1145/2684464.2684472. URL: https://doi.org/10.1145/2684464.2684472.

[Sia+15] Dimitrios Siakavaras κ.ά. "Performance Analysis of Concurrent Red-Black Trees on HTM Platforms". Στο: Ιαν. 2015.

[Nam16] Dmitry Namiot. "On lock-free programming patterns". Στο: *WSEAS Transactions on Computers* 15 (Ιαν. 2016), σσ. 117–124.

[BER17] Trevor Brown, Faith Ellen και Eric Ruppert. *A General Technique for Non-blocking Trees*. 2017. arXiv: 1712.06687 [cs.DC].

[Sia+17] D. Siakavaras κ.ά. "RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees". Στο: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Σεπτ. 2017, σσ. 1–13. DOI: 10.1109/PACT.2017.17.

[WSJ18] Kjell Winblad, Konstantinos Sagonas και Bengt Jonsson. "Lock-Free Contention Adapting Search Trees". Στο: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. SPAA '18. Vienna, Austria: Association for Computing Machinery, 2018, σσ. 121–132. ISBN: 9781450357999. DOI: 10.1145/3210377.3210413. URL: https://doi.org/10.1145/3210377.3210413.