ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη και Αξιολόγηση του TSX στους Haswell επεξεργαστές της Intel: Παραλληλοποίηση Red Black Tree

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΕΩΡΓΙΟΥ ΜΑΠΠΟΥΡΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

Η σελίδα αυτή είναι σκόπιμα λευκή.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη και Αξιολόγηση του TSX στους Haswell επεξεργαστές της Intel: Παραλληλοποίηση Red Black Tree

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΕΩΡΓΙΟΥ ΜΑΠΠΟΥΡΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Ιουλίου 2014

.....	.....	.....
Νεκτάριος Κοζύρης	Παναγιώτης Δ. Τσανάκας	Δημήτριος Τσουμάκος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ι.Π.

Αθήνα, Ιούλιος 2014

.....

Γεώργιος Μαππούρας

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

Copyright © Γεώργιος Μαππούρας, 2014

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Έχουμε πλέον φτάσει στην εποχή που όλοι οι επεξεργαστές στηρίζονται σε πολυπύρηνες αρχιτεκτονικές για να μπορέσουν να αυξήσουν την απόδοσή τους και να παρακάμψουν τους τεχνολογικούς περιορισμούς που συναντούσαν τα μονοπύρηνα συστήματα. Ωστόσο η ταυτόχρονη αξιοποίηση πολλαπλών πυρήνων από μια εφαρμογή δεν είναι εύκολη διαδικασία.

Στην προσπάθεια να υλοποιήσουμε παράλληλα προγράμματα που θα χρησιμοποιούν πολλαπλά νήματα (threads), ώστε να εκμεταλλεύονται όλους τους πυρήνες του επεξεργαστή ήρθαμε μπροστά στην πρόκληση του συγχρονισμού τους στις προσβάσεις σε κοινή μνήμη. Ο συγχρονισμός αυτός αποδείχτηκε μια περίπλοκη και δύσκολη διαδικασία. Οι συμβατικοί τρόποι συγχρονισμού περιόρισαν την απόδοση των παράλληλων προγραμμάτων και οδήγησαν στην αναζήτηση νέων καλύτερων μεθόδων που θα προσφέρουν μεγαλύτερη κλιμακωσιμότητα.

Την λύση στο πρόβλημα αυτό φαίνεται να δίνει το Transactional Memory. Πιο συγκεκριμένα το Hardware Transactional Memory υπόσχεται να δώσει ένα εύρωστο, απλό και κλιμακώσιμο τρόπο για συγχρονισμό παράλληλων νημάτων σε κοινή μνήμη. Μέσα από τους Haswell επεξεργαστές της Intel που προσφέρουν για πρώτη φορά σε εμπορική μορφή μια υλοποίηση Hardware Transactional Memory, θέλουμε να εξετάσουμε τις δυνατότητές του, να δούμε τα πλεονεκτήματά του, αλλά και τους περιορισμούς που εισάγει η χρήση του.

Ο πειραματισμός και η αξιολόγηση του Hardware Transactional Memory των Haswell επεξεργαστών γίνεται μέσα από την παραλληλοποίηση δομών Red Black Tree. Τα Red Black Trees είναι γνωστά για την περίπλοκη δομή τους και την δυσκολία παραλληλοποίησής τους, με τις μέχρι σήμερα συμβατικές μεθόδους συγχρονισμού.

Λέξεις Κλειδιά: Transactional Memory, Hardware Transactional Memory, Transactional Synchronization Extensions, Red Black Tree

Η σελίδα αυτή είναι σκόπιμα λευκή.

Abstract

Nowadays every processor is based on multi-core architectures to enable them to increase their performance and to circumvent technological limitations encountered in single-core systems. However the simultaneous use of multiple cores by a single application is not an easy process.

Trying to implement parallel programs that use multiple threads, to exploit all CPU cores, we came across the challenge of synchronizing their accesses to shared memory. This synchronization has proved to be a complex and difficult process. Conventional methods of synchronization have limited the performance of parallel programs and led to the pursuit of new better methods that will offer greater scalability.

The solution to this problem seems to be given by the Transactional Memory. More specifically, the Hardware Transactional Memory, promises to provide a robust, simple and scalable way to synchronize parallel threads using shared memory. Through Intel's Haswell processors, that offer for the first time in a commercial form an implementation of Hardware Transactional Memory, we want to observe its capabilities, to find out its advantages and also its limitations.

The experimentation and evaluation of Haswell's Hardware Transactional Memory implementation is becoming possible by implementing parallel Red Black Tree structures. Red Black Trees are known for their complex structure and the difficulty of implementing parallel versions of them, with the conventional synchronization methods.

Keywords: Transactional Memory, Hardware Transactional Memory, Transactional Synchronization Extensions, Red Black Tree

Η σελίδα αυτή είναι σκόπιμα λευκή.

Πίνακας περιεχομένων

1

Εισαγωγή

1.1 Παράλληλος Προγραμματισμός και Προβλήματα Συγχρονισμού

Στην προσπάθεια να δημιουργήσουμε ταχύτερους επεξεργαστές πολύ σύντομα οδηγηθήκαμε σε περιορισμούς που προηγουμένως αγνοούσαμε. Η κατανάλωση ενέργειας, η απαγωγή θερμότητας και η διαφορά μεταξύ της ταχύτητας του επεξεργαστή και της μνήμης πολύ σύντομα μας οδήγησαν στα πολυπύρηνα συστήματα. Σε υπερυπολογιστές, κατανεμημένα συστήματα, προσωπικούς υπολογιστές ακόμα και ενσωματωμένα συστήματα, συναντάμε πλέον πολυπύρηνους επεξεργαστές. Στόχος των πολυπύρηνων επεξεργαστών είναι να μας δώσουν μεγαλύτερες ταχύτητες και καλύτερες επιδόσεις. Ωστόσο η εμπειρία και η τριβή με τα συστήματα αυτά έδειξε πως η εκμετάλλευση και η αξιοποίηση τους δεν είναι απλή υπόθεση. Ενώ κανείς θα περίμενε η επίδοση ενός συστήματος να αυξάνεται ανάλογα με τον αριθμό των επεξεργαστών/πυρήνων που προσθέτει, αυτό δεν γίνεται στην πραγματικότητα. Τα πολυπύρηνα συστήματα φέρνουν μαζί τους προβλήματα επικοινωνίας και συγχρονισμού. Επίσης μεγάλο πρόβλημα είναι η κλιμακωσιμότητα των συστημάτων αυτών καθώς αυξάνεται ο αριθμός των επεξεργαστών.

Όσο αφορά την κλιμακωσιμότητα, τα συστήματα χωρίζονται σε δύο μεγάλες κατηγορίες. Σε συστήματα με κατανεμημένη μνήμη και σε συστήματα με κοινή μνήμη.

Τα συστήματα κατανεμημένης μνήμης έχουν μεγαλύτερη κλιμακωσιμότητα. Αυτό το πετυχαίνουν καταργώντας το δίαυλο επικοινωνίας (bus) και εντάσσοντας ένα δίκτυο διασύνδεσης. Κάθε επεξεργαστής μπορεί να έχει δική του ξεχωριστή μνήμη. Το δίκτυο διασύνδεσης δίνει μεγαλύτερο εύρος (bandwidth) και ταχύτητα επικοινωνίας επιτρέποντας σε εκατοντάδες χιλιάδες επεξεργαστές να επικοινωνούν. Ωστόσο αυτό συνεπάγεται δυσκολότερο προγραμματισμό, πολυπλοκότερη επικοινωνία, και φυσικά μεγαλύτερο κόστος για την υλοποίηση του. Ένα διαδεδομένο εργαλείο για προγραμματισμό συστημάτων κατανεμημένης μνήμης είναι το MPI. Οι διεργασίες χρησιμοποιούν μηνύματα για να επικοινωνούν και να στέλνουν δεδομένα η μια στην άλλη.

Τα συστήματα κοινής μνήμης είναι ευκολότερα τόσο στην υλοποίηση όσο και στο προγραμματισμό τους. Ωστόσο έχουν περιορισμένη κλιμακωσιμότητα και στην πράξη δεν ξεπερνούν τις μερικές δεκάδες επεξεργαστών (προσωπικοί υπολογιστές). Ένα γνωστό εργαλείο για παράλληλο προγραμματισμό σε συστήματα κοινής μνήμης είναι το OpenMP. Το OpenMP είναι ένα εργαλείο που μπορεί πολύ εύκολα να μετατρέψει ένα σειριακό κώδικα σε παράλληλο.

Ανεξάρτητα όμως από το εργαλείο που χρησιμοποιούμε και το είδος του συστήματος, όταν προγραμματίζουμε σε παράλληλο περιβάλλον πρέπει πάντα να έχουμε υπόψη μας πιθανές συγκρούσεις (conflicts) και χρονικές εξαρτήσεις (race conditions) που ίσως εμφανιστούν μεταξύ των παράλληλων διεργασιών. Για να επιτευχθεί συγχρονισμός μεταξύ διαφορετικών διεργασιών έχουν παρουσιαστεί πολλές διαφορετικές μεθοδολογίες τόσο σε software όσο και σε hardware.

Οι πιο γνωστές λύσεις που προτάθηκαν σε software είναι τα κλειδώματα (locks), και τα barriers.Τα κλειδώματα αν και τα βρίσκουμε σε πολλές διαφορετικές υλοποιήσεις γενικότερα χρησιμοποιούνται για να διασφαλίσουν πως μόνο μια διεργασία θα μπορεί να έχει πρόσβαση σε συγκεκριμένο κομμάτι της μνήμης, το οποίο και ονομάζουμε κρίσιμο τμήμα (critical section), κάθε φορά. Μας εξασφαλίζουν δηλαδή τον αμοιβαίο αποκλεισμό (mutual exclusion). Τα ποιο γνωστά locks είναι τα spinlock και mutex. Τα barriers επίσης ποικίλουν σε είδος. Διασφαλίζουν πως όλες οι διεργασίες μιας εφαρμογής, έχουν φτάσει ή ολοκληρώσει συγκεκριμένες εργασίες προτού συνεχίσουν με τις επόμενες εργασίες που απαιτεί ο κώδικας.

Λύσεις συγχρονισμού που έχουν προταθεί στο hardware είναι οι ατομικές εντολές (atomic instructions) και το transactional memory. Οι ατομικές εντολές είναι εντολές που χρησιμοποιούν ειδικό κομμάτι του hardware για να εξασφαλίσουν πως θα εκτελεστούν ατομικά, δηλαδή χωρίς να μπορεί ενδιάμεσα να παρέμβει άλλη εντολή. Το transactional memory, αν και αρχικά παρουσιάστηκε σε software μορφή (καθώς και υβριδική), έδειξε ενθαρρυντικά αποτελέσματα στην hardware έκδοση του. Η γενικότερη ιδέα πίσω από το transactional memory είναι πως προγραμματιστής θα έχει την δυνατότητα να επισημαίνει ένα

κομμάτι του κώδικα σαν transaction και το hardware αναλαμβάνει να εκτελέσει αυτό το κομμάτι ατομικά. Θεωρητικά ένα τέτοιο πρότυπο προγραμματισμού απλοποιεί την υλοποίηση παράλληλου κώδικα ενώ ταυτόχρονα απαλείφει προβλήματα "deadlock" που παρουσιάζουν τα κλασσικά locks. Τόσο οι ατομικές εντολές όσο και το transactional memory, δεν διασφαλίζουν αμοιβαίο αποκλεισμό. Αντίθετα χρησιμοποιώντας διάφορες μεθόδους ανιχνεύουν αν υπήρξε conflict και ανάλογα αντιδρούν ώστε να διασφαλίσουν την συνάφεια του κώδικα.

1.2 Αντικείμενο διπλωματικής

Η διπλωματική αυτή εστιάζει στο πρόβλημα του συγχρονισμού σε περιβάλλον κοινής μνήμης. Πιο συγκεκριμένα έχει ως στόχο να εξετάσει την επίδοση του hardware transactional memory που προσφέρουν οι Haswell επεξεργαστές της Intel, μέσα από τα transactional synchronization extensions (TSX) και να συγκρίνει την επίδοση αυτού, με τους κλασσικούς τρόπους συγχρονισμού. Το TSX στους Haswell επεξεργαστές προσφέρει δύο διεπαφές (interfaces) στο προγραμματιστή, το Restricted Transactional Memory (RTM) και το Hardware Lock Elision (HLE) και δίνει για πρώτη φορά σε εμπορική μορφή την ευκαιρία για πειραματισμό με το hardware transactional memory. Γενικότερα οι κλασσικοί τρόποι συγχρονισμού (locks) υποφέρουν σε θέματα κλιμακωσιμότητα καθώς περιορίζονται από το διαθέσιμο bandwidth στο bus. Επίσης πολλές φορές τα κλειδώματα μπορεί να είναι ουσιαστικά αχρείαστα καθώς δεν εμφανίζονται αληθινά conflicts στο hardware. Το TSX της Intel υπόσχεται μεγαλύτερη κλιμακωσιμότητα, απλούστερη υλοποίηση παράλληλου κώδικα, καθώς και συχνά καλύτερες επιδόσεις σε σχέση με τα locks.

1.2.1 Συνεισφορά

Η συνεισφορά της διπλωματικής συνοψίζεται ως εξής:

1. Μελέτη και επεξήγηση εργαλείων παράλληλου προγραμματισμού με έμφαση στο TSX.
2. Υλοποίηση παράλληλων αλγορίθμων με την χρήση του TSX, εστιάζοντας στα Red Black Trees.
3. Ανάλυση αποτελεσμάτων των παράλληλων υλοποιήσεων.
4. Αξιολόγηση των αλγορίθμων που υλοποιήθηκαν.
5. Αξιολόγηση και γενικότερα συμπεράσματα για το TSX της Intel.

1.3 Οργάνωση κειμένου

Στο Κεφάλαιο 2 αναλύονται λεπτομερώς τα πιο γνωστά εργαλεία για υλοποίηση παράλληλου κώδικα σε περιβάλλον κοινής μνήμης. Εστιάζουμε στην επεξήγηση γενικότερα του Transactional Memory καθώς και πιο συγκεκριμένα του Hardware Transactional Memory που προσφέρουν οι Haswell επεξεργαστές της Intel. Επίσης γίνεται παρουσίαση του TSX και αναπτύσσεται ο τρόπος χρήσης του. Ακόμα παρουσιάζονται διάφορες υλοποιήσεις TM και σχετικές εργασίες όπου γίνεται χρήση του TM για παραλληλοποίηση διαφόρων αλγορίθμων. Στο Κεφάλαιο 3 γίνεται μια εισαγωγή στα Red Black Trees, αναφέρονται οι προκλήσεις παραλληλοποίησής τους ενώ επίσης γίνεται παρουσίαση τριών διαφορετικών αλγορίθμων παράλληλων Red Black Trees με την χρήση του TSX. Τα αποτελέσματα των αλγορίθμων που υλοποιήθηκαν στο Κεφάλαιο 3 καθώς και σύγκριση τους γίνεται στο Κεφάλαιο 4. Τέλος στο Κεφάλαιο 5 γίνεται μια ανασκόπηση όσο αφορά το TM και πιο συγκεκριμένα το TSX. Παρουσιάζονται τόσο τα πλεονεκτήματα όσο και τα μειονεκτήματα του HTM των Haswell επεξεργαστών.

2

Εργαλεία Παράλληλου Προγραμματισμού Κοινής Μνήμης.

Στο κεφάλαιο αυτό θα παρουσιάσουμε και θα επεξηγήσουμε διάφορα εργαλεία που μας βοηθούν στην υλοποίηση παράλληλου κώδικα και ειδικότερα σε περιβάλλον κοινής μνήμης. Τα πλείστα από τα εργαλεία αυτά χρησιμοποιούνται στους κώδικες που αναπτύχθηκαν για τους σκοπούς αυτής της διπλωματικής και είναι σημαντικό να επεξηγηθούν αναλυτικά στον αναγνώστη.

2.1 Κλειδώματα - Locks

Το πιο γνωστό εργαλείο όσο αφορά τον συγχρονισμό παράλληλων διεργασιών είναι το κλείδωμα με την χρήση κάποιας υλοποίησης lock. Είναι ένας αυστηρός τρόπος συγχρονισμού που εξασφαλίζει αμοιβαίο αποκλεισμό στην κοινή μνήμη. Ωστόσο υποφέρει σε θέματα κλιμακωσιμότητας ενώ σε πολλές περιπτώσεις η χρήση του κάνει το προγραμματισμό περίπλοκο. Η χρήση κλειδωμάτων πάντα κρύβει το κίνδυνο οι διεργασίες/νήματα (threads) να οδηγηθούν σε αδιέξοδο (deadlock). Στη συνέχεια παρουσιάζουμε δύο γνωστές υλοποιήσεις κλειδωμάτων.

2.1.1 Mutex

Το mutex είναι μια γνωστή υλοποίηση lock η οποία, όπως υποδηλώνει και το όνομά της εξασφαλίζει mutual exclusion. Στόχος του είναι να επιτρέπει μόνο σε μια διεργασία να έχει πρόσβαση σε συγκεκριμένο κομμάτι της μνήμης ανά πάσα στιγμή. Γενικότερα δουλεύει με τις βασικές αρχές κλειδώματος. Μια διεργασία περιμένει να πάρει το "κλειδί". Όταν πάρει το "κλειδί" τότε εισέρχεται στο κρίσιμο τμήμα. Αφού εκτελέσει το κρίσιμο τμήμα ελευθερώνει το κλειδί ώστε να μπορέσει στη συνέχεια να το πάρει κάποια άλλη διεργασία.

Ο τρόπος που γίνεται η πιο πάνω διαδικασία είναι η εξής:

```
pthread_mutex_lock(&mtx);
//critical section
pthread_mutex_unlock(&mtx);
```

Η βασική διαφορά του mutex από τις υπόλοιπες υλοποιήσεις για locks είναι πως όταν μια διεργασία προσπαθήσει να πάρει το lock και αποτύχει (κάποια άλλη διεργασία βρίσκεται ήδη στο κρίσιμο τμήμα) τότε η συγκεκριμένη διεργασία "κοιμάται" για ένα χρονικό διάστημα πριν ξαναπροσπαθήσει να πάρει το lock. Αυτό έχει σαν αποτέλεσμα να μην απασχολεί άδικα τον πυρήνα αλλά να δίνει την ευκαιρία σε άλλες διεργασίες να εκτελούν χρήσιμη εργασία όσο το lock δεν είναι διαθέσιμο. Για τον λόγο αυτό το mutex προτιμάται σε desktop εφαρμογές.

2.1.2 Spinlock

Το spinlock είναι επίσης μια υλοποίηση lock. Ο τρόπος που χρησιμοποιείται και λειτουργεί είναι αντίστοιχος με αυτός του mutex:

```
pthread_spinlock_lock(&lock);
//critical section
pthread_spinlock_unlock(&lock);
```

Η διαφορά στο spinlock είναι πως όταν μια διεργασία προσπαθήσει να πάρει το lock και αποτύχει (κάποια άλλη διεργασία βρίσκεται ήδη στο κρίσιμο τμήμα) τότε η συγκεκριμένη διεργασία ξαναπροσπαθεί συνεχώς μέχρι να επιτύχει να πάρει το lock. Αυτό έχει ως αποτέλεσμα να χρησιμοποιεί άσκοπα το πυρήνα και να δημιουργεί αλλεπάλληλες προσβάσεις στην θέση μνήμης που βρίσκεται αποθηκευμένο το αντίστοιχο lock. Ωστόσο προτιμάται σε real-time εφαρμογές ή όταν το critical section έχει πολύ μικρό υπολογιστικό κόστος.

2.2 Transactional Memory (TM)

Μεταβαίνοντας από μονοεπεξεργαστικά συστήματα σε πολυεπεξεργαστικά συστήματα πολύ μεγαλύτερης κλίμακας παρουσιάστηκε η ανάγκη για εύρεση non-blocking μηχανισμών που

θα έδιναν μεγαλύτερη κλιμακωσιμότητα και θα απάλλασσαν το προγραμματιστή από τις δυσκολίες που παρουσιάζει η υλοποίηση παράλληλου κώδικα. Η ανάγκη αυτή οδήγησε στο transactional memory.

Το transactional memory είναι μια μεθοδολογία που εξασφαλίζει ατομική πρόσβαση στην μνήμη και βασίζεται στην αρχή του transaction. Transaction ονομάζουμε μια ομάδα εντολών που θέλουμε να εκτελεστεί ατομικά, δηλαδή σαν μια ενιαία εντολή. Έτσι όλοι οι επεξεργαστές του συστήματός μας θα ενημερωθούν για την εκτέλεση του transaction χωρίς να μπορεί να παρέμβει ενδιάμεσα, εκτέλεση άλλης εντολής. Το transactional memory χρησιμοποιεί transactions για να εγγυηθεί πως οι αλλαγές στη μνήμη, που επιφέρει η ομάδα εντολών που ανήκει στο transaction, θα γίνουν ατομικά.

Η γενικότερη ιδέα στην οποία στηρίζεται το TM είναι πως κατά την εκτέλεση ενός transaction γίνεται η υπόθεση πως δεν χρειάζεται συγχρονισμός. Ταυτόχρονα κατά την εκτέλεσή του ανιχνεύονται πιθανές εξαρτήσεις (conflicts) που μπορεί να προκύψουν λόγω της παράλληλης εκτέλεσης διεργασιών από πολλαπλούς πυρήνες. Ως conflicts χαρακτηρίζουμε δύο περιπτώσεις:

1. Το transaction γράφει σε θέσεις μνήμης στις οποίες άλλες διεργασίες γράφουν ή διαβάζουν.
2. Το transaction διαβάζει θέσεις μνήμης στις οποίες άλλες διεργασίες γράφουν.

Αν δεν παρουσιαστούν conflicts κατά την εκτέλεση του transaction τότε το TM προσπαθεί να οριστικοποιήσει της αλλαγές του transaction γνωστοποιώντας τις αλλαγές αυτές σε όλους τους επεξεργαστές. Η διαδικασία αυτή ονομάζεται **transactional commit**. Διαφορετικά, αν ανιχνευθούν conflicts πρέπει να αναιρεθούν οι αλλαγές που προκάλεσε το transaction επαναφέροντας το σύστημα στην κατάσταση που βρισκόταν πριν το ξεκίνημα της εκτέλεσης του. Τότε λέμε πως οδηγηθήκαμε σε **transactional abort**. Μπορεί εύκολα κανείς να καταλάβει πως η ταχύτητα με την οποία ανιχνεύονται τα conflicts, και εκτελούνται τα aborts ή τα commits καθορίζουν σε μεγάλο βαθμό την επίδοση της εκάστοτε υλοποίησης TM.

Το TM ανάλογα με την υλοποίηση του χωρίζεται σε δύο μεγάλες κατηγορίες. Το Software TM (STM) και το Hardware TM (HTM). Αν και πρώτη φορά, η ιδέα για υποστήριξη transaction μέσα από το hardware, έγινε το 1986 από τον Tom Knight [1], οι αρχικές προσπάθειες έγιναν σε software, ενώ από το 2005 υπήρξε έντονη έρευνα στο τομέα αυτό.

2.2.1 Software Transactional Memory (STM)

Στην κατηγορία του STM ανήκουν οι υλοποιήσεις που στηρίζονται αποκλειστικά στο software και δεν χρησιμοποιούν επιπλέον hardware για την ανίχνευση των conflicts, ή για τους μηχανισμούς abort και commit κατά την εκτέλεση του transaction.

Βασικό πλεονέκτημα του STM είναι πως μπορεί να χρησιμοποιηθεί από οποιοδήποτε επεξεργαστή καθώς δεν απαιτεί συγκεκριμένη υποστήριξη από το υλικό. Ωστόσο οι STM υλοποιήσεις υποφέρουν από το επιπρόσθετο χρόνο που εντάσσει η όλη διαδικασία της εκτέλεσης του transaction. Κατά κανόνα η ανίχνευση των conflicts και η διαδικασία που πρέπει να ακολουθηθεί σε περίπτωση abort είναι πολύ χρονοβόρες στο STM. Αυτό έχει ως αποτέλεσμα η χρήση του STM να επιφέρει αυξημένη απόδοση μόνο σε πολύ συγκεκριμένες περιπτώσεις.

2.2.2 Hardware Transactional Memory (HTM)

Το HTM αναπτύχθηκε σαν ανάγκη βελτίωσης της απόδοσης του STM. Οι HTM υλοποιήσεις χρησιμοποιούν το hardware για να μπορέσουν να εφαρμόσουν το TM μεταφέροντας την δουλειά της ανίχνευσης των conflicts, του abort και του commit από το software στο hardware. Στόχος είναι η μείωση του χρόνου με τον οποίο επιβαρύνεται το σύστημα κατά την εκτέλεση του TM.

Ωστόσο και το HTM συνεχίζει να προσθέτει ένα σημαντικό (αλλά πολύ μικρότερο σε σχέση με το STM) κόστος κατά την εκτέλεση του transaction, ειδικότερα στην περίπτωση αλλεπάλληλων aborts. Το μεγαλύτερο όμως μειονέκτημα του είναι οι περιορισμένοι πόροι του υλικού που μπορεί να οδηγήσουν σε αδυναμία εκτέλεσης του transaction. Επίσης όταν χρησιμοποιούνται επεκτάσεις στο σύνολο εντολών ενός επεξεργαστή για την χρήση του HTM, αυτό συνεπάγεται πως για κάθε διαφορετικό επεξεργαστή που υποστηρίζει διαφορετική υλοποίηση HTM, πρέπει το πρόγραμμα να ξαναγράφεται χρησιμοποιώντας τις επεκτάσεις του εκάστοτε επεξεργαστή. Τέλος είναι αυτονόητο πως ένα πρόγραμμα που αξιοποιεί μια HTM υλοποίηση δεν μπορεί να τρέξει σε υλικό που δεν υποστηρίζει HTM.

2.2.3 Υλοποιήσεις Transactional Memory

Αν και έγιναν πολλές δημοσιεύσεις που πρότειναν συστήματα για υλοποίηση non-blocking υλοποιήσεων μέσω software για ταυτόχρονη πρόσβαση πολλών πυρήνων σε κοινή μνήμη η πρώτη φορά που ορίστηκε επίσημα η έννοια του Software Transactional Memory (STM) ήταν το 1995 από τους N. Shavit και D. Touitou [2].

Οι V. J. Marathe, W. N. S. III και M. L. Scott [3] παρουσιάζουν διαφορετικά μοντέλα STM καθώς και τις επιδώσεις τους κάτω από διαφορετικές συνθήκες ενώ αναλύουν και σημαντικούς συμβιβασμούς τους οποίους επιφέρουν τα συστήματα αυτά. Επίσης σε ακόμα μια δημοσίευσή τους οι V. J. Marathe, W. N. S. III και M. L. Scott μαζί με τους M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, [4] εστιάζουν στο κόστος που επιφέρει γενικότερα το STM και πως θα μπορούσε αυτό να μειωθεί.

Το 1993 οι M. Herlily και J. Eliot B. Moss [5] παρουσίασαν πολύ πιο καλά ορισμένα πως θα μπορούσε να υλοποιηθεί μια lock-free υποδομή όπως το transactional memory με υποστήριξη από την αρχιτεκτονική του επεξεργαστή. Επίσης μια ομάδα από το πανεπιστήμιο του Stanford [6] παρουσίασε το Transactional memory Coherence and Consistency (TCC) σύστημα το οποίο με την βοήθεια του υλικού επιτρέπει σε transactions να εκτελούνται ατομικά (atomic commit). Η υλοποίηση αυτή χρησιμοποιεί εξειδικευμένους καταχωρητές που παρακολουθούν την κατάσταση του transaction. Ταυτόχρονα δίνει τη δυνατότητα στον προγραμματιστή να καθορίσει σειρά εκτέλεσης μεταξύ των διαφορετικών transactions που ορίζει στο πρόγραμμά του. Το 2001 οι R. Rajwar και R. Goodman [7] παρουσίασαν για πρώτη φορά το Speculative Lock Elision (SLE) το οποίο μπορεί να υποστηριχτεί χωρίς την ένταξη νέων εντολών στο set του επεξεργαστή. Είναι τελείως διαφανές στο προγραμματιστή και στόχος του είναι να απαλείφει τα locks όταν δεν παρουσιάζονται πραγματικά conflicts στο υλικό.

Η πρώτη hardware υλοποίηση του TM (HTM) παρουσιάστηκε το 2009 από την Sun Microsystems [8]. Ο επεξεργαστής είχε την ονομασία Rock και είχε κυκλοφορήσει σε περιορισμένο αριθμό, στους οποίους είχε πρόσβαση μόνο η ερευνητική κοινότητα. Αργότερα ωστόσο η προσπάθεια αυτή εγκαταλείφθηκε. Σήμερα δύο πολύ σημαντικοί επεξεργαστές που υποστηρίζουν TM είναι, ο Haswell της Intel [9] και ο Power 8 της IBM [10]. Μάλιστα ο Haswell με τον οποίο ασχολείται η διπλωματική αυτή είναι ο πρώτος που διαθέτει σε εμπορική μορφή HTM, δίνοντας την δυνατότητα σε οποιοδήποτε χρήστη να μπορεί να έχει πρόσβαση στο εργαλείο αυτό. Ενδιαφέρον έχουν επίσης υβριδικές μορφές μεταξύ STM και HTM που έχουν ως στόχο την επιλογή της καλύτερης προσέγγισης ανάλογα με τις απαιτήσεις του κώδικα [11], [12].

2.3 Intel's Haswell HTM

Όπως προαναφέρθηκε υπάρχουν πολλές υλοποιήσεις TM που διαφέρουν σημαντικά στο τρόπο που διαχειρίζονται τα transactions. Καθώς στόχος αυτής της διπλωματικής εργασίας είναι να εστιάσει στο HTM που προσφέρει η Intel μέσα από τους Haswell επεξεργαστές, είναι σκόπιμο να αναλυθεί σε βάθος ο τρόπος υλοποίησης του. Η επεξήγηση που ακλουθεί βασίζεται στο Intel® Architecture Instruction Set Extensions Programming Reference [13].

Για την αξιοποίηση του HTM οφείλει ο προγραμματιστής να επισημάνει την περιοχή του κώδικα που θέλει να εκτελέσει ως transaction. Καθώς το transaction εκτελείται λέμε ότι βρισκόμαστε σε transactional mode. Καθώς λοιπόν η διεργασία μας βρίσκεται σε transactional mode όλες οι θέσεις μνήμης στις οποίες ζητά πρόσβαση, μεταφέρονται στην L1 cache και ταυτόχρονα κατηγοριοποιούνται σε write και read set ως εξής:

1. **Write set** είναι όλες οι θέσεις μνήμης στις οποίες γράφει το transaction.

2. **Read set** είναι όλες οι θέσεις μνήμης τις οποίες διαβάζει το transaction.

Καθώς εκτελείται αυτή η διαδικασία ταυτόχρονα ανιχνεύονται πιθανά conflicts που μπορεί να προκύψουν από την παράλληλη εκτέλεση διεργασιών. Ο τρόπος που γίνεται η ανίχνευση των conflicts είναι με την χρήση του πρωτοκόλλου συνάφειας μνήμης (π.χ. MOESI). Εν συντομία όταν μια διεργασία, που τρέχει σε ένα πυρήνα διαβάσει ή γράψει σε μια θέση μνήμης, το πρωτόκολλο συνάφειας είναι υπεύθυνο για να ενημερώνει τους υπόλοιπους πυρήνες, που έχουν ήδη φορτωμένη στην cache αυτή τη θέση μνήμης, για την αλλαγή που προέκυψε. Καθώς κατά την εκτέλεση του transaction μεταφέρθηκαν όλες οι κρίσιμες θέσεις μνήμη στην cache, για οποιαδήποτε αλλαγή που θα προκύψει λόγω πρόσβασης στις θέσεις αυτές από άλλο πυρήνα θα μας ενημερώσει άμεσα το πρωτόκολλο συνάφειας. Έτσι χωρίς να χρειάζεται επιπρόσθετη επικοινωνία μεταξύ των διεργασιών που τρέχουν παράλληλα, κάτι που θα περιόριζε το διαθέσιμο εύρος του διαύλου επικοινωνίας (bus bandwidth), έχουμε την δυνατότητα να ελέγχουμε για πιθανές αναγνώσεις ή εγγραφές στις θέσεις μνήμης που αφορούν το τρέχον transaction.

Αν κατά την εκτέλεση του transaction ανιχνευθεί κάποιο conflict, τότε το transaction αποτυγχάνει (transactional abort) και το hardware επιστρέφει στην κατάσταση (state) που βρισκόταν πριν αρχίσει η εκτέλεση του transaction. Αυτό μπορεί να γίνει σχετικά εύκολα καθώς όλες οι εγγραφές που προκαλούσε το transaction γίνονταν απευθείας στη L1 cache και άρα δεν έχει ενημερωθεί για αυτές η κύρια μνήμη του συστήματος. Έτσι, απλά ακυρώνοντας τις τιμές που έχει η cache απορρίπτονται οι αλλαγές και δεν γνωστοποιούνται στο σύστημα. Ωστόσο αν έχει τελειώσει η εκτέλεση του transaction και δεν έχει ανιχνευτεί conflict το hardware προσπαθεί να ενημερώσει το σύστημα για τις αλλαγές που έγιναν (transaction commits) μεταφέροντας τις τιμές που έχει αποθηκευμένες στην cache στην κύρια μνήμη.

2.4 Transactional Synchronizations Extensions (TSX)

Για την αξιοποίηση του HTM που προσφέρουν οι Haswell επεξεργαστές έχει επεκταθεί το σύνολο εντολών της x86 αρχιτεκτονικής με νέες εντολές που δίνουν την δυνατότητα αξιοποίησης και πειραματισμού με το HTM. Επίσης προσφέρονται συναρτήσεις απευθείας στη γλώσσα προγραμματισμού C/C++ για ευκολότερη χρήση του HTM. Οι επεκτάσεις αυτές με την ονομασία Transactional Synchronization Extensions (TSX) προσφέρουν στον προγραμματιστή δύο διεπαφές (interfaces) με τις οποίες μπορεί να εκμεταλλευτεί το HTM. Οι διεπαφές αυτές είναι:

1. Το Hardware Lock Elision (HLE) και
2. Το Restricted Transactional Memory (RTM).

Μια βασική διαφορά μεταξύ HLE και RTM είναι πως αν δοκιμάσουμε να τρέξουμε RTM σε επεξεργαστή που δεν υποστηρίζει TSX τότε το πρόγραμμα θα αποτύχει δίνοντας σχετικό

μήνυμα. Αντίθετα το HLE απλώς θα παρακαμφθεί. Επίσης για τη χρήση του TSX μέσω της C πρέπει ο προγραμματιστής να έχει στη διάθεσή του compiler gcc-4.8.x (ή πιο σύγχρονο) και να εισάγει στο πρόγραμμά του την βιβλιοθήκη "immintrin.h". Αν διαθέτει πιο παλιά έκδοση του gcc τότε πρέπει να εισάγει στο πρόγραμμά του την βιβλιοθήκη "rtm.h" που μπορεί να βρει στην σελίδα της Intel [14].

2.4.1 HLE

Αν και αυτή η διπλωματική εργασία επικεντρώνεται στην χρήση του RTM, για σκοπούς πληρότητας παρουσιάζουμε και το HLE. Το HLE απαλείφει τα κλειδώματα κάνοντας την υπόθεση ότι δεν χρειάζονται και προσπαθεί να εκτελέσει το κρίσιμο τμήμα σαν transaction με την βοήθεια του HTM. Αν ανιχνευθεί κάποιο conflict, κάνει abort το transaction και επανεκτελεί το κρίσιμο τμήμα, αυτή τη φορά με την χρήση του κλειδώματος. Διαφορετικά κάνει commit τις αλλαγές. Αν και το HLE είναι απλό στη χρήση μας περιορίζει στο ότι ταυτόχρονα πρέπει να χρησιμοποιούμε locks. Ένα παράδειγμα χρήσης του HLE σε C όπως αυτό δίνεται από την Intel [14] είναι το εξής:

```
while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE)
!= 0) {
int val;
/* Wait for lock to become free again before retrying. */
do {
_mm_pause();
/* Abort speculation */
__atomic_load_n(lock, &val, __ATOMIC_CONSUME);
} while (val == 1);
}
```

2.4.2 RTM

Το RTM δίνει ακόμη μεγαλύτερη ελευθερία στο προγραμματιστή ως προς την χρήση του HTM. Γενικότερα ο προγραμματιστής σημειώνει το κομμάτι του κώδικα που θέλει να εκτελεστεί ατομικά με τις εντολές XBEGIN και XEND (assembly). Το XBEGIN υποδεικνύει το ξεκίνημα του transaction και το XEND το τέλος. Επίσης με την εντολή XTEST μπορεί να διαπιστώνει ανά πάσα στιγμή αν βρίσκεται ή όχι σε transactional mode. Ακόμα έχει στην διάθεσή του την εντολή XABORT με την οποία μπορεί ρητά να διακόψει την εκτέλεση του transaction.

Κατά την εκτέλεση του transaction μπορεί να παρουσιαστεί ανάγκη για abort του transaction. Στην περίπτωση αυτή ενημερώνεται κατάλληλα ο καταχωρητής EAX με την αιτία που προκάλεσε το abort. Πιο συγκεκριμένα, διαβάζοντας τον καταχωρητή, ο προγραμματιστής μπορεί να μάθει ποια από τις πιο κάτω αιτίες οδήγησαν το transaction σε abort:

1. Conflict: Παρουσιάστηκε conflict λόγω της παράλληλης εκτέλεσης διεργασιών.
2. Capacity: Κάποιος από τους καταχωρητές του επεξεργαστή οδηγήθηκε σε υπερχείλιση.
3. Explicit: Το transaction διακόπηκε μετά από ρητή εντολή του προγραμματιστή
4. Unknown: Δεν μπόρεσε να γίνει γνωστή η αιτία που οδήγησε το transaction σε abort

Αναλυτικότερα στο Πίνακα 2-1 φαίνονται οι τιμές του EAX καθώς και η σημασία τους

Θέση bit του EAX καταχωρητή	**Σημασία**
0	Είναι στο λογικό '1' αν το abort επήλθε από την εντολή XABORT
1	Αν είναι στο λογικό '1' τότε πιθανόν το transaction να επιτύχει σε επόμενη επαναπροσπάθεια
2	Είναι στο λογικό '1' εάν ένας άλλος λογικός επεξεργαστής ήρθε σε σύγκρουση με διεύθυνση μνήμης που ήταν μέρος του transaction
3	Είναι στο λογικό '1' εάν κάποιος εσωτερικός καταχωρητής υπερχείλισε
4	Είναι στο λογικό '1' εάν εντοπίστηκε κάποιο debug σημείο διακοπής
5	Είναι στο λογικό '1' εάν το abort συνέβηκε κατά την διάρκεια εκτέλεσης ενός φωλιασμένου transaction
23:6	Reserved
31:24	Το argument που δόθηκε με την εντολή XABORT. Έγκυρο μόνο αν το bit 0 είναι στο λογικό '1'

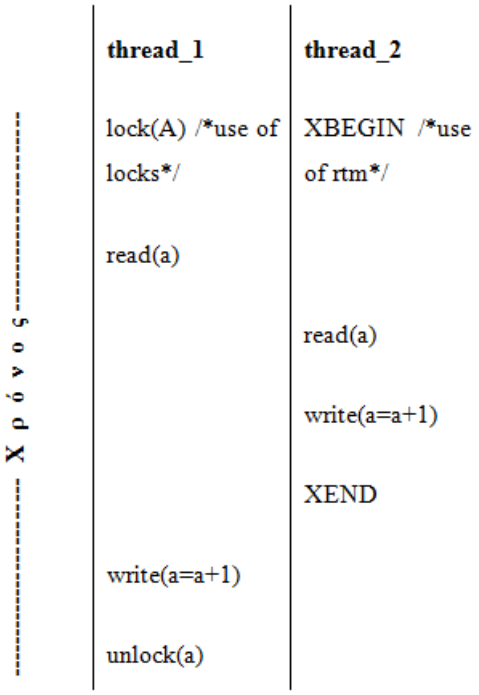
Πίνακας 2-1

Μετά το transactional abort, τη συνέχεια εκτέλεσης του προγράμματος την αναλαμβάνει ο abort handler. Με τον abort handler, ο προγραμματιστής έχει την δυνατότητα να ορίσει αν θέλει να ξαναδοκιμάσει να εκτελέσει τον κώδικά του με την βοήθεια του HTM ή προτιμά να καταφύγει σε κάποιο back-off μηχανισμό. Αυτό που χρειάζεται να επισημάνουμε έντονα είναι πως συγκεκριμένες αιτίες μπορούν να οδηγούν το transaction πάντα σε abort. Αυτό έχει ως αποτέλεσμα αν προσπαθούμε συνεχώς να εκτελέσουμε το κώδικα με την χρήση του HTM, το πρόγραμμά μας να μην τελειώνει ποτέ. Μια τέτοια αιτία είναι το μέγεθος του transaction

να υπερβαίνει το μέγεθος της L1 cache, οπότε και θα οδηγούμαστε πάντοτε σε capacity aborts.

Για να αποφύγουμε το πρόβλημα αυτό, που εντάσσουν οι περιορισμοί του hardware, εισάγουμε την έννοια του fallback-path. Ως fallback-path ορίζουμε μια εναλλακτική υλοποίηση του κώδικά μας η οποία δεν χρησιμοποιεί το RTM. Μετά από ένα αριθμό aborts ο προγραμματιστής μπορεί να επιλέξει την εκτέλεση του fallback-path αντί του transaction. Η υλοποίηση αυτή είναι απαραίτητη για να εγγυηθούμε πρόοδο στην εκτέλεση του προγράμματος.

Ακόμα ένα σημείο που χρειάζεται πολύ προσοχή στο RTM είναι η περίπτωση κατά την οποία το fallback-path που χρησιμοποιείται, έχει υλοποιηθεί με την χρήση locks. Στην περίπτωση αυτή ο προγραμματιστής οφείλει να εισάγει το lock στο read set του transaction. Αυτή η απαίτηση προκύπτει από το γεγονός ότι αν μια διεργασία ξεκινήσει την εκτέλεση του transaction με την χρήση lock και έχει ήδη διαβάσει το write set ενώ ταυτόχρονα μια δεύτερη διεργασία ξεκινήσει και ολοκληρώσει το transaction με χρήση RTM τότε πιθανόν να παραβιαστεί η συνάφεια του κώδικα. Για καλύτερη επεξήγηση στην Εικόνα 2-1 παρουσιάζεται ένα παράδειγμα όπου ταυτόχρονα δύο διεργασίες προσπαθούν να εκτελέσουν τον ίδιο κώδικα, η μια με locks και η άλλη με TSX. Στο παράδειγμα αυτό, το thread_2 δεν θα ανιχνεύσει conflicts και θα κάνει commit κανονικά αλλάζοντας την τιμή του "a", κάτι για το οποίο δεν θα ενημερωθεί ποτέ το thread_1.



Εικόνα 2-1

Για τον λόγο αυτό ο προγραμματιστής οφείλει να εντάξει στο read set του transaction το lock. Επίσης στο ξεκίνημα του RTM πρέπει να ελέγξει αν το lock χρησιμοποιείται από άλλη διεργασία και στην περίπτωση αυτή να δώσει ρητά εντολή για απόρριψη του transaction (explicit abort). Αν και η διαδικασία αυτή ακούγεται απλή περιπλέκει αρκετά το προγραμματισμό. Για να ενταχθεί το lock στο read set, πρέπει να διαβάσουμε την τιμή του χωρίς όμως να κλειδώσουμε. Για να γίνει αυτό πρέπει να μελετήσουμε το αρχείο στο οποίο ορίζεται το lock που χρησιμοποιούμε (π.χ. pthread_mutex_lock.c, pthread_spin_lock.c). Ο τρόπος με το οποίο μπορεί να γίνει αυτό για το mutex και το spinlock είναι ο παρακάτω:

```
if ((int)spin_lock != 1) _xabort();
if (pthread_mutex_t.__data.__lock != 0) _xabort ();
```

Ένας απλός τρόπος χρήσης του RTM σε C που προτείνει στη σελίδα της, η Intel [15] είναι ο εξής:

```
if (_xbegin() == _XBEGIN_START) {
 /* transaction */
 } else {
 /* fallback path -- take lock */
 }
```

2.5 *Εφαρμογές και χρήση του Transactional Memory*

Γενικότερα το TM προσφέρει μια εύρωστη non-blocking υλοποίηση για ταυτόχρονη πρόσβαση πολλαπλών πυρήνων σε κοινή μνήμη. Θεωρητικά προσφέρει τόσο κλιμακωσιμότητα όσο και απλούστερο κώδικα, σε σχέση με μια fine-grained locking υλοποίηση, ενώ υπόσχεται και υψηλές αποδώσεις. Για το λόγω αυτό η ερευνητική κοινότητα προσπάθησε να εκμεταλλευτεί το TM σε παράλληλους αλγορίθμους που μέχρι σήμερα υλοποιούνταν με συμβατικά locks.

Μια τέτοια προσπάθεια παρουσιάζεται από τους K. Nikas, N. Anastopoulos, G. Goumas και N. Koziris [16] όπου υλοποιούν μια παράλληλη έκδοση του Dijkstra αλγορίθμου με χρήση TM με στόχο να αυξήσουν την επίδοσή του.

Επίσης οι D. Dice, Y. Lev και V. J. Marathe [17] με χρήση του HTM που προσφέρει ο Rock επεξεργαστής της Sun επιδεικνύουν πως μπορεί να απλοποιηθεί ο κώδικας υφιστάμενων αλγορίθμων πετυχαίνοντας τις ίδιες ή και καλύτερες επιδώσεις. Πιο συγκεκριμένα εστιάζουν στην υλοποίηση "double ended queues", "work stealing queues" καθώς και άλλων αλγορίθμων πετυχαίνοντας αξιοσημείωτα αποτελέσματα.

Ακόμα οι S. Kang και D. A. Bader [18] παρουσιάζουν ένα αλγόριθμο για υπολογισμό "minimum spanning forest" με τη χρήση του TM. Εστιάζουν στο πως το TM απλοποιεί την υλοποίηση σύνθετων παράλληλων αλγορίθμων και δοκιμάζουν την υλοποίηση τους με την χρήση STM. Καταλήγουν ωστόσο στους περιορισμούς που εισάγει το STM και την αναγκαιότητα για HTM υλοποιήσεις.

Αξιωσημείωτα είναι και τα αποτελέσματα που δίνουν οι D. Dice, Y. Lev, M. Moir and D. Nussbaum [19] στην προσπάθεια τους να αξιωλογήσουν το HTM που προσφέρει ο Rock επεξεργαστής της Sun, εστιάζοντας σε παραλληλοποίηση "hash tables" και "red black trees".

Ακόμα μια χρήση του TM περιγράφεται από τους J. Chung, M. Dalton, H. Kannan και C. Kozyrakis [20] οι οποίοι παρουσιάζουν το Dinamic Binary Translator (DBT) και τη χρήση του για μονονηματικά προγράμματα, ενώ στην συνέχεια με την βοήθεια του TM το επεκτείνουν για πολυνηματικά προγράμματα.

Ιδιαίτερο ενδιαφέρον έχει και η προσέγγιση των F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris και M. Valero [21] που εφαρμόζουν το TM σε ένα interactive multiplayer game server. Αποτελεί μια απόπειρα χρήσης του TM σε ένα ρεαλιστικό πρόβλημα μεγάλης κλίμακας ενώ καταλήγει πως η χρήση του TM δεν είναι η καταλληλότερη για όλα τα προβλήματα συγχρονισμού.

Οι M. Mehrara, J. Hao, P.-C. Hsu και S. Mahlke [22] παρουσιάζουν την υλοποίηση ενός STM μικρού κόστους ενώ στην συνέχεια εστιάζουν στην απόδοση του συστήματος αυτού. Για να εξετάσουν την απόδοση του συστήματος δοκιμάζουν τόσο διαφορετικά benchmarks όσο και σειριακά προγράμματα που τα παραλληλοποιούν με την βοήθεια του TM.

Τέλος μια ιδιαίτερη σκοπιά του TM παρουσιάζεται από τους T. Moreshet, R. I. Bahar και M. Herlihy [23] όπου και αναλύεται πως το TM μπορεί να βοηθήσει ένα πολυεπεξεργαστικό σύστημα σε μείωση της κατανάλωση ενέργειας. Αν και τα αποτελέσματα δεν είναι πάντα θετικά για το TM, δίχνει πως σε περιπτώσεις με μικρό αριθμό από conflicts μπορεί να οδηγήσει σε μείωση της ενέργειας και ταυτόχρονη αύξηση της επίδοσης του συστήματος.

Η σελίδα αυτή είναι σκόπιμα λευκή.

3

Παράλληλη Υλοποίηση Red Black Tree με χρήση TSX

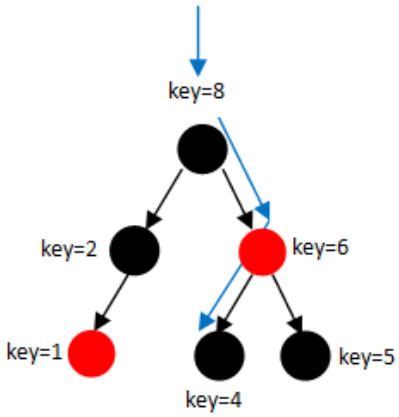
Στο κεφάλαιο αυτό θα παρουσιάσουμε τρεις διαφορετικές υλοποιήσεις Red Black Tree (RBT) οι οποίες θα μας βοηθήσουν όχι μόνο να αξιολογήσουμε το TSX αλλά και να δούμε προβλήματα που εγείρει η χρήση του σε σύνθετες δομές όπως αυτή του RBT. Ωστόσο αρχικά θα ήταν σκόπιμο να παρουσιάσουμε αναλυτικότερα την δομή αυτή και να αναλύσουμε τα προβλήματα που παρουσιάζονται στην προσπάθεια παραληλλοποίησής της.

3.1 Παρουσίαση και επεξήγηση των Red Black Trees

Το RBT είναι μια αρκετά διαδεδομένη δομή δυαδικού δέντρου, η οποία είναι γνωστή για το πολύ μικρό κόστος αναζήτησης κόμβου που προσφέρει. Πιο συγκεκριμένα τα RBTs έχουν ταχύτητα αναζήτησης κόμβου $O(\log_2 n)$. Για τον λόγο αυτό χρησιμοποιούνται σε περιπτώσεις που θέλουμε να αποθηκεύσουμε μεγάλο όγκο δεδομένων στο οποίο και εκτελούμε συχνές αναζητήσεις. Κάθε κόμβος του δέντρου αντιπροσωπεύει μια καταχώρηση. Την ταχύτητα αυτή αναζήτησης την πετυχαίνουν τα RBT τηρώντας δύο προϋποθέσεις που δεν βρίσκουμε σε άλλες δομές δυαδικών δέντρων.

Πρώτη προϋπόθεση είναι πως κάθε κόμβος ενός RBT έχει ένα μοναδικό αριθμό κλειδιού (key). Επίσης στο δεξιό υποδέντρο κάθε κόμβου μπορούν να υπάρχουν μόνο κόμβοι με μεγαλύτερο κλειδί από τον τρέχοντα κόμβο. Αντίστοιχα στο αριστερό υποδέντρο έχουμε κόμβους με μικρότερα κλειδιά. Έχοντας αυτόν το κανόνα υπόψη μας είναι εύκολο να

αντιληφθούμε πως για την αναζήτηση ενός κόμβου αρκεί απλά να συγκρίνουμε το κλειδί του τρέχοντος κόμβου και του κόμβου που ψάχνουμε και αναλόγως να επιλέγουμε κατεύθυνση αναζήτησης στο δέντρο. Ένα τέτοιο παράδειγμα φαίνεται στην Εικόνα 3-1, όπου με μπλε σημειώνεται το μονοπάτι αναζήτησης για το κόμβο '4'. Ωστόσο η προϋπόθεση αυτή δεν επαρκεί από μόνη της για να πετύχουμε ταχύτητα αναζήτησης $O(\log_2 n)$.



Εικόνα 3-1

Η δεύτερη προϋπόθεση που πρέπει να τηρείται είναι πως ανά πάσα στιγμή πρέπει το RBT να είναι ισοζυγισμένο. Εδώ κρύβεται και όλη η επιτυχία αλλά και η δυσκολία υλοποίησης των RBT. Για να επιτευχθεί αυτή η απαίτηση πρέπει μετά από κάθε εισαγωγή ή αφαίρεση ενός κόμβου να ελέγχεται αν το δέντρο είναι ισοζυγισμένο. Αν δεν είναι πλέον ισοζυγισμένο απαιτούνται κάποια επιπλέον βήματα για την διόρθωση του δέντρου.

Το πρώτο ερώτημα που εγείρεται είναι πως μπορεί κανείς εύκολα να ανιχνεύει αν το δέντρο είναι ισοζυγισμένο ή όχι. Ο τρόπος που γίνεται αυτό στα RBTs είναι με την εισαγωγή και τήρηση των πιο κάτω κανόνων:

1. *Κάθε κόμβος είναι κόκκινος ή μαύρος*
2. *Η ρίζα είναι μαύρη*
3. *Όλα τα φύλλα είναι μαύρα*
4. *Κάθε κόκκινος κόμβος πρέπει να έχει δύο μαύρα παιδιά*
5. *Κάθε δυνατό μονοπάτι από ένα συγκεκριμένο κόμβο προς οποιοδήποτε φύλλο πρέπει να περιέχει τον ίδιο αριθμό μαύρων κόμβων*

Πολύ απλά όταν με την εισαγωγή ή αφαίρεση ενός κόμβου σταματήσει να τηρείται οποιοσδήποτε από τους πιο πάνω κανόνες τότε σημαίνει πως το δέντρο δεν είναι πλέον ισοζυγισμένο. Για την περαιτέρω απλούστευση του αλγορίθμου γίνεται η σύμβαση πως κάθε νέος κόμβος που εισάγεται είναι πάντα κόκκινος, ενώ οι κανόνες **(2)** και **(3)** μπορούν εύκολα να παρακαμφθούν.

Εύκολα μπορούμε να διακρίνουμε πως η εισαγωγή ενός κόμβου μπορεί να προκαλέσει παράβαση μόνο του κανόνα **(4)** (*red violation*), ενώ η αφαίρεση ενός κόμβου παράβαση μόνο του κανόνα **(5)** (*black violation*). Διορθώνοντας τις παραβιάσεις στους κανόνες, που μπορεί να προκύψουν μετά την εισαγωγή ή αφαίρεση ενός κόμβου, ταυτόχρονα αναδιατάσσεται το δέντρο για να ξαναέρθει σε ισοζυγισμένη μορφή.

Για την διόρθωση των παραβιάσεων και την ισοζύγιση του δέντρου χρησιμοποιούνται τρεις μηχανισμοί:

1. **Αναχρωματισμοί των κόμβων**
2. **Απλή περιστροφή κόμβων**
3. **Διπλή περιστροφή κόμβων**

Με την χρήση των πιο πάνω μηχανισμών μπορούν να διορθωθούν τοπικά στο δέντρο οι παραβιάσεις. Ωστόσο με κάθε διόρθωση μιας παραβίασης μπορεί να προκληθεί, αλυσιδωτά, παραβίαση σε γειτονικό κόμβο. Για τον λόγο αυτό μπορεί να χρειαστούν πολλαπλές διορθώσεις σε διαφορετικά βάθη στο δέντρο για την πλήρη εξισορρόπηση του δέντρου. Υπάρχουν δύο υλοποιήσεις για την εξυπηρέτηση των παραβιάσεων. Οι υλοποιήσεις αυτές αναλύονται και επεξηγούνται με βάση το [22]:

1. **Μέθοδος $1^{η}$ - Bottom - Up:** Στην υλοποίηση αυτή αρχικά γίνεται αναζήτηση για να βρεθεί το σημείο που πρέπει να γίνει εισαγωγή ή αφαίρεση κόμβου. Στην συνέχεια εκτελείται η αλλαγή και εξετάζεται κατά πόσο προκάλεσε ή όχι κάποια παραβίαση. Αν ναι, τότε εφαρμόζονται οι μηχανισμοί αναδιάταξης που προαναφέραμε. Με κάθε αναδιάταξη μπορεί να προκληθεί επιπλέον παραβίαση μόνο σε κόμβο που βρίσκεται σε μικρότερο βάθος από το τρέχον σημείο. Έτσι ανεβαίνοντας από το σημείο που έγινε η εισαγωγή ή η αφαίρεση και κατευθυνόμενοι προς την ρίζα του δέντρου, αναδιατάσσουμε το δέντρο μέχρι να γίνει πλήρως ισοζυγισμένο και να μην υπάρχουν άλλες παραβιάσεις. Αυτή η υλοποίηση μπορεί να οδηγήσει σε μια διάσχιση ενός μονοπατιού, από την ρίζα μέχρι ένα φύλλο του δέντρου, και στην συνέχεια αλυσιδωτές αναδιατάξεις από το φύλλο πίσω στην ρίζα.
2. **Μέθοδος $2^{η}$ - Top-Down:** Η ιδέα πίσω από την υλοποίηση αυτή είναι πως αν αφαιρούμε πάντα κόκκινο κόμβο ή προσθέτουμε νέους κόμβους μόνο κάτω από μαύρους κόμβους δεν πρόκειται να προκαλέσουμε παραβίαση στους κανόνες του RBT. Έτσι ταυτόχρονα με την αναζήτηση του σημείου που πρέπει να γίνει εισαγωγή ή αφαίρεση κόμβου, γίνονται αναδιατάξεις καθώς διασχίζουμε το δέντρο, ώστε να εξασφαλίσουμε πως την ώρα που θα γίνει η εκάστοτε επεξεργασία δεν θα προκληθεί κάποια παραβίαση. Η υλοποίηση αυτή έχει ως αποτέλεσμα να γίνεται επεξεργασία και αναδιάταξη του δέντρου σε μια διάσχιση (σε αντίθεση με την πρώτη μέθοδο που χρειάζεται δύο διασχίσεις). Ωστόσο πολλές φορές μπορεί να εκτελεστούν αναδιατάξεις που ουσιαστικά να ήταν αχρείαστες, καθώς δεν

μπορούμε να γνωρίζουμε από πριν αν μια εισαγωγή (ή αφαίρεση) κόμβου θα προκαλούσε ή όχι παραβίαση στους κανόνες του RBT.

Πρέπει επίσης να σημειωθεί πως ένα ισοζυγισμένο RBT δεν σημαίνει πως έχει τον ίδιο αριθμό κόμβων σε κάθε υποδέντρο του. Ωστόσο η τήρηση των προαναφερθέντων κανόνων εξασφαλίζει ένα "αρκετά καλά" ισοζυγισμένο δέντρο.

Έχοντας επεξηγήσει την δομή και λειτουργία των RBT εύκολα αντιλαμβάνεται κανείς την δυσκολία παραλληλοποίησής τους. Ο πιο απλός τρόπος για να παραλληλοποιηθεί μια RBT δομή είναι η χρήση ενός κλειδώματος για όλο το δέντρο (coarse grained locking). Αυτό έχει ως αποτέλεσμα οι διεργασίες να σειριοποιούνται. Η χρήση πολλαπλών κλειδωμάτων σε μια δομή RBT μπορεί να προκαλέσει πολλά προβλήματα και δυσκολίες ενώ συνήθως καταλήγει σε deadlock. Για τον λόγο αυτό θα παρουσιάσουμε και θα αναλύσουμε τρεις διαφορετικές υλοποιήσεις με την βοήθεια του TSX με στόχο να απλοποιήσουμε την διαδικασία παραλληλοποίησης του RBT.

3.2 Ευθύς Παραλληλοποίηση Σειριακού Αλγορίθμου Red Black Tree με RTM (RTM-RBT)

Η πρώτη απόπειρα παραλληλοποίησης του RBT έγινε επεκτείνοντας απευθείας το σειριακό κώδικα RBT, με την βοήθεια του TSX και πιο συγκεκριμένα με τη χρήση του RTM. Κάθε πρόσβαση στο δέντρο γίνεται σε ένα transaction εξασφαλίζοντας έτσι τον συγχρονισμό των διεργασιών. Η υλοποίηση αυτή είναι πολύ εύκολη καθώς δεν απαιτεί τίποτα άλλο παρά να επισημάνουμε την αρχή και το τέλος του transaction.

Ουσιαστικά κάθε μια από τις εργασίες που εκτελούνται στο δέντρο (εισαγωγή, αφαίρεση, αναζήτηση) περικλείεται από τις εντολές XBEGIN και XEND. Έτσι το transaction αρχικά ξεκινά με την αναζήτηση του κόμβου που θέλουμε να επεξεργαστούμε. Στην συνέχεια αφού βρίσκει το κόμβο που προτίθεται να επεξεργαστεί, εκτελεί τις απαιτούμενες αλλαγές και αναδιατάσσει το δέντρο αν χρειάζεται. Αφού τελειώσει και η αναδιάταξη του δέντρου τότε τελειώνει και η εκτέλεση του transaction.

Πιο συγκεκριμένα το transaction που θα εκτελεστεί αποτελείται από τα εξής βήματα:

1. Αναζήτηση του σημείου επεξεργασίας
2. Εκτέλεση επεξεργασίας
3. Εκτέλεση αναδιατάξεων (αν απαιτούνται)

3.3 Top-Down Red Black Tree - Window Transactions (WT-RBT)

Η δεύτερη υλοποίηση κώδικα βασίστηκε στην ιδέα των A. Natarajan, L. H. Savoie και N. Mittal [23]. Για τον αλγόριθμο αυτό χρησιμοποιείται Top-Down υλοποίηση Red Black Tree. Η ιδέα που παρουσιάζεται από τους A. Natarajan, L. H. Savoie και N. Mittal [23] είναι πως κάθε διεργασία που θέλει να εκτελέσει εισαγωγή ή αφαίρεση εργάζεται σε ένα παράθυρο από κόμβους. Αρχικά η διεργασία κλειδώνει ατομικά το παράθυρο στο οποίο θα εργαστεί. Στη συνέχεια δημιουργεί ένα αντίγραφο του παραθύρου και εκτελεί τις απαραίτητες αλλαγές πάνω στο αντίγραφο αυτό. Έπειτα ενημερώνει ατομικά το δέντρο για τις αλλαγές που προέκυψαν, αντικαθιστώντας το αρχικό παράθυρο με το αντίγραφο. Η αντικατάσταση αυτή, του αντιγράφου με το πραγματικό παράθυρο, γίνεται με χρήση κάποια ατομικής διαδικασίας, που στην περίπτωσή μας τη προσφέρει το RTM. Αυτή η μεθοδολογία δίνει την δυνατότητα οι αναζητήσεις κόμβων να γίνονται ασύγχρονα.

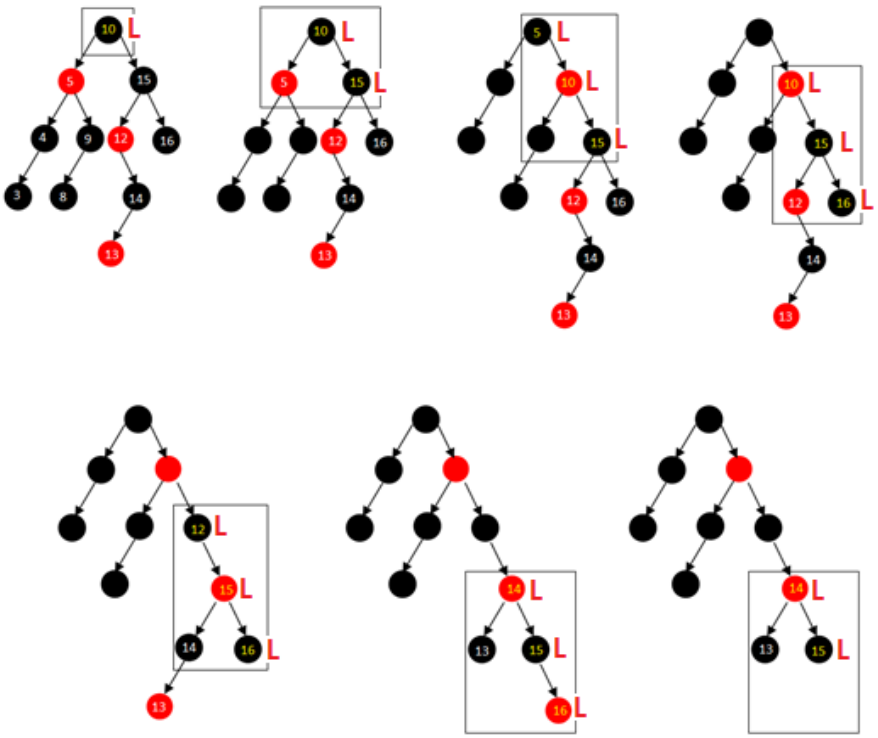
Για να γίνει η υλοποίηση αυτή δυνατή με την χρήση του RTM την τροποποιήσαμε ελαφρώς. Η βασική ιδέα παραμένει η ίδια καθώς κάθε διεργασία "κλειδώνει λογικά" ένα παράθυρο από κόμβους και δουλεύει ατομικά σε αυτούς. Κάθε κόμβος περιέχει μια μεταβλητή lock (integer) η οποίο χρησιμεύει σαν ένδειξη για το αν ο κόμβος αυτός χρησιμοποιείται ή όχι από κάποια διεργασία. Η αλλαγή των μεταβλητών lock γίνεται ατομικά με την χρήση του HTM. Αναλυτικότερα ξεκινώντας από την ρίζα, τα βήματα για να εκτελεστεί μια εργασία στο δέντρο είναι τα εξής:

1. Με την χρήση του HTM ατομικά κλειδώνεται το πρώτο παράθυρο στη ρίζα του δέντρου
2. Εκκίνηση εκτέλεσης transaction
3. Εκτέλεση των απαραίτητων αλλαγών στο παράθυρο
4. Απελευθέρωση του τρέχοντος παραθύρου
5. Μετακίνηση του παραθύρου προς τα κάτω κατά ένα κόμβο
6. Κλείδωμα του καινούριου παραθύρου
7. Τέλος εκτέλεσης του transaction
8. Αν η εργασία έχει φτάσει στο τέλος της τερματίζει. Αλλιώς μεταβαίνει ξανά στο βήμα 2

Για να επιτευχτεί το κλείδωμα του παραθύρου οι διεργασίες κλειδώνουν ένα αριθμό από τους κόμβους του παραθύρου στο οποίο θα εργαστούν. Πιο συγκεκριμένα η διαδικασία αφαίρεσης κόμβου απαιτεί το κλείδωμα τριών συνεχόμενων κόμβων σε κάθε παράθυρο ενώ η διαδικασία εισαγωγής τεσσάρων κόμβων. Αντίθετα η διαδικασία αναζήτησης απαιτεί το

κλείδωμα μόλις δύο κόμβων. Θα μπορούσαμε δηλαδή να πούμε ότι κλειδώνεται μόνο ο "σκελετός" του παραθύρου και όχι όλο το παράθυρο. Λόγω αυτού η διαδικασίες αφαίρεσης, εισαγωγής και αναζήτησης έχουν διαφορετικό μέγεθος παραθύρου το οποίο έχει τέτοιο μέγεθος ώστε να συμπεριλαμβάνει όλους τους κόμβους οι οποίοι θα υποστούν αλλαγές. Πρέπει επίσης να σημειωθεί πως αν μια διεργασία θελήσει να κλειδώσει κάποιο κόμβο που είναι ήδη κλειδωμένος (χρησιμοποιείται από άλλο thread) τότε η διεργασία κάνει abort το transaction του τρέχοντος παραθύρου και ξαναδοκιμάζει. Η αντιγραφή του παραθύρου που προαναφέραμε έχει αντικατασταθεί με την χρήση του RTM που βασικά δημιουργεί αντίγραφα στην cache, των κόμβων που επεξεργάζεται. Ωστόσο αυτή η διαδικασία μας αναγκάζει να χρησιμοποιούμε Window-Transactions και για την αναζήτηση κόμβων σε αντίθεση με τον αρχικό αλγόριθμο που έκανε την αναζήτηση ασύγχρονα. Επίσης αναφέρουμε πως τα παράθυρα της ίδιας διεργασίας αλληλεπικαλύπτονται και μετακινούνται κατά ένα κόμβο κάθε φορά. Παράθυρα διαφορετικών διεργασιών μπορεί επίσης να επικαλύπτονται εν μέρει.

Για την καλύτερη επεξήγηση του αλγορίθμου παρουσιάζεται στην Εικόνα 3-2 η διαδικασία αφαίρεσης ενός κόμβου. Στην εικόνα αυτή παρουσιάζεται η διαδικασία αφαίρεσης του κόμβου '16'. Ξεκινώντας από την ρίζα το παράθυρο καταβαίνει προς τα κάτω αναδιατάσσοντας τους κόμβους ώστε να εξασφαλίσει πως κατά την αφαίρεση του κόμβου



Εικόνα 3-2

'16' δεν θα παρουσιαστούν παραβιάσεις στους κανόνες του RBT. Με το σύμβολο 'L' φαίνονται οι κόμβοι που κλειδώνονται κάθε φορά.

Για την περαιτέρω βελτιστοποίηση του αλγορίθμου κάνουμε την παρατήρηση πως η αναζήτηση ενός κόμβου είναι πολύ πιο γρήγορη διαδικασία από ότι η αφαίρεση ή η εισαγωγή. Για τον λόγο αυτό πριν την εκτέλεση εισαγωγής ή αφαίρεσης, αναζητούμε να δούμε αν ο κόμβος αυτός υπάρχει. Αν θέλουμε να εισάγουμε ένα κόμβο που ήδη υπάρχει δεν χρειάζεται να πράξουμε περαιτέρω. Αντίστοιχα αν θέλουμε να αφαιρέσουμε ένα κόμβο που δεν υπάρχει στο δέντρο μας και πάλι δεν χρειάζεται να κάνουμε κάτι παραπάνω. Επίσης όταν μια διεργασία που εκτελεί αφαίρεση ή πρόσθεση κόμβου διαπιστώσει ότι παρεμποδίζεται συνεχώς λόγω explicit ή conflict aborts, τότε διακόπτει προσωρινά την εκτέλεσή, της δίνοντας χρόνο στην διαδικασία με την οποία έρχεται σε σύγκρουση να προχωρήσει.

3.4 Bottom-Up Red Black Tree - Relaxed Balanced RBT (RB-RBT)

Η τελευταία υλοποίηση είναι βασισμένη στον αλγόριθμο που παρουσιάζουν οι S. Hanke, T. Ottmann και E. Soisalon-Soininen [24]. Μια παρόμοια υλοποίηση χρησιμοποιήθηκε κατά την αξιολόγηση του HTM στο Rock επεξεργαστή από τους D. Dice, Y. Lev, M. Moir and D. Nussbaum [8]. Πρόκειται για Bottom-Up Red Black Tree το οποίο όμως δεν είναι συνεχώς ισοζυγισμένο. Για το λόγω αυτό ονομάζεται "relaxed balanced". Η ιδέα είναι πως οι κόμβοι αφαιρούνται λογικά ενώ επίσης λογικά υποσημειώνονται τα αιτήματα για αναδιάταξη του δέντρου μετά από εισαγωγές ή αφαιρέσεις κόμβων. Επίσης κατά την αφαίρεση κόμβου αντικαθίσταται αρχικά ο προς αφαίρεση κόμβος, με το κατάλληλο κόμβο ώστε το δέντρο να παραμένει συνεκτικό, και στην συνέχεια υποσημειώνεται με αίτημα διαγραφής. Μετά από κάποιο χρονικό διάστημα τα αιτήματα αυτά εξυπηρετούνται και το δέντρο επανέρχεται στην γνωστή RBT μορφή του.

Για τη συγκεκριμένη υλοποίηση, που έγινε για τους σκοπούς αυτής της διπλωματικής εργασίας, τροποποιήσαμε ελαφρώς τον αλγόριθμο που περιγράφεται στο [24] ως εξής:

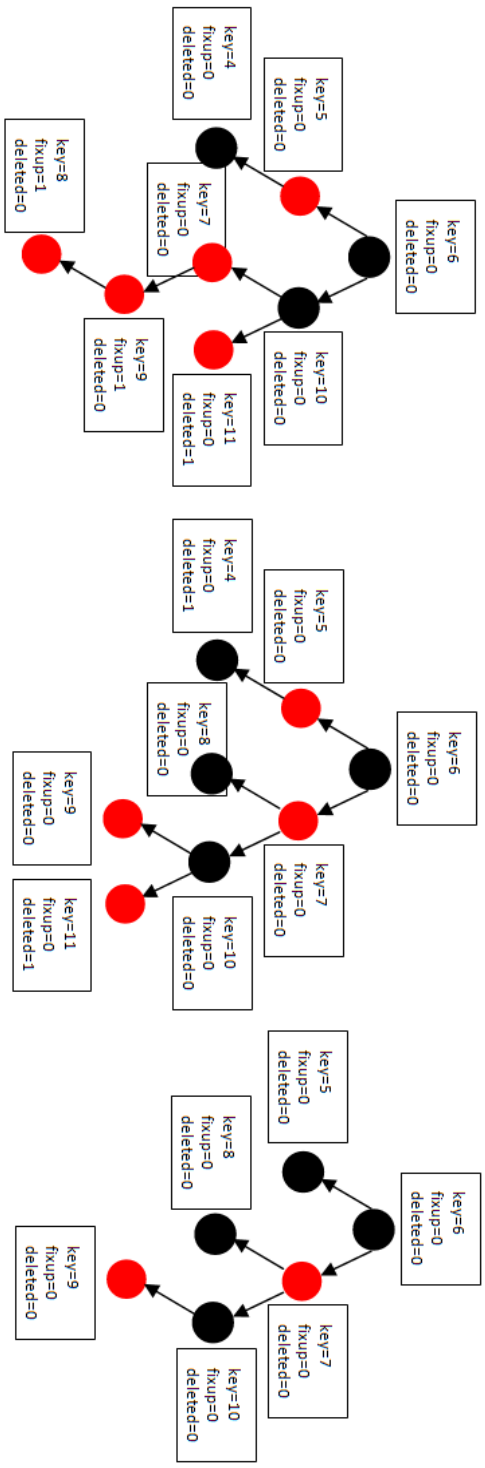
1. Κατά την αφαίρεση κόμβου η διαδικασία σταματά μόλις βρεθεί ο κόμβος που θέλουμε να αφαιρέσουμε και δεν βρίσκουμε τον αντικαταστάτη του. Στη συνέχεια υποσημειώνουμε το κόμβο αυτό με αίτημα διαγραφής. Όταν το αίτημα αυτό εξυπηρετείται, τότε ο κόμβος αυτός αντικαθίσταται κατάλληλα και επίσης εξυπηρετούνται οι αναδιατάξεις που μπορεί να προκαλέσει.
2. Στην περίπτωση της εισαγωγής κόμβου, ο νέος κόμβος εντάσσεται στο δέντρο χωρίς να γίνει όμως αναδιάταξη. Αντίθετα ο νέος κόμβος σημειώνεται με αίτημα για αναδιάταξη το οποίο εξετάζεται σε μεταγενέστερη φάση.

3. Μετά από ένα αριθμό εργασιών στο δέντρο, διακόπτονται προσωρινά όλες οι διεργασίες που εκτελούνται στο δέντρο και σειριακά εξυπηρετούνται τα λογικά αιτήματα διαγραφής και αναδιάταξης που έχουν συσσωρευτεί.

Όταν γίνει αίτημα για εισαγωγή κόμβου που βρίσκεται ήδη στο δέντρο αλλά είναι λογικά διαγραμμένος, τότε απλά απορρίπτεται το αίτημα διαγραφής και ο κόμβος επανεντάσσεται στο δέντρο. Φυσικά η αναζήτηση κόμβων που έχουν διαγραφεί λογικά επιστρέφει αποτυχία εύρεσής τους. Η αναζήτηση κόμβου στο δέντρο γίνεται ασύγχρονα.

Καθώς εντάσσονται νέοι κόμβοι, χωρίς το δέντρο να αναδιατάσσεται, συνεχώς και αυξάνεται ο χρόνος αναζήτησης ενός τυχαίο κόμβου μέσα στο δέντρο. Αυτό συμβαίνει τόσο επειδή δεν έχουν αφαιρεθεί φυσικά οι λογικά διαγραμμένοι κόμβοι αλλά και επειδή το δέντρο παύει να είναι ισοζυγισμένο. Έτσι εγείρεται η ανάγκη για εξυπηρέτηση των αιτημάτων που έχουν μαζευτεί. Αν και οι S. Hanke, T. Ottmann και E. Soisalon-Soininen περιγράφουν το πως θα μπορούσαν τα αιτήματα για αναδιοργάνωση του δέντρου να εκτελούνται παράλληλα, αυτό έχει ως αποτέλεσμα να περιπλέκεται κατά πολλή η διαδικασία. Για τον λόγο αυτό, προτιμήσαμε να γίνεται η αναδιάταξη του δέντρου σειριακά. Η παράλληλη εκτέλεση των αιτημάτων δημιουργούσε πολλά conflicts που έπρεπε να λαμβάνουμε υπόψη μας εγείροντας περαιτέρω προβλήματα συγχρονισμού.

Έτσι μετά από ένα συγκεκριμένο αριθμό εργασιών στο δέντρο μια από τις διεργασίες αναλαμβάνει να εξυπηρετήσει όλα τα αιτήματα σειριακά. Για να γίνει αυτό ορθά πρέπει να εξυπηρετούνται τα αιτήματα αυστηρά με την σειρά που βρίσκονται στο δέντρο, ξεκινώντας από πάνω προς τα κάτω. Για τον λόγο αυτό γίνεται διάσχιση του δέντρου κατά πλάτος, εκτελώντας τις αναδιατάξεις με την σειρά που τις βρίσκουμε και μαζεύοντας σε μια λίστα όλους τους κόμβους με αίτημα διαγραφής. Στην συνέχεια, σειριακά, διαγράφονται φυσικά οι κόμβοι που έχουν μαζευτεί με το αντίστοιχο αίτημα. Φυσικά κατά την αναδιάταξη του δέντρου, μόνο η διεργασία που έχει αναλάβει την συγκεκριμένη δουλειά έχει πρόσβαση στο δέντρο. Η διαδικασία αυτή παρουσιάζεται και σχηματικά στην Εικόνα 3-3 όπου βλέπουμε πως αρχικά εξυπηρετούνται όλα τα αιτήματα αναδιάταξης ενώ στην συνέχεια όλα τα αιτήματα διαγραφής. Στην εικόνα αυτή η μεταβλητή 'fixup' δηλώνει αίτημα αναδιάταξης λόγω προηγούμενης εισαγωγής ενώ η μεταβλητή 'deleted' αίτημα διαγραφής.


key=4
fixup=0
deleted=0
key=5
fixup=0
deleted=0
key=6
fixup=0
deleted=0
key=7
fixup=0
deleted=0
key=8
fixup=1
deleted=0
key=9
fixup=1
deleted=0
key=10
fixup=0
deleted=0
key=11
fixup=0
deleted=1
key=4
fixup=0
deleted=1
key=5
fixup=0
deleted=0
key=6
fixup=0
deleted=0
key=7
fixup=0
deleted=0
key=8
fixup=0
deleted=0
key=9
fixup=0
deleted=0
key=10
fixup=0
deleted=0
key=11
fixup=0
deleted=1
key=5
fixup=0
deleted=0
key=6
fixup=0
deleted=0
key=7
fixup=0
deleted=0
key=8
fixup=0
deleted=0
key=9
fixup=0
deleted=0
key=10
fixup=0
deleted=0

Εικόνα 3-3

Η σελίδα αυτή είναι σκόπιμα λευκή.

4

Πειραματική Αξιολόγηση Παράλληλων Υλοποιήσεων Red Black Tree

Στο κεφάλαιο αυτό θα δούμε αναλυτικότερα τους αλγορίθμους, θα εστιάσουμε σε προβλήματα που προέκυψαν κατά την υλοποίηση τους, θα αναφέρουμε πως αντιμετωπίστηκαν και φυσικά θα παρουσιάσουμε τα αποτελέσματα τους. Στόχος είναι όχι μόνο να δείξουμε ποιος αλγόριθμος και ποια μεθοδολογία είναι καλύτερη αλλά ταυτόχρονα να παρουσιάσουμε την πορεία πειραματισμού που ακολουθήθηκε και τι συμπεράσματα μπορεί αυτή να μας δώσει για την χρήση του TSX.

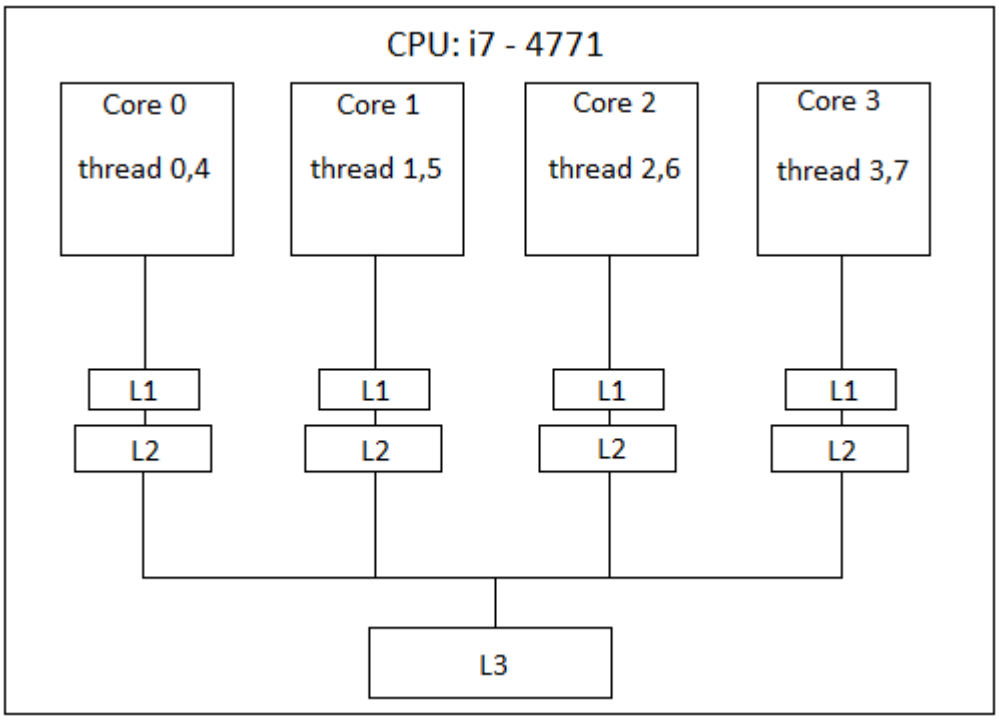
Αρχικά ωστόσο θα αφιερώσουμε λίγο χρόνο για να παρουσιάσουμε τον επεξεργαστή που χρησιμοποιήσαμε για να πάρουμε τα αποτελέσματα που ακολουθούν.

4.1 Περιγραφή Πλατφόρμας και Πειραμάτων

Για την εξαγωγή των μετρήσεων χρησιμοποιήθηκε ο i7-4771 επεξεργαστής της Intel. Ο επεξεργαστής αυτό τρέχει στα 3.5GHz και είναι ένα από τα μοντέλα 4ης γενιάς της Intel που προσφέρουν το TSX. Ο συγκεκριμένος επεξεργαστής αποτελείται από 4 πυρήνες οι οποίοι επίσης διαθέτουν τεχνολογία hyperthreading δίνοντας την δυνατότητα να τρέχουν παράλληλα μέχρι και 8 threads (δύο ανά πυρήνα). Φυσικά πρέπει να έχουμε υπόψη μας πως ανά δύο τα threads μοιράζονται ένα φυσικό πυρήνα και κατ' επέκταση τους πόρους αυτού.

Κάθε φυσικός πυρήνας έχει στην διάθεσή του μια ξεχωριστή L1 και L2 cache, ενώ και οι τέσσερις πυρήνες μοιράζονται μαζί μια L3 cache. Οι L1 και L2 caches έχουν μέγεθος γραμμής 64Byte ενώ διαθέτουν 8-way associativity. Η L1 έχει μέγεθος 32KB, ενώ η L2 256KB. Η L3 cache έχει επίσης μέγεθος γραμμής 64Byte αλλά με 16-way associativity, ενώ το μέγεθος της είναι 8192KB.

Στην Εικόνα 4-1 παρουσιάζεται και σχηματικά η δομή του επεξεργαστή.



Εικόνα 4-1

Όλες οι υλοποιήσεις που προαναφέρθηκαν δοκιμάστηκαν για τρεις διαφορετικούς φόρτους εργασίας. Αρχικά έχουμε μια ελαφριά δοκιμή όπου οι διεργασίες εκτελούν μόλις 2% εισαγωγές και διαγραφές κόμβων ξεχωριστά και 96% αναζητήσεις (2/96/2). Στην συνέχεια οι αλγόριθμου δοκιμάζονται σε ένα μέτριο πείραμα με 20% εισαγωγές και διαγραφές, και 60% αναζητήσεις (20/60/20). Τέλος έχουμε ένα βαρύ φόρτο εργασίας που αποτελείται από 50% εισαγωγές και διαγραφές και 0% αναζητήσεις (50/0/50). Κρατάμε συνεχώς στο ίδιο ποσοστό τις διαγραφές και τις εισαγωγές καθώς δε θέλουμε οι μετρήσεις μας να επηρεάζονται από αυξομειώσεις στο μέγεθος του δέντρου καθώς κάτι τέτοιο θα μας έδινε παραπλανητικά αποτελέσματα.

Επίσης οι πιο πάνω δοκιμές γίνονται για τρεις διαφορετικές αρχικοποιήσεις RBT. Αρχικά έχουμε ένα σχετικά μικρό RBT με μόλις 1000 κόμβους και εύρος κλειδιών από 1-10000. Στην συνέχεια δοκιμάζουμε RBT με 10000 κόμβους και εύρος 1-100000. Τέλος τρέχουμε τις υλοποιήσεις σε ένα μεγάλο RBT με 100000 κόμβους και εύρος 1000000. Είναι σημαντικό να

εξετάσουμε τους αλγορίθμους για μεγάλο αριθμό κόμβων, καθώς τα RBT's προτιμούνται σε τέτοιες περιπτώσεις όπου ο χρόνος αναζήτησης είναι σημαντικός.

Οι μετρικές που χρησιμοποιήθηκαν για την αξιολόγηση των αλγορίθμων ήταν, ο χρόνος εκτέλεσής τους (σε κύκλους), η κλιμακωσιμότητά τους (scalability) καθώς και ο αριθμός των εργασιών που εξυπηρετεί μια διεργασία ανά microsecond (throughput). Επίσης έμφαση δίνεται στον αριθμό των aborts που γίνονται κατά την χρήση του RTM ανά εργασία εκτέλεσης (abort ratio).

4.2 Αξιολόγηση RTM-RBT

Η πρώτη προσπάθεια παραλληλοποίησης του RBT έγινε απλά εντάσσοντας τις εντολές XBEGIN και XEND στην αρχή και στο τέλος κάθε εργασίας. Σε περίπτωση αποτυχίας του transaction ξαναδοκιμάζαμε να το εκτελέσουμε μέχρι να επιτύχει. Αυτό είχε ως αποτέλεσμα η υλοποίηση μας, για μεγέθη δέντρου πάνω από 100 κόμβους, ουσιαστικά να μην τερματίζει ποτέ. Αντίθετα προσπαθούσε επ' άπειρο δίνοντας όλο και περισσότερα conflict και capacity aborts. Η συμπεριφορά αυτή φαινόταν έντονα ακόμα και για πολύ μικρό αριθμό εργασιών όπως παρουσιάζεται και στο Πίνακα 4-1.

Operations per Thread = 1000, Number of Threads =4, Workload= 2% insertions, 96% lookups, 2% deletions	
Number of nodes:	**Cycles**
20	79165864
40	123478072
60	248631548
80	1230347548
100	$> 100^{10}$

Πίνακας 4-1

Το πρόβλημα που παρουσιάστηκε οφειλόταν βασικά σε δύο κύριες αιτίες:

1. Το πολύ μεγάλο μέγεθος του transaction που είχε ως αποτέλεσμα να παρουσιάζονται συχνά conflict aborts.
2. Οι περιορισμένοι πόροι του hardware και πιο συγκεκριμένα το όριο του μεγέθους της L1 cache που μας οδηγούσαν σε πολύ συχνά capacity aborts.

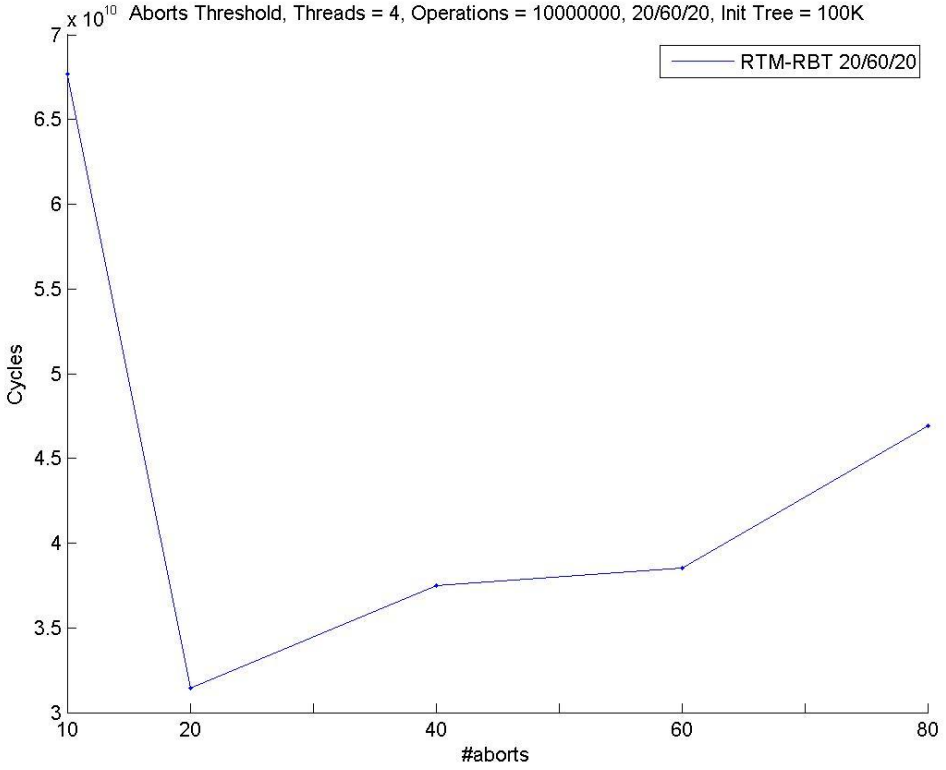
Ο λόγος που γινόταν αυτό είναι επειδή για την εκτέλεση μιας εργασίας στο δέντρο, ένας αρκετά μεγάλος αριθμός από κόμβους εντάσσεται στο transaction. Οι κόμβοι αυτοί είναι όσοι ανήκουν στο μονοπάτι που χρειάστηκε να διασχίσουμε για να βρούμε το σημείο που θα

επεξεργαστούμε, καθώς και όλοι όσοι έπρεπε να συμπεριλάβουμε για τυχόν αναδιατάξεις που προέκυπταν. Πρέπει να σημειώσουμε πως η ρίζα του δέντρου υποχρεωτικά εντάσσεται σε όλα τα transaction και άρα μια αλλαγή στην ρίζα συνεπάγεται αλλεπάλληλα aborts. Είδαμε όμως πως είναι πιθανόν οι αλυσιδωτές αναδιατάξεις που προκαλούνται λόγω διαγραφής και εισαγωγής κόμβων να φτάσουν μέχρι την ρίζα του δέντρου. Επίσης πρέπει να έχουμε υπόψη μας πως για μεγάλα σχετικά δέντρα, η επεξεργασία κόμβων που βρίσκονται σε μεγάλο βάθος στο δέντρο έχει ως αποτέλεσμα να οδηγούμαστε σε συνεχόμενα capacity aborts, λόγω των περιορισμένων πόρων του hardware.

Για να λύσουμε το πρόβλημα αυτό εντάξαμε fallback-path. Το fallback-path που επιλέχτηκε ήταν μια coarse grained υλοποίηση όπου ένα lock κλειδώνει όλο το δέντρο, αναγκάζοντας όλες τις υπόλοιπες διεργασίες να σταματούν την εκτέλεσή τους. Ο λόγος που επιλέχτηκε coarse grained υλοποίηση είναι πως αποτελεί ένα απλό και εύκολο εναλλακτικό τρόπο να εγγυηθούμε πρόοδο στις διεργασίες που τρέχουν. Επίσης θυμίζουμε πως η χρήση πολλαπλών κλειδωμάτων δημιουργεί πολλά προβλήματα στην δομή του RBT. Πιο συγκεκριμένα χρησιμοποιούσαμε το fallback path αν:

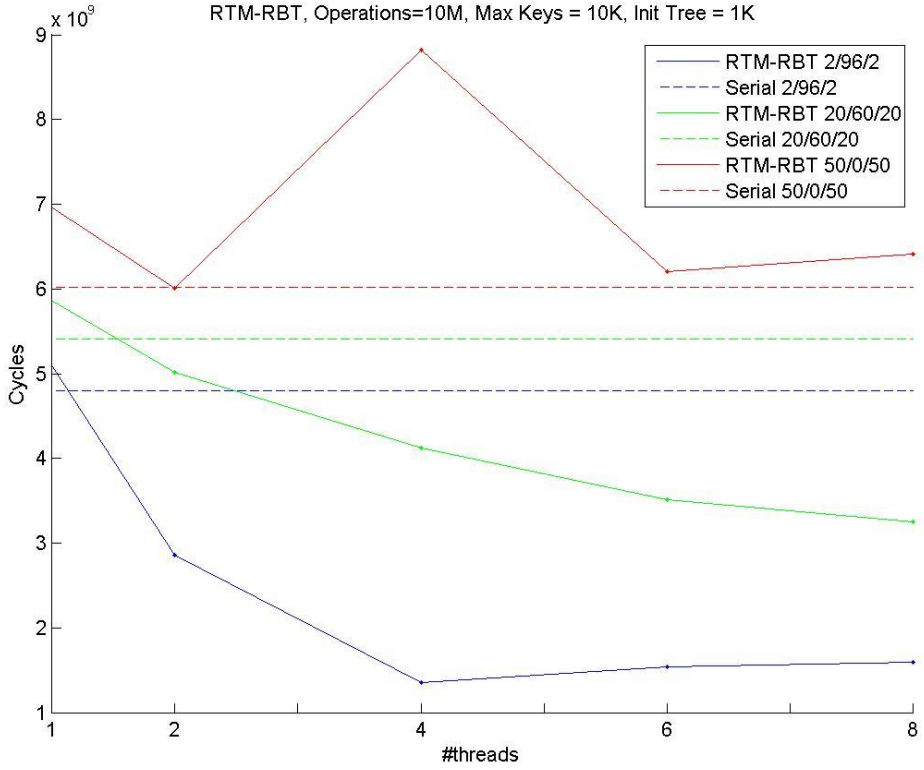
1. Το transaction είχε αποτύχει πάνω από 20 φορές
2. Το transaction οδηγήθηκε σε πάνω από 3 capacity aborts οπότε και δεν είχε νόημα να ξαναπροσπαθήσουμε να το εκτελέσουμε.

Στο σημείο αυτό πρέπει να σημειώσουμε πως ο αριθμός των aborts που επιτρέπουμε να γίνουν, πριν την χρήση του fallback path επιλέχτηκε πειραματικά. Γενικότερα το νούμερο αυτό ήταν διαφορετικό για τις διαφορετικές παραμέτρους του προγράμματος. Ωστόσο επιλέξαμε την τιμή που είδαμε να δίνει γενικότερα καλύτερα αποτελέσματα. Η επιλογή αυτή δικαιολογείται και από την Εικόνα 4-2, όπου για την εκτέλεση 10^7 εργασιών από 4 διεργασίες σε ένα δέντρο με αριθμό αρχικοποιημένων κόμβων ίσο με 10^5, βλέπουμε πως πετυχαίνουμε ελάχιστο χρόνο για επιλογή των 20 aborts ως κατώφλι για την χρήση του fallback path.

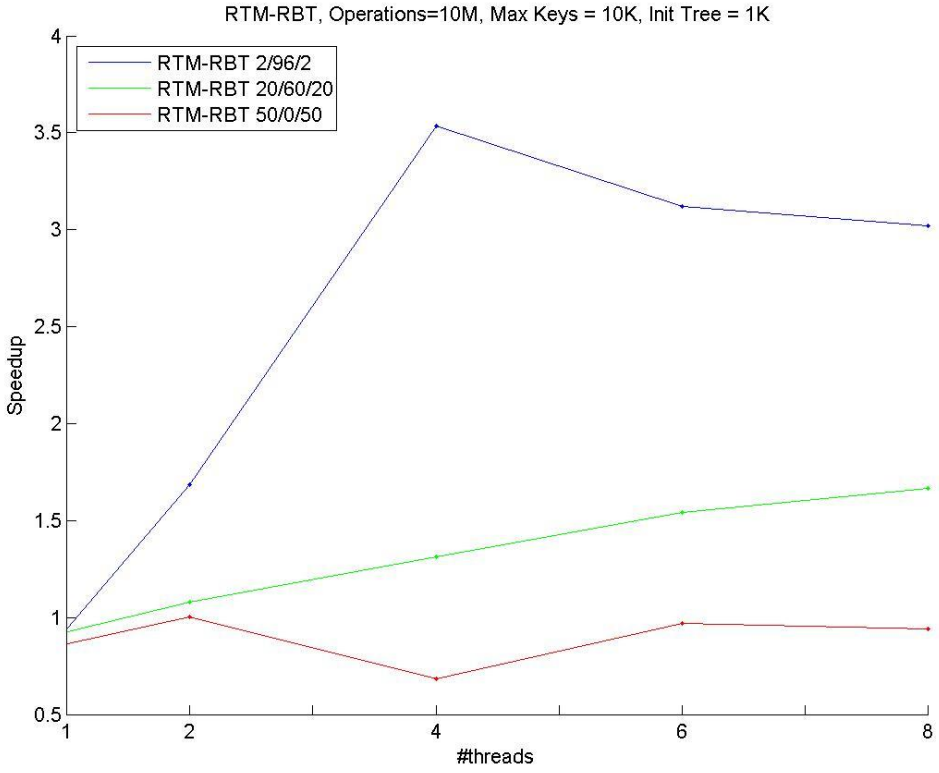


Εικόνα 4-2

Με την εισαγωγή του fallback-path η απόδοση του αλγορίθμου αυξήθηκε θεαματικά. Ειδικά για τις περιπτώσεις με λίγες εισαγωγές και διαγραφές κόμβων, όπου πετυχαίνει να κλιμακώνει σχεδόν 4.5 φορές σε σχέση με τη σειριακή εκτέλεση, όπως παρουσιάζεται και στις Εικόνες 4-3, 4-4. Στην Εικόνα 4-3 βλέπουμε το χρόνο εκτέλεσης της παράλληλης υλοποίησης σε σύγκριση με τη σειριακή για τους τρεις διαφορετικούς φόρτους εργασίας όπως παρουσιάστηκαν στο υποκεφάλαιο 4.1 για αρχικοποιημένο δέντρο 10^3 κόμβων. Στον άξονα 'x' έχουμε πάντα τον αριθμό των διεργασιών που τρέχουν παράλληλα. Αντίστοιχα στην Εικόνα 4-4 βλέπουμε πως κλιμακώνει η περίπτωση αυτή. Μια ακόμα αξιοσημείωτη παρατήρηση είναι πως η συγκεκριμένη υλοποίηση εισάγει ένα πολύ μικρό επιπλέον κόστος (overhead) σε σχέση με την σειριακή. Αυτό φαίνεται παρατηρώντας το χρόνο εκτέλεσής της για μια διεργασία. Ακόμα πρέπει να σημειωθεί πως μετά τις 4 διεργασίες όπου και περνούμε σε hyperthreading ο αλγόριθμος αυτός ουσιαστικά σταματά να κλιμακώνει. Η αιτία είναι πως πλέον τους ίδιους πόρους του hardware τους μοιράζονται παραπάνω διεργασίες ενώ ταυτόχρονα αυξάνεται η πιθανότητα δύο διεργασίες να έρθουν σε σύγκρουση.

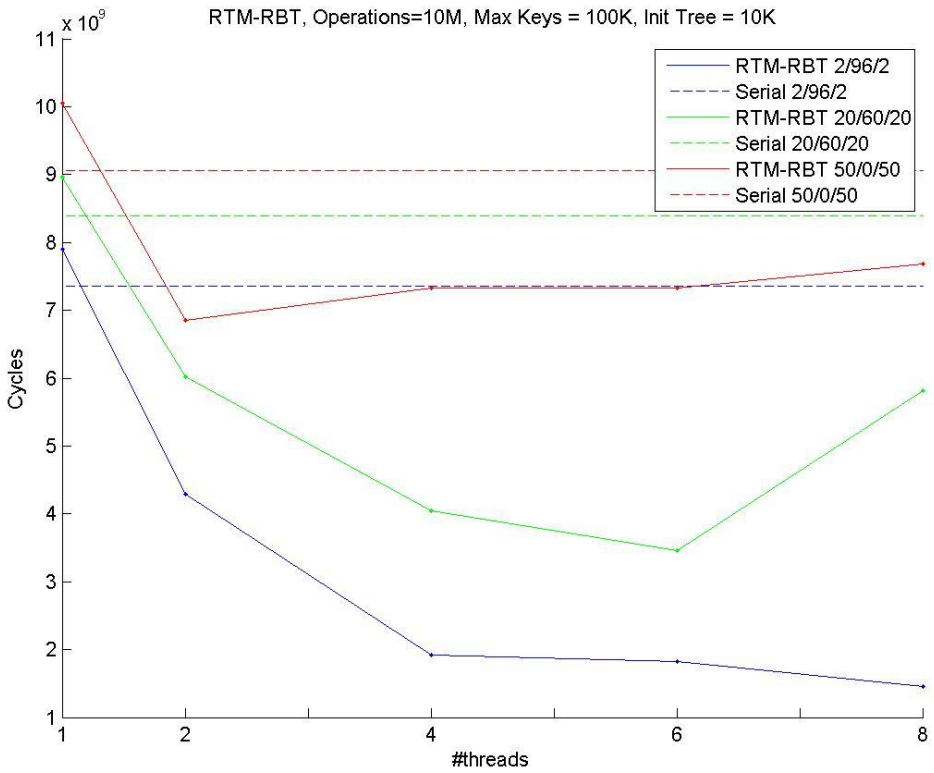


Εικόνα 4-3

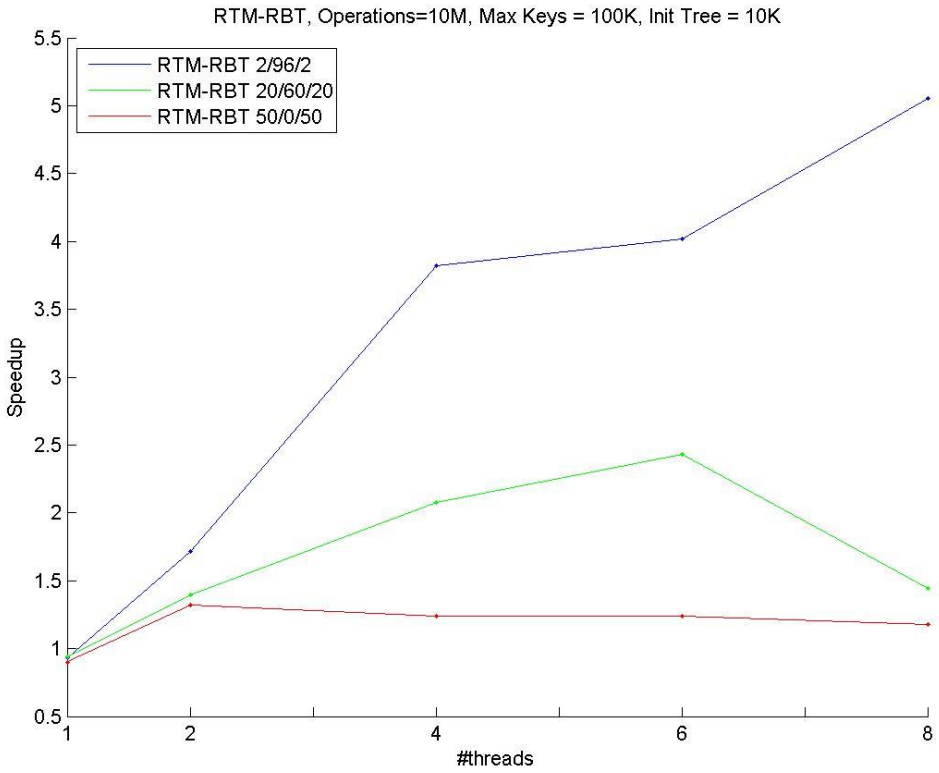


Εικόνα 4-4

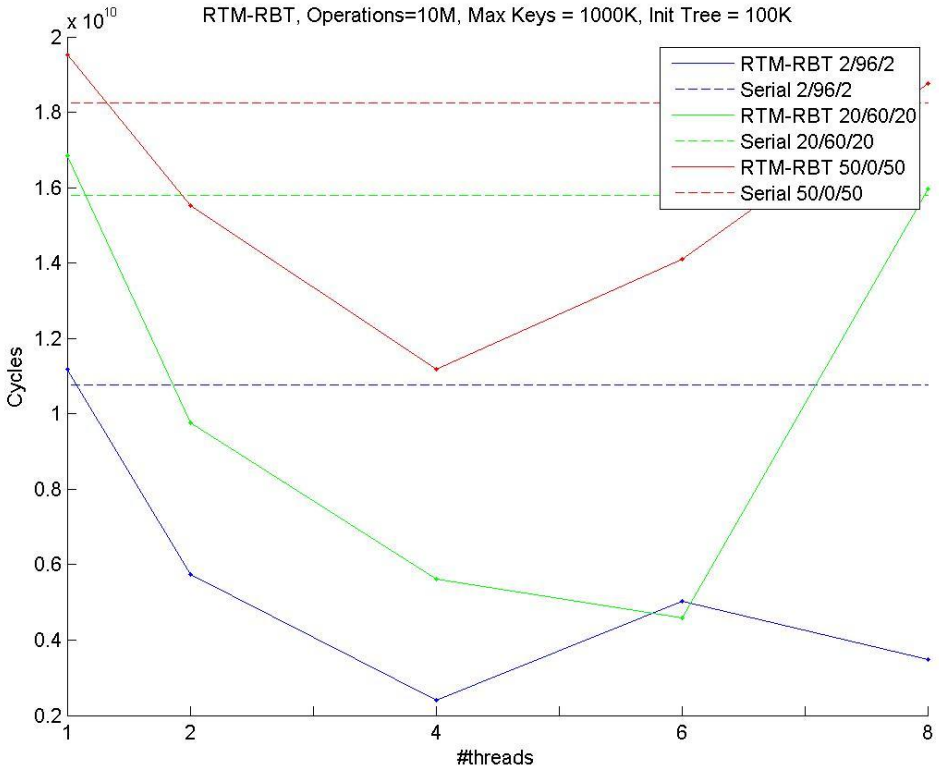
Αυξάνοντας το μέγεθος του αρχικοποιημένου δέντρου, Εικόνες 4-5, 4-6, 4-7, 4-8, παρατηρούμε πως για μικρό αριθμό από διεργασίες παίρνουμε πάλι πολύ καλές αποδόσεις καθώς μεγαλύτερο δέντρο συνεπάγεται και μικρότερη πιθανότητα δύο διεργασίες να εργάζονται στο ίδιο μονοπάτι του δέντρου, προκαλώντας έτσι ανάγκη για συγχρονισμό. Ωστόσο πολύ πιο έντονη είναι πλέον η επίδραση του hyperthreading λόγω του πολύ μεγάλου μεγέθους του transaction.



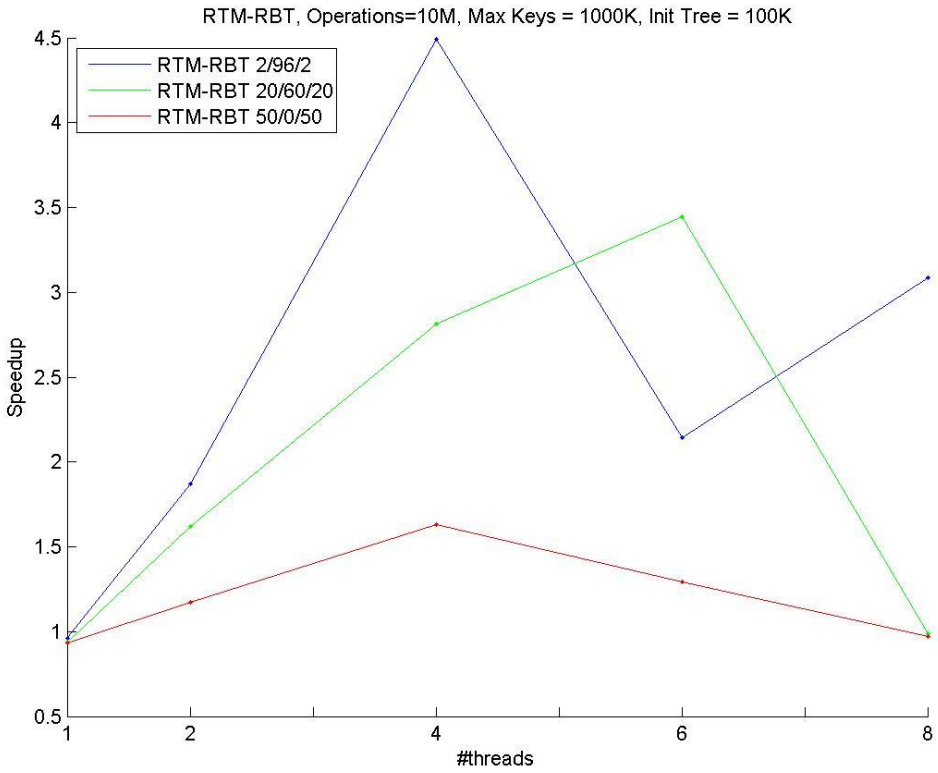
Εικόνα 4-5



Εικόνα 4-6

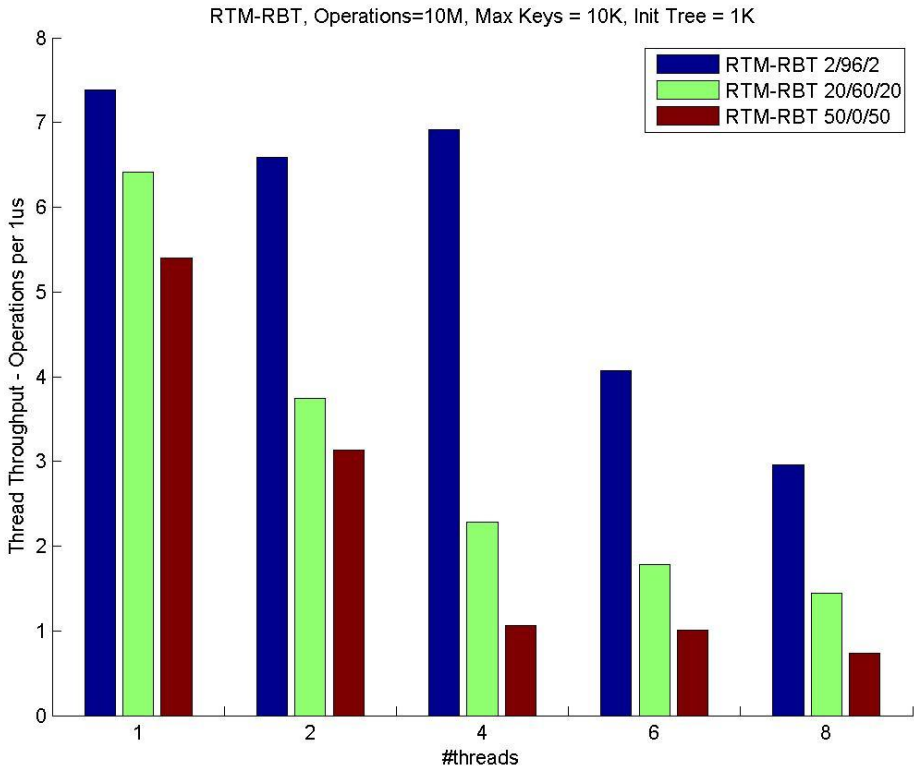


Εικόνα 4-7

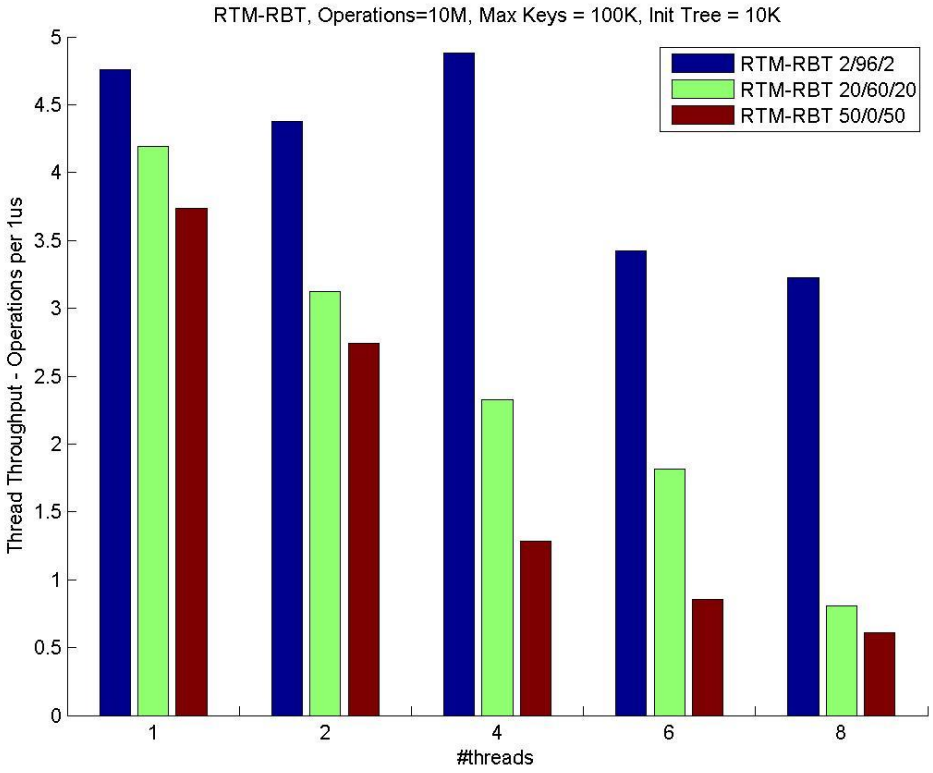


Εικόνα 4-8

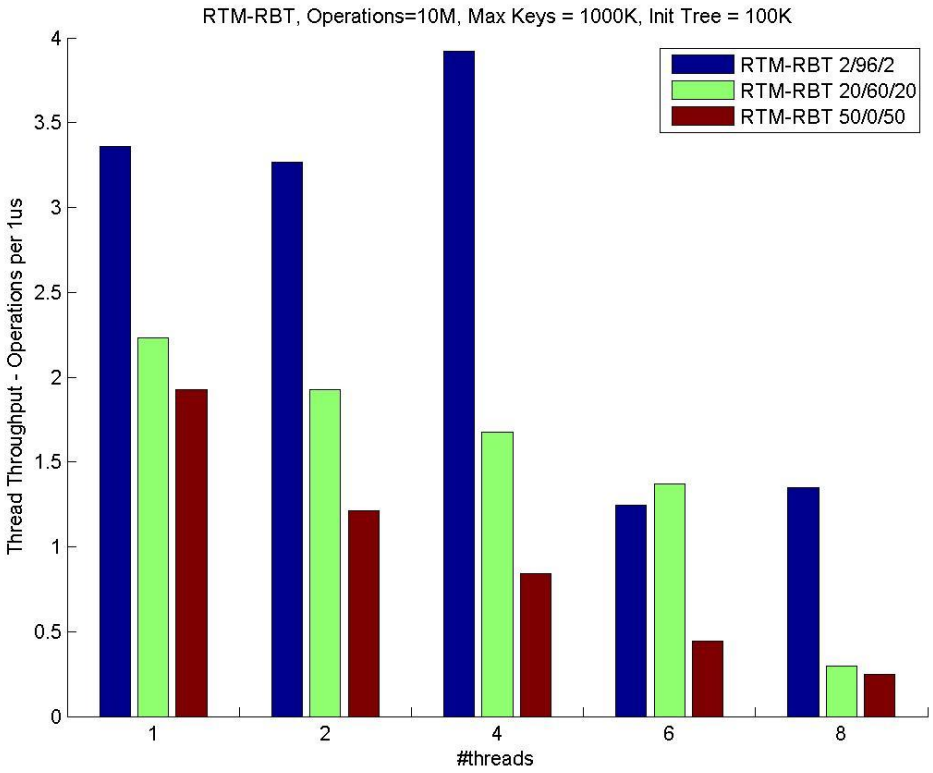
Επίσης ο αλγόριθμος αυτός είχε πολύ κακή συμπεριφορά όσο αφορά το throughput ανά διεργασία. Όπως παρατηρούμε και στις Εικόνες 4-9, 4-10, 4-11 το throughput πέφτει έντονα με αύξηση του αριθμού των διεργασιών που τρέχουν παράλληλα, ενώ ακόμα πιο έντονα μειώνεται στην περίπτωση των 6 και 8 διεργασιών (hyperthreading).



Εικόνα 4-9

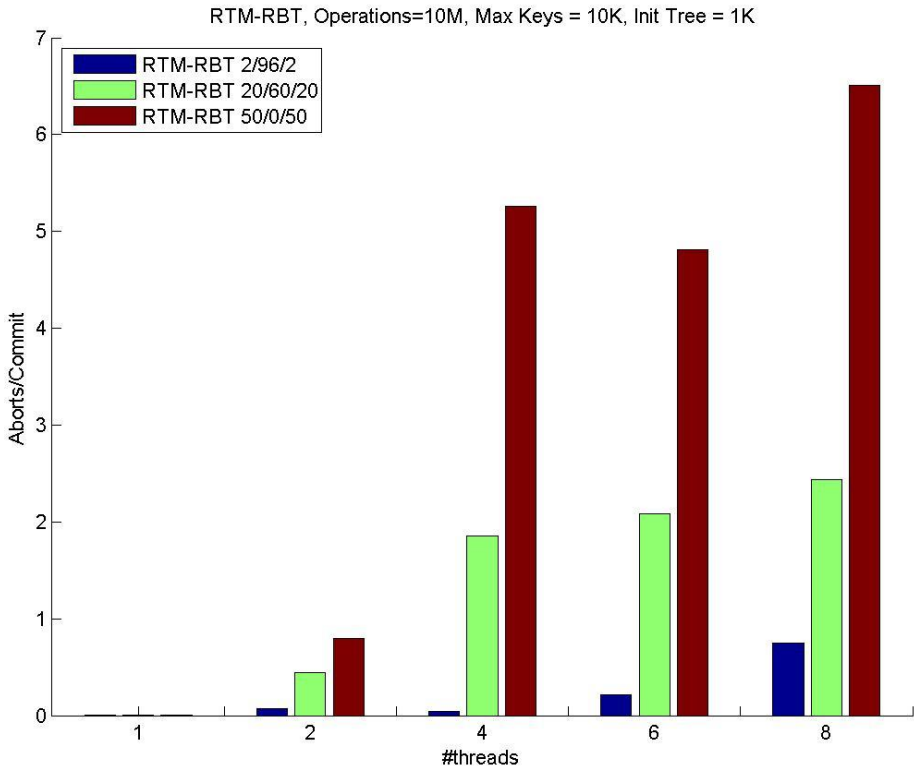


Εικόνα 4-10

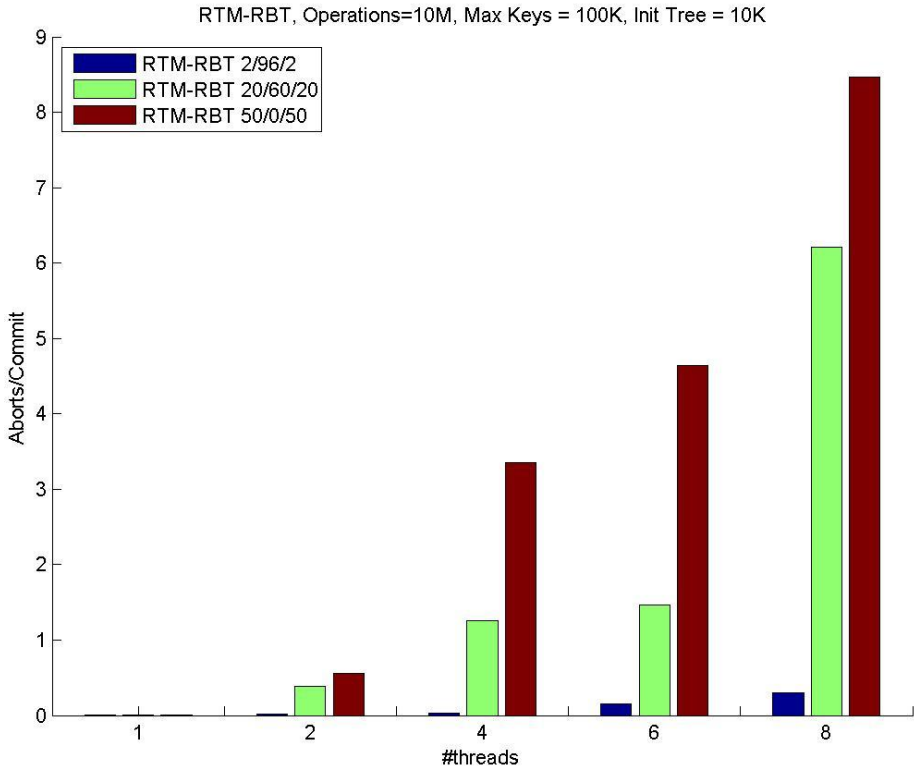


Εικόνα 4-11

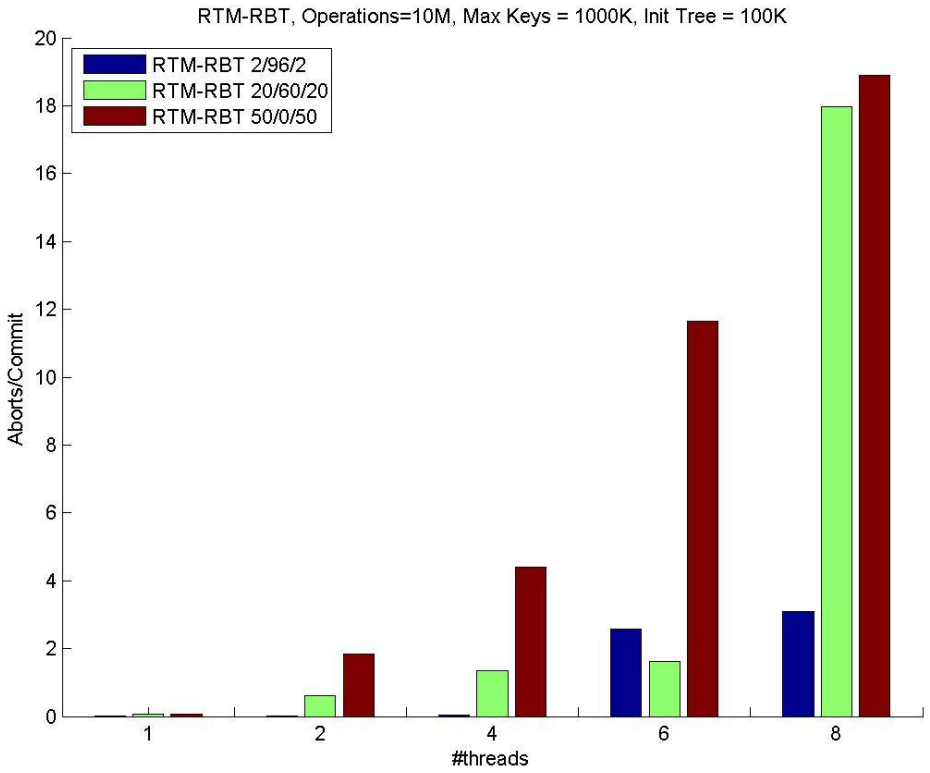
Παρατηρώντας τον λόγο aborts per commit (abort rate), Εικόνες 4-12, 4-13, 4-14, βλέπουμε πως για την περίπτωση του χαμηλού φόρτου εργασίας με μόλις 2% εισαγωγές και διαγραφές, παραμένει μικρός. Αυτό μας δείχνει πως οι περιπτώσεις που οδηγούσαν την αρχική υλοποίηση στο να τρέχει ασταμάτητα, είναι ουσιαστικά μεμονωμένες. Αντιμετωπίζοντας αυτές τι περιπτώσεις με κλείδωμα του δέντρου, καταφέρνουμε να δώσουμε την ευκαιρία στις υπόλοιπες εργασίες να τρέξουν παράλληλα. Επίσης φαίνεται έντονα πως η απόδοση μας περιορίζεται από τον αριθμό των aborts καθώς στις περιπτώσεις που πέφτει η απόδοση του αλγορίθμου μας, τις οποίες επισημάναμε προηγουμένως, έχουμε αντίστοιχη αύξηση του abort rate. Αυτό που αξίζει να επισημάνουμε είναι πως όταν αυξάνεται το μέγεθος του δέντρου έχουμε μεγαλύτερο abort rate. Αυτό συμβαίνει καθώς το μονοπάτι αναζήτησης ενός κόμβου αυξάνει σε μέγεθος κάτι που έχει ως αποτέλεσμα την αύξηση του μεγέθους του transaction και άρα περισσότερα capacity aborts. Επιπλέον πρέπει να σημειώσουμε πως αν και ο αριθμός των aborts ανά διεργασία φαίνεται μικρός στην πραγματικότητα για αυτό οφείλεται το κατώφλι που έχουμε εισάγει για χρήση του fallback path.



Εικόνα 4-12



Εικόνα 4-13



Εικόνα 4-14

Συνοψίζοντας η υλοποίηση αυτή μας προσφέρει ένα πολύ εύκολο και γρήγορο τρόπο να παραλληλοποιήσουμε τα RBTs. Επίσης πετυχαίνει αξιοσημείωτη απόδοση και κλιμακωσιμότητα, ειδικότερα για τέσσερις ή λιγότερες διεργασίες.

Ωστόσο έχει ένα μεγάλο μειονέκτημα. Το transaction έχει μεταβλητό μέγεθος που εξαρτάται από την εργασία που θα εκτελεστεί στο δέντρο, καθώς και από το μέγεθος του δέντρου. Αυτό μπορεί να έχει ως αποτέλεσμα σε ακόμα μεγαλύτερα δέντρα να οδηγούμαστε πάντα στην χρήση του lock. Επιπρόσθετα η υλοποίηση αυτή υποφέρει από μεγάλο αριθμό aborts ενώ μειώνεται πολύ έντονα η απόδοσή της στην περιοχή του hyperthreading. Πολύ κακή είναι και η συμπεριφορά της ως προς το throughput που μειώνεται έντονα με αύξηση του αριθμού των διεργασιών.

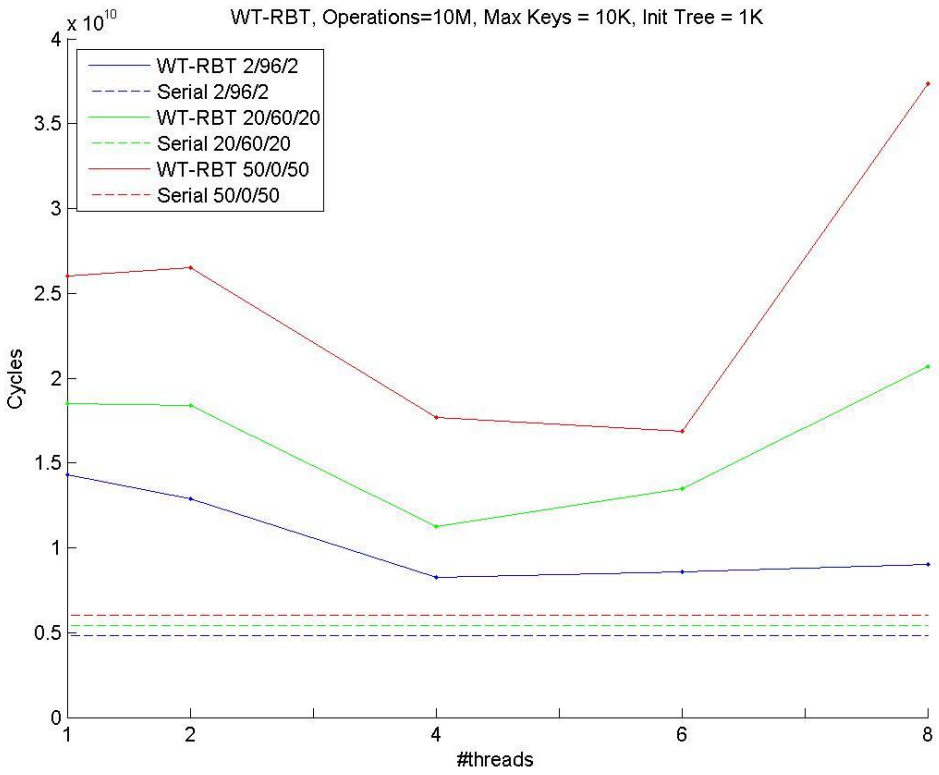
4.3 Πειραματικά Αποτελέσματα: WT-RBT

Η υλοποίηση με Window-Transactions μας δίνει την λύση στο πρόβλημα που αναφέραμε προηγουμένως. Η κάθε εργασία που εκτελείται στο δέντρο σπάζει σε πολλαπλά transactions μικρότερου μεγέθους, με συγκεκριμένο μέγεθος ανά εργασία.

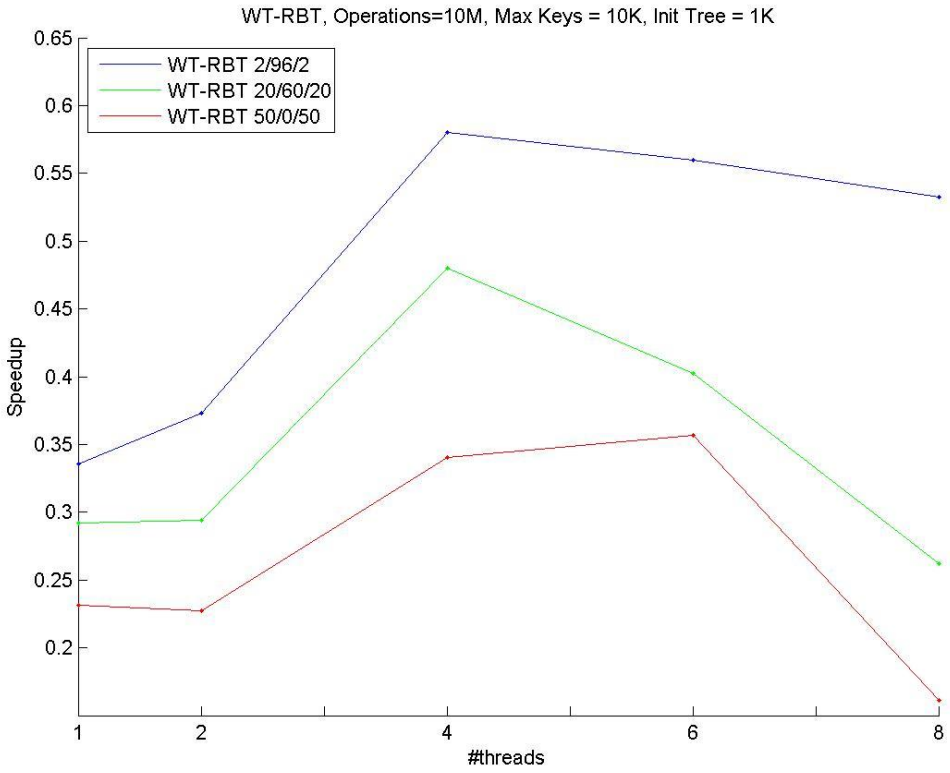
Στην αρχική δοκιμή αυτού του αλγορίθμου κάθε παράθυρο κλείδωνε λογικά όλους τους κόμβους τους οποίους πιθανών να χρειαζόταν να επεξεργαστεί. Θυμίζουμε πως ανάλογα με

το αν θα εκτελεστεί περιστροφή και το είδος της περιστροφής, χρειάζεται να γίνει πρόσβαση σε διαφορετικό αριθμό από κόμβους. Ωστόσο αυτό οδηγούσε σε μεγάλο αριθμό από explicit aborts. Για τον λόγο αυτό μειώσαμε τον αριθμό των κλειδωμάτων κρατώντας μόνο ένα μικρό αριθμό κόμβων λογικά κλειδωμένους σε κάθε παράθυρο όπως επεξηγήθηκε στο 3.3. Την δυνατότητα αυτή μας την δίνει το RTM καθώς σε περίπτωση πραγματικών conflicts οδηγεί τα παράθυρα σε aborts. Έτσι μειώθηκαν σημαντικά τα aborts και αυξήθηκε η επίδοση του αλγορίθμου.

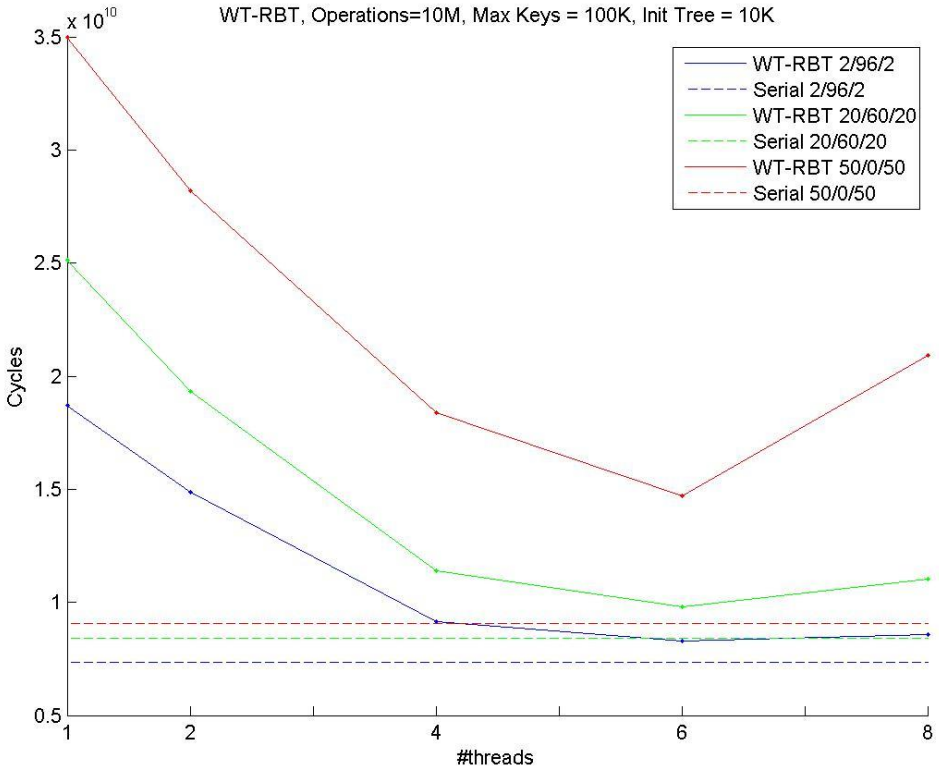
Ωστόσο παρατηρώντας τις Εικόνες 4-15, 4-16, βλέπουμε πως για αρχικοποιημένο δέντρο μόλις 10^3 κόμβων, έχουμε πολύ κακή επίδοση. Το πρόγραμμά μας έχει ένα αρκετά μεγάλο overhead που οφείλεται στο κόστος που εισάγουν τα πολλαπλά transactions που πρέπει να γίνουν για μόνο μια διεργασία. Επίσης δεν καταφέρνει ποτέ να ξεπεράσει το χρόνο της σειριακής εκτέλεσης. Ωστόσο μεταβαίνοντας σε μεγαλύτερο δέντρο των 10^4 κόμβων, Εικόνες 4-17, 4-18, η επίδοση του αλγορίθμου βελτιώνεται αισθητά. Επίσης πολύ έντονα φαίνεται πως με αύξηση των διεργασιών από 6 σε 8 η απόδοση πέφτει δραματικά. Αυτό συμβαίνει καθώς αυξάνεται ο αριθμός των παραθύρων που προσπαθούν να επεξεργαστούν το δέντρο και άρα αυξάνεται η πιθανότητα σύγκρουσης των διεργασιών.



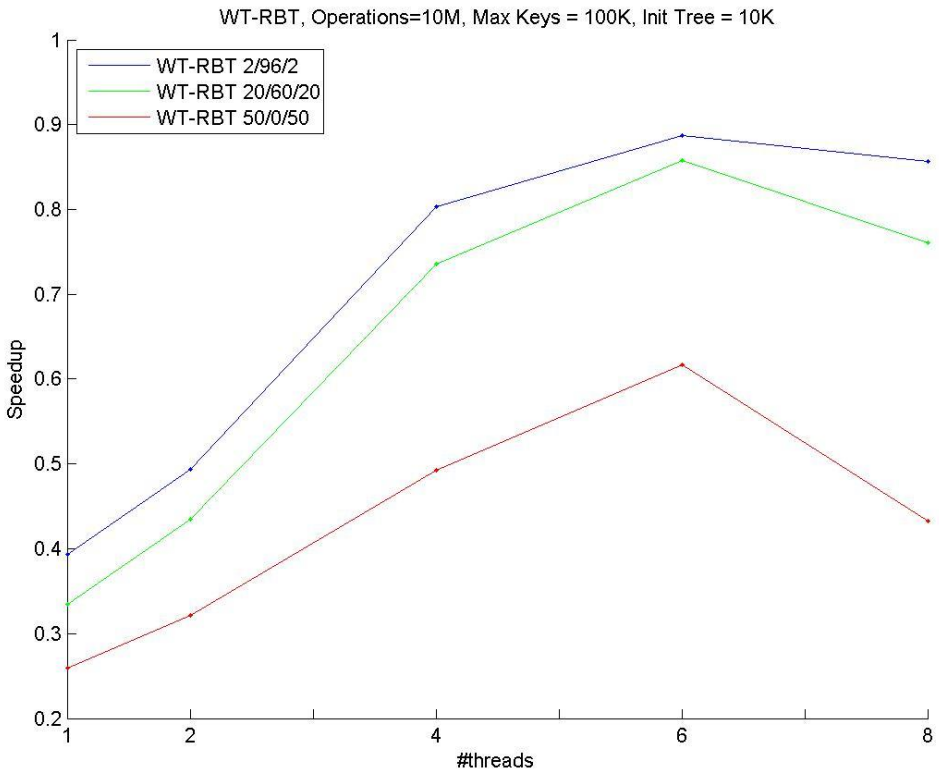
Εικόνα 4-15



Εικόνα 4-16

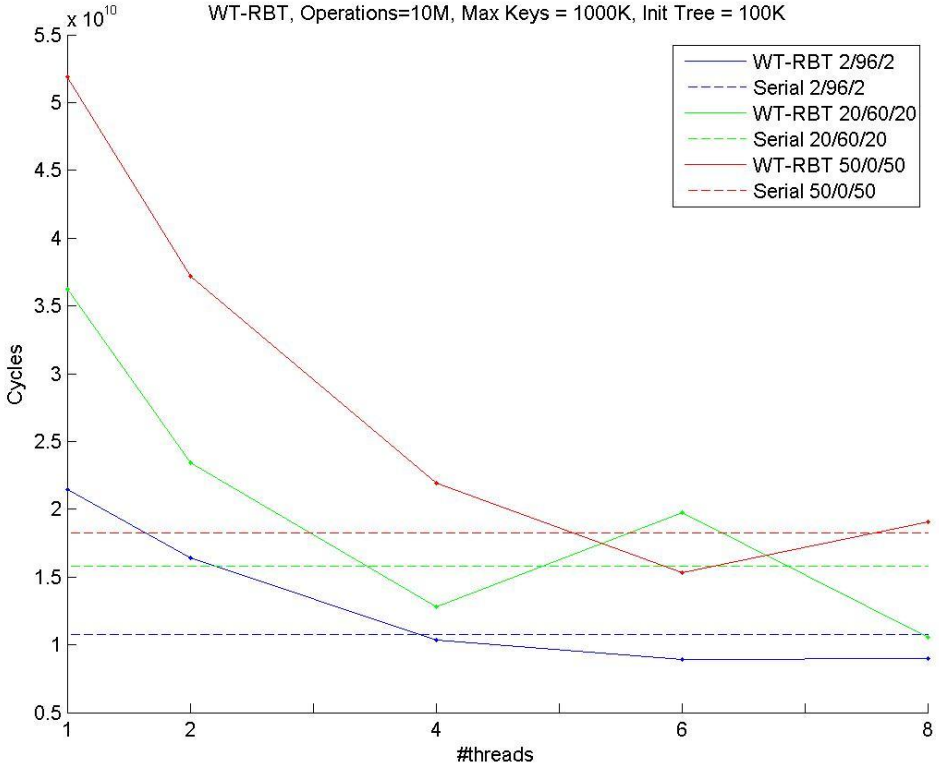


Εικόνα 4-17

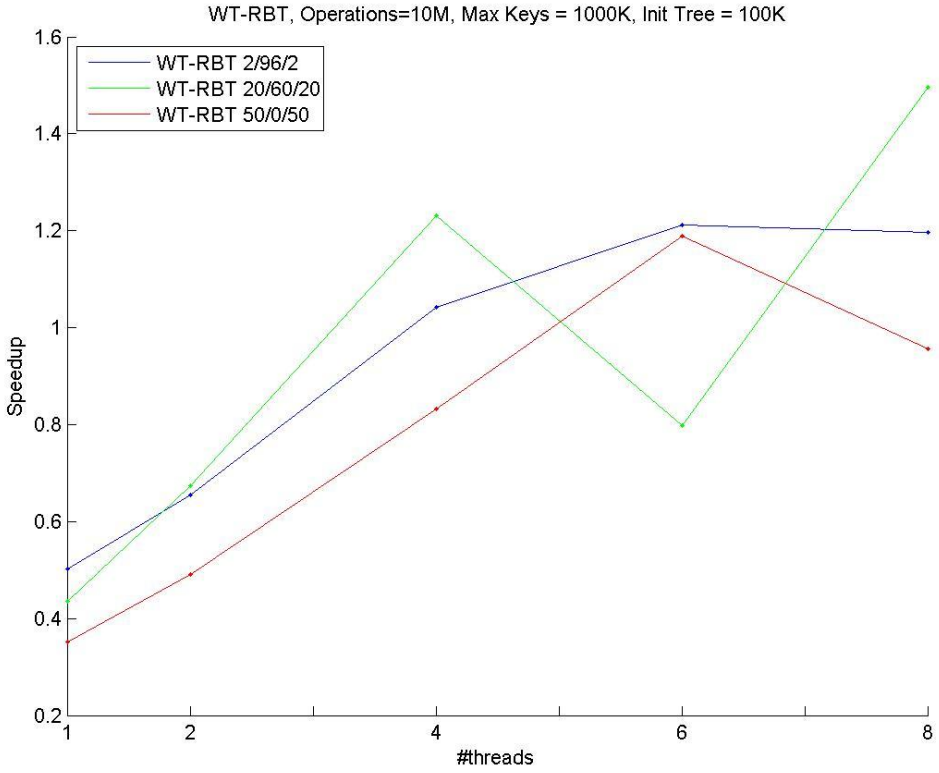


Εικόνα 4-18

Η πρώτη φορά που ο αλγόριθμος καταφέρνει να ξεπεράσει σε απόδοση την σειριακή εκτέλεση είναι για τις δοκιμές σε αρχικοποιημένο δέντρο με 10^5 κόμβους όπως φαίνεται και στις Εικόνες 4-19, 4-20. Τα αποτελέσματα αυτά μας δείχνουν πως ο αλγόριθμος τείνει να δώσει ακόμα καλύτερα αποτελέσματα για μεγαλύτερα δέντρα. Αυτό είναι σημαντικό καθώς γνωρίζουμε πως τα RBT χρησιμοποιούνται για αποθήκευση μεγάλου όγκου δεδομένων. Επίσης παρατηρούμε πως έχει όμοια συμπεριφορά για τα διαφορετικά σύνολα εργασίας. Φυσικά αναμενόμενη ήταν η αύξηση του χρόνου εκτέλεσης για μεγαλύτερο αριθμό εισαγωγών και αφαιρέσεων κόμβων, καθώς οι διαδικασίες αυτές είναι πιο χρονοβόρες ενώ παράλληλα απαιτούν και μεγαλύτερα Window-Transactions από ότι οι αναζητήσεις κόμβων.

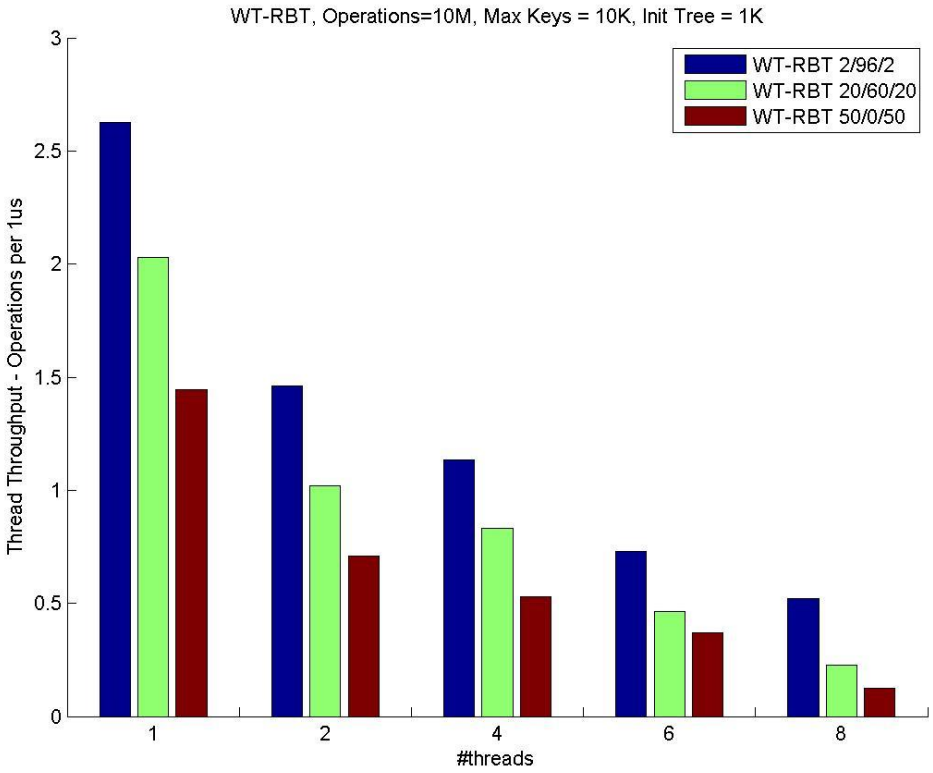


Εικόνα 4-19

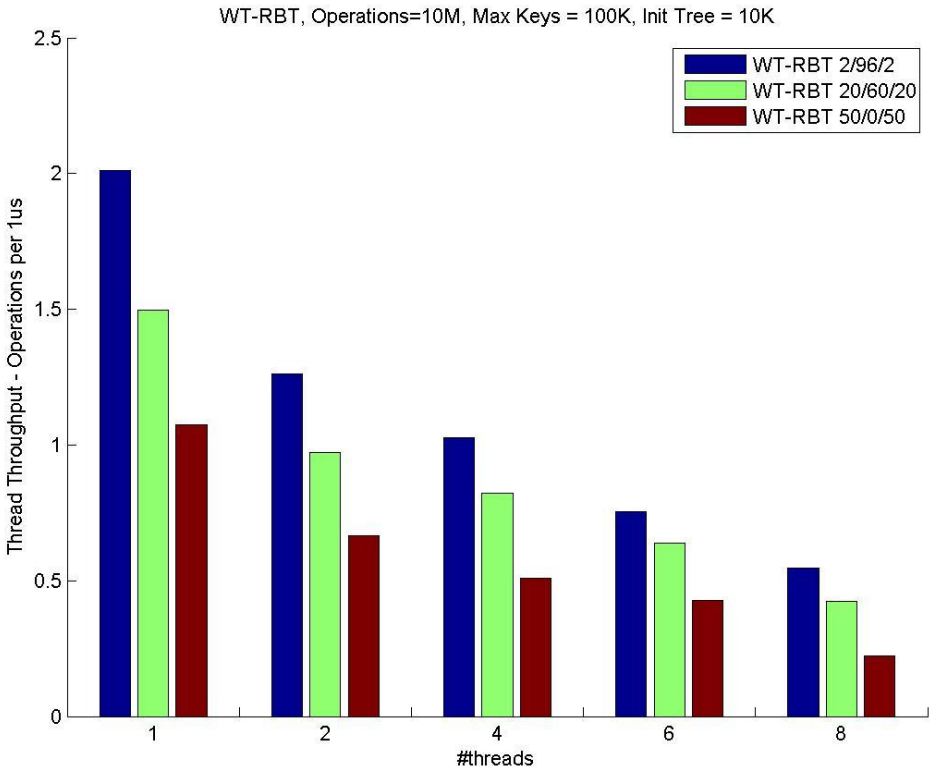


Εικόνα 4-20

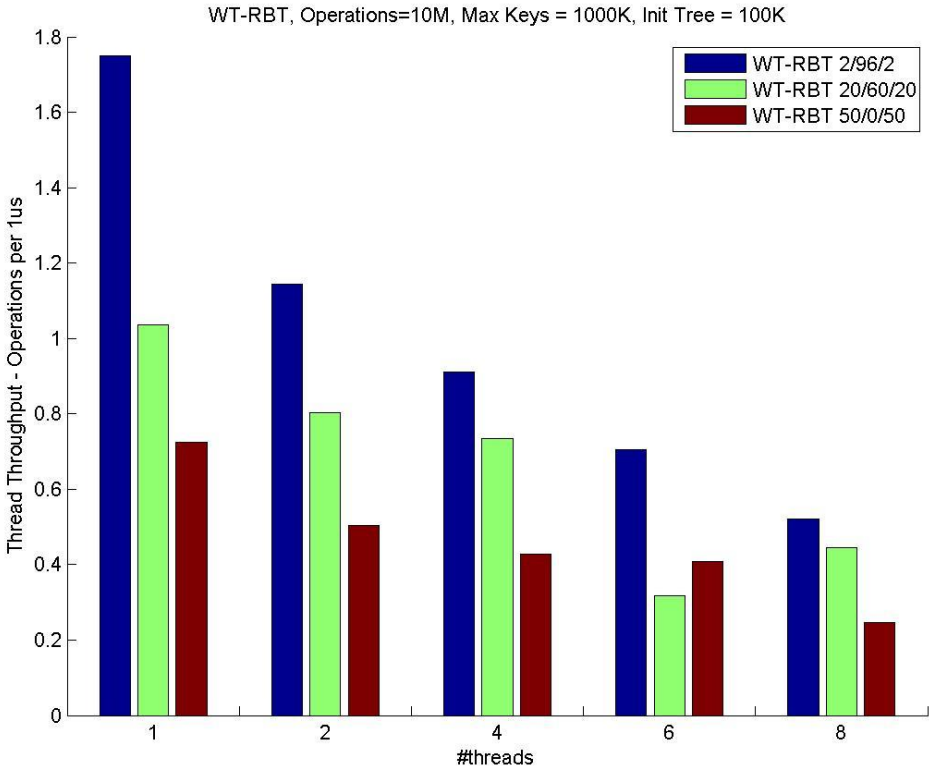
Η κακή επίδοση του αλγορίθμου έχει αντίκτυπο και στο throughput κάθε διεργασίας που μειώνεται σημαντικά με αύξηση του αριθμού των διεργασιών που τρέχουν παράλληλα, όπως παρουσιάζεται και στις Εικόνες 4-21, 4-22, 4-23. Παρατηρούμε πως για όλα τα σύνολα εργασίας καθώς και για τα τρία διαφορετικά μεγέθη δέντρων παρουσιάζεται η ίδια συμπεριφορά. Ακόμα και στην περίπτωση του μεγάλου δέντρου με 10^5 κόμβους το throughput από τη μια διεργασία στις οκτώ μειώνεται σχεδόν κατά 50%.



Εικόνα 4-21

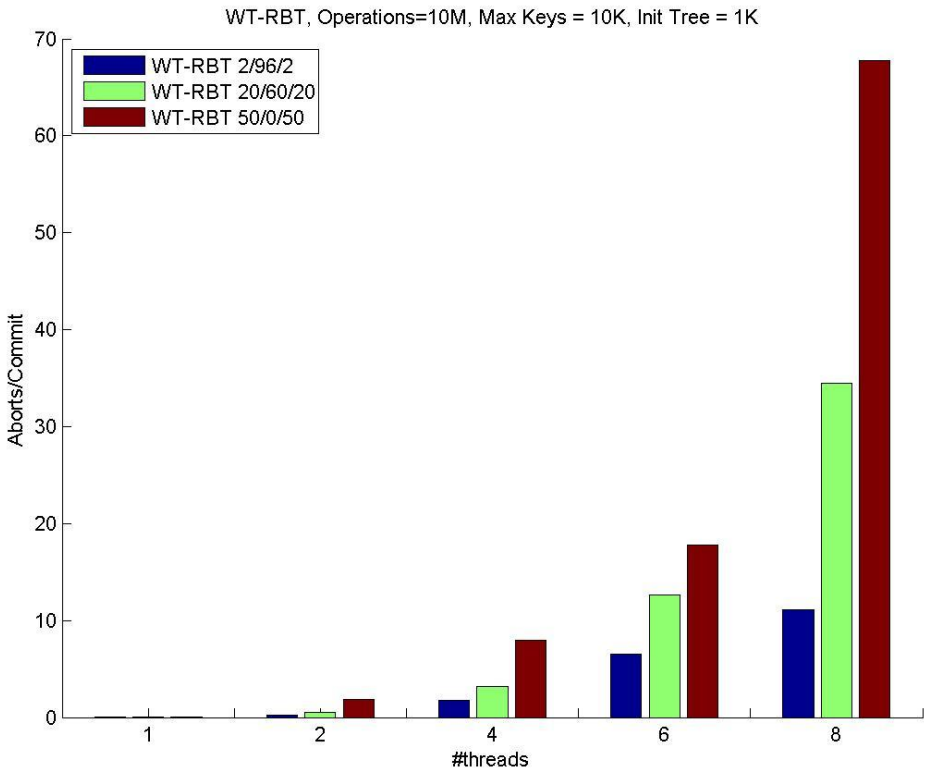


Εικόνα 4-22

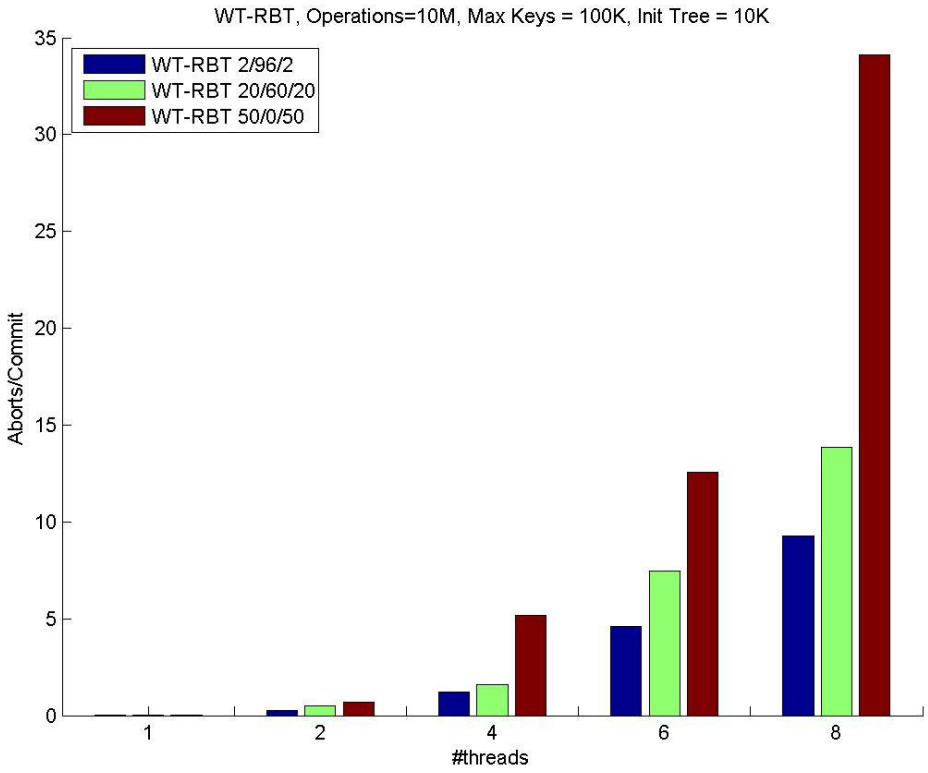


Εικόνα 4-23

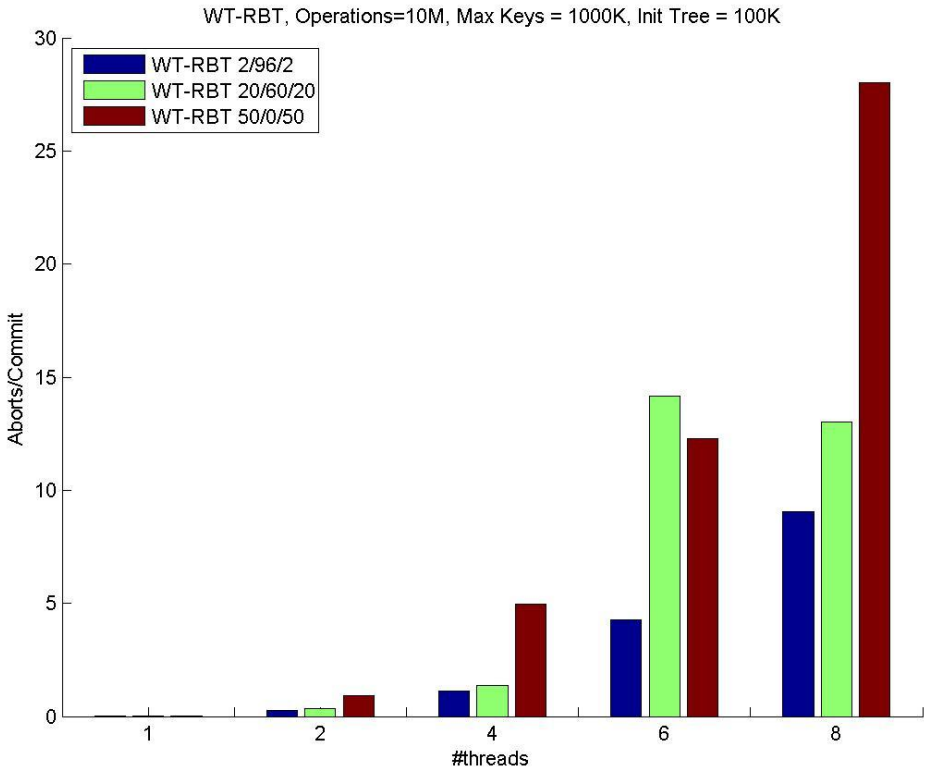
Παρατηρώντας τις Εικόνες 4-24, 4-25, 4-26, διαπιστώνουμε πως και αυτός ο αλγόριθμος υποφέρει από το μεγάλο αριθμό aborts, κάτι που έχει έντονη επίπτωση στην απόδοσή του. Το πρόβλημα αυτό γίνεται ποιο έντονο καθώς αυξάνουμε των αριθμό των threads που τρέχουν παράλληλα και άρα τον αριθμό των Window-Transactions που επιχειρούν να διασχίσουν το δέντρο ταυτόχρονα. Παρατηρούμε ωστόσο πως καθώς αυξάνουμε το μέγεθος του δέντρου ο αριθμός των aborts μειώνεται, κάτι που αντιστοιχεί με την αύξηση της επίδοσης του αλγορίθμου στην περίπτωση αυτή. Μάλιστα από το μικρότερο στο μεγαλύτερο δέντρο τα aborts έχουν μειωθεί σχεδόν κατά 50%. Αυτό συμβαίνει επειδή το δέντρο αυξάνει σε πλάτος και ύψος, δίνοντας έτσι την δυνατότητα σε περισσότερα Window-Transactions να τρέχουν παράλληλα χωρίς να δημιουργούνται conflicts.



Εικόνα 4-24



Εικόνα 4-25



Εικόνα 4-26

Εξαιτίας των μεγάλων αριθμών από aborts, επιχειρήσαμε να οδηγήσουμε τις διεργασίες σε fallback-path, το οποίο υλοποιήσαμε με coarse grained locking, μετά από ένα αριθμό aborts. Αυτή η απόπειρα ωστόσο είχε δυσμενή αποτελέσματα. Ο κυριότερος λόγος που συμβαίνει αυτό, είναι πως όταν μια διεργασία κλειδώνει το δέντρο, αναγκάζει όλες τις άλλες να πάνε σε explicit abort. Αυτό έχει ως αποτέλεσμα οι διεργασίες να πρέπει να επανεκκινήσουν την εργασία που εκτελούσαν από την αρχή, δηλαδή ξαναρχίζοντας από την ρίζα του δέντρου. Η απαίτηση αυτή προκύπτει από το γεγονός ότι η διεργασία που κλειδώνει το δέντρο πιθανόν να αλλάξει το μονοπάτι που διέσχισαν οι υπόλοιπες διεργασίες και άρα αυτές πρέπει να ξαναδιασχίσουν το δέντρο για την επιλογή του ορθού μονοπατιού. Επιπλέον, αυτή η διαδικασία, οδηγώντας τις διεργασίες πίσω στην ρίζα, αυξάνει περαιτέρω τα aborts καθώς όλα τα Window-Transactions συνωστίζονται στον ίδιο χώρο. Τελικά οι διεργασίες σειριοποιούνται στο lock αφού πρώτα έχει προστεθεί ένα μεγάλο χρονικό κόστος λόγω των πολλαπλών aborts.

Τελικά η υλοποίηση των Window-Transaction καταφέρνει να μας δώσει σταθερό μέγεθος transaction κάτι που μας απελάσει από το πρόβλημα των capacity aborts. Επίσης αυτό έχει ως αποτέλεσμα να μπορεί να εκτελεστεί σε δέντρο οποιουδήποτε μεγέθους, ενώ φαίνεται να δίνει καλύτερα αποτελέσματα για μεγαλύτερα δέντρα.

Παρόλα αυτά, παρουσιάζει μεγάλο αριθμό aborts. Επίσης για την εκτέλεση μιας μόνο εργασίας στο δέντρο απαιτούνται $log_2 n$ transactions, (όπου n ο αριθμός των κόμβων του δέντρου) κάτι που οδηγεί σε ένα μεγάλο overhead. Ακόμα το speedup και το throughput του αλγορίθμου αυτού είναι προβληματικά. Πρέπει επίσης να σημειωθεί πως σε σύγκριση με την προηγούμενη υλοποίηση του RTB-RBT πρόκειται για μια πολύ πιο περίπλοκη και επίπονη υλοποίηση.

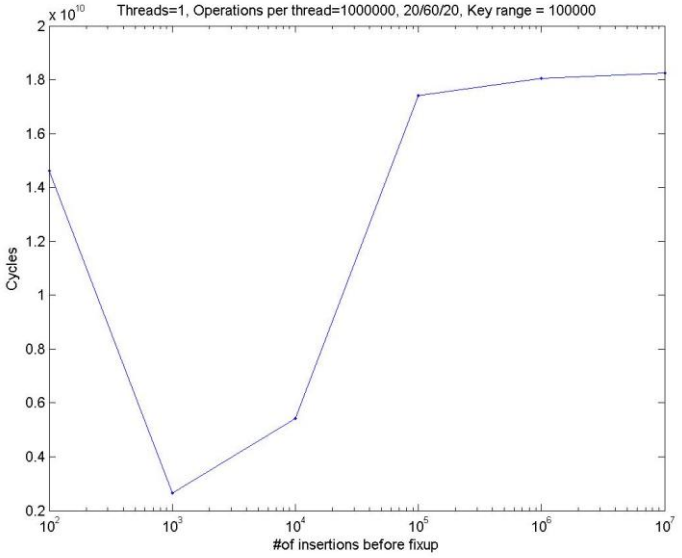
4.4 Πειραματικά Αποτελέσματα: RB-RBT

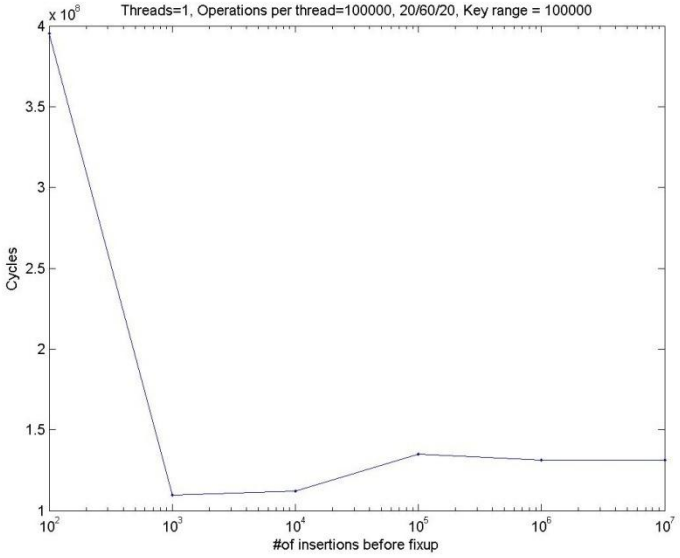
Η υλοποίηση αυτή καταφεύγει στην γνωστή ιδέα του παράλληλου προγραμματισμού, για λογικές εκτελέσεις εργασιών με στόχο να μειώσουμε την ανάγκη για συγχρονισμό και άρα τα πιθανά conflicts που αναπτύσσονται.

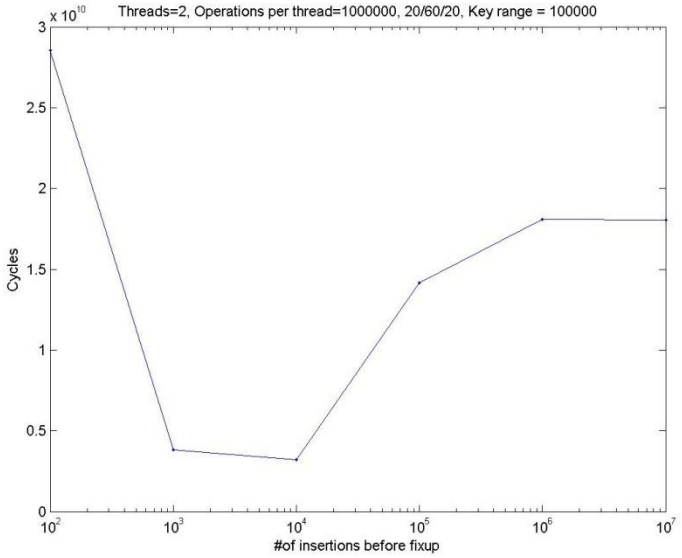
Ωστόσο, όπως εξηγήθηκε και στο κεφάλαιο 3.4, καθώς αφαιρούμε λογικά κόμβους και υποσημειώνουμε τα σημεία του δέντρου που χρειάζονται αναδιάταξη χωρίς όμως να εκτελούμε τις φυσικές εργασίες, αυξάνουμε το χρόνο αναζήτησης στο δέντρο. Για τον λόγο αυτό αρχικά προσπαθήσαμε να βρούμε κάθε πότε πρέπει να διακόπτουμε την εκτέλεση των εργασιών και να αναδιατάσσουμε το δέντρο, εξυπηρετώντας όλα τα λογικά αιτήματα που έχουν μαζευτεί.

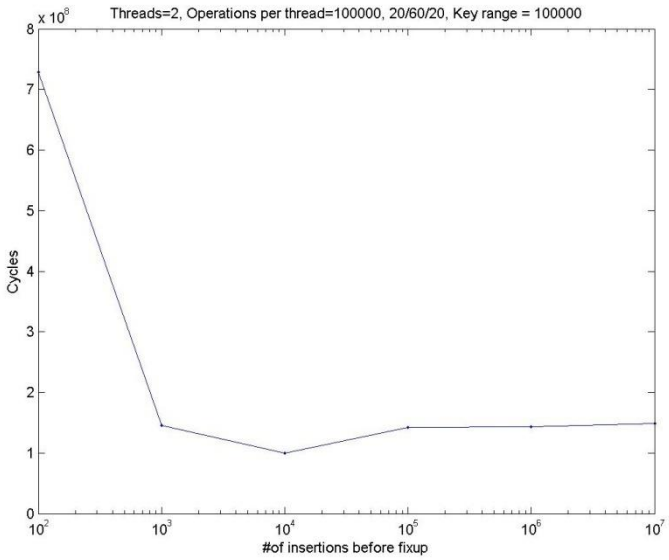
Για να το πετύχουμε αυτό, έχουμε μια επιπλέον διεργασία που έχει στόχο την αναδιάταξη του δέντρου. Παρακολουθεί τον αριθμό των εισαγωγών κόμβων που έχουν γίνει από τις υπόλοιπες διεργασίες και αποφασίζει ασύγχρονα να διακόψει προσωρινά την εκτέλεσή τους και να αναδιατάξει το δέντρο. Ο λόγος που παρακολουθεί μόνο τον αριθμό εισαγωγών είναι πως ουσιαστικά αυτές είναι που ευθύνονται για την μετατροπή του δέντρου μας από ισοζυγισμένο σε μη-ισοζυγισμένο, πράγμα που αυξάνει το χρόνο αναζήτησης κόμβου.

Στα πειράματα που ακολούθησαν, Εικόνες 4-27, 4-28, είδαμε πως παίρνουμε σημαντικά καλύτερη επίδοση όταν διακόπτουμε τις διεργασίες για αναδιάταξη ανά 10^4 εισαγωγές κόμβων. Στις γραφικές αυτές, παρατηρούμε πως οι πολύ συχνές διακοπές έχουν ως αποτέλεσμα να έχουμε μεγάλους χρόνους εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι η διακοπή της εκτέλεσης εισάγει ένα χρονικό κόστος, που έχει να κάνει με το συγχρονισμό των διεργασιών. Οπότε έχουμε δύο παραμέτρους που συναγωνίζονται, που είναι το κόστος του συγχρονισμού και το κόστος αναζήτησης του μη-αναδιαταγμένου δέντρου. Το κόστος του συγχρονισμού γίνεται εντονότερο όταν αυξάνουμε τον αριθμό των διεργασιών. Οπότε έχοντας πλέον επιλέξει τις $\mathbf{10^4}$ **εισαγωγές** σαν συχνότητα αναδιάταξης του δέντρου και εξυπηρέτησης των λογικών αιτημάτων πήραμε τις μετρήσεις που παρουσιάζονται στην συνέχεια.

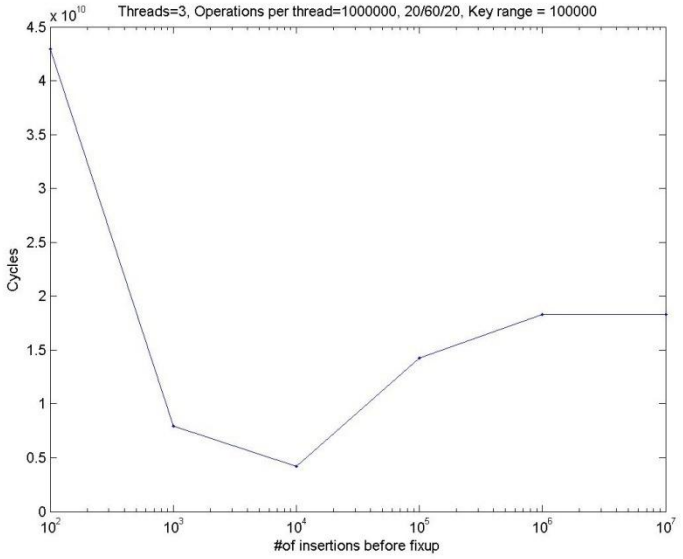

x 10^10
Threads=1, Operations per thread=1000000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

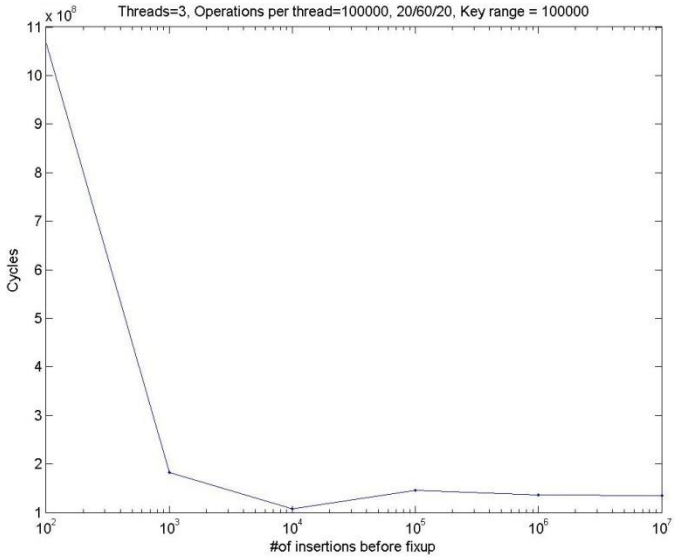

x 10^8
Threads=1, Operations per thread=100000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

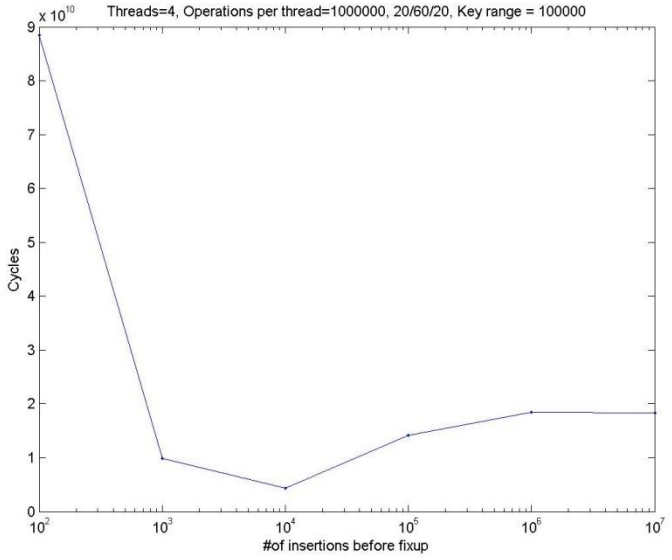

x 10^10
Threads=2, Operations per thread=1000000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

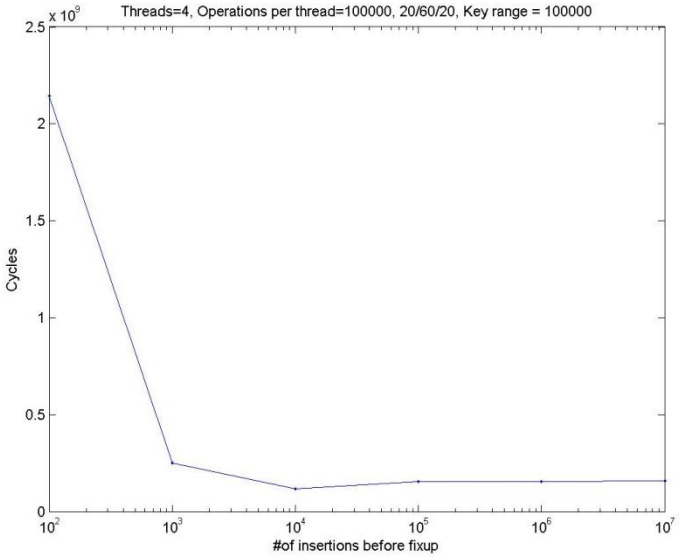

x 10^8
Threads=2, Operations per thread=100000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

Εικόνα 4-27


x 10^10
Threads=3, Operations per thread=1000000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

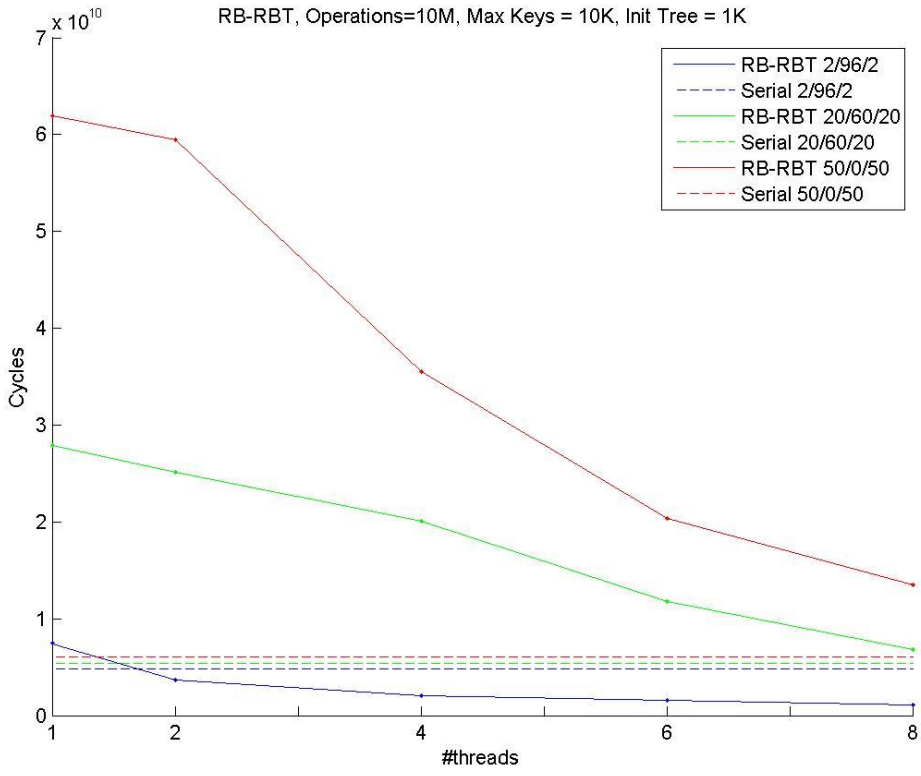

x 10^8
Threads=3, Operations per thread=100000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup


x 10^10
Threads=4, Operations per thread=1000000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

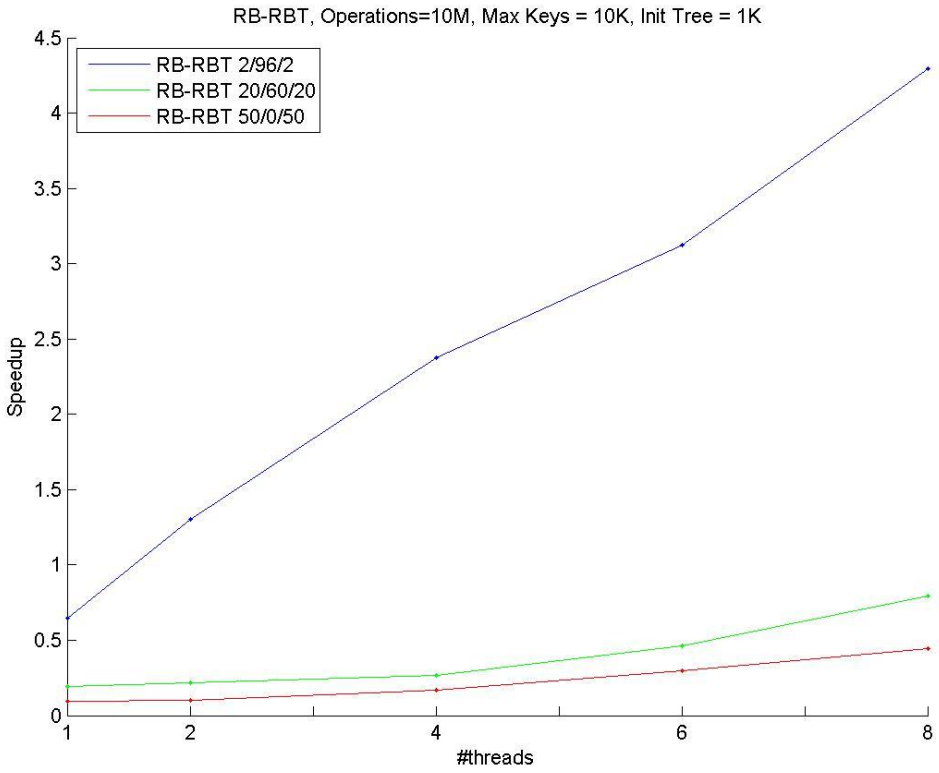

x 10^9
Threads=4, Operations per thread=100000, 20/60/20, Key range = 100000
Cycles
#of insertions before fixup

Εικόνα 4-28

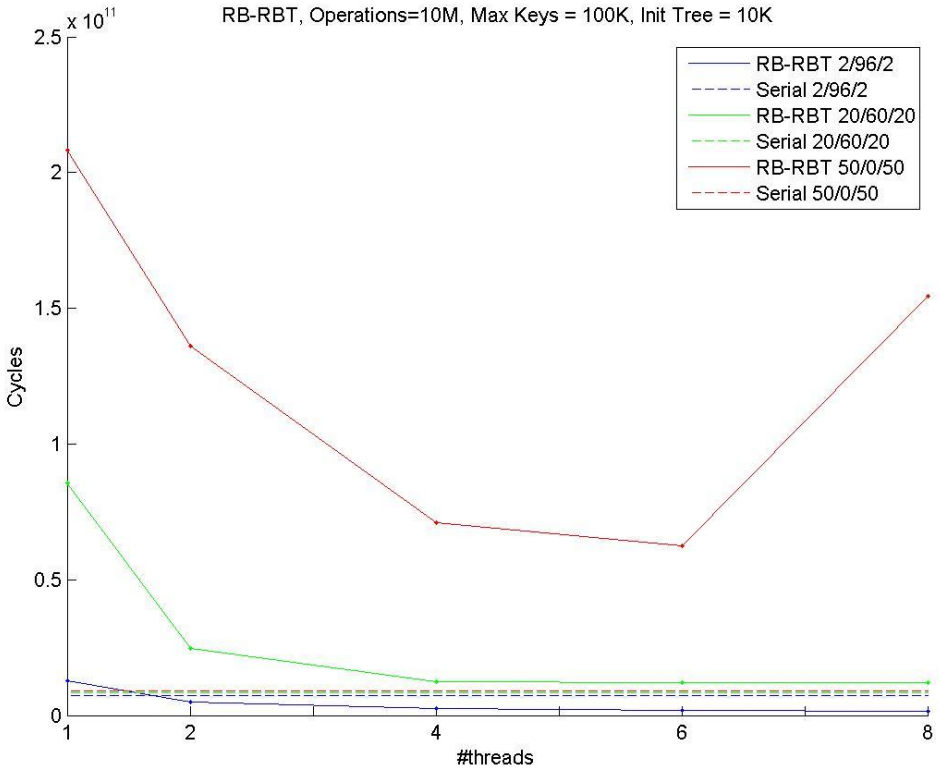
Το πρώτο πράγμα που παρατηρούμε είναι πως η υλοποίηση αυτή παρουσιάζει πολύ καλά αποτελέσματα στις περιπτώσεις όπου έχουμε ελαφρύ φόρτο εργασίας με έμφαση στις αναζητήσεις κόμβων. Αυτό ήταν κάτι που αναμέναμε καθώς οι αναζητήσεις γίνονται ασύγχρονα δίνοντας την δυνατότητα οι διεργασίες να τρέχουν παράλληλα χωρίς να παρουσιάζονται conflicts. Η ασύγχρονη αναζήτηση γίνεται δυνατή καθώς όλες οι μεταβολές στο δέντρο γίνονται λογικά. Αυτό φαίνεται και στις Εικόνες 4-29 έως 4-34. Στις εικόνες αυτές παρατηρούμε επίσης πως το overhead που εντάσσει η υλοποίησή μας ως προς την σειριακή, διαφέρει για τα διαφορετικά σύνολα εργασιών. Αυτό συμβαίνει καθώς για μεγαλύτερο αριθμό εισαγωγών αναγκαζόμαστε πιο συχνά να συγχρονίσουμε τις διεργασίες και να αναδιατάξουμε το δέντρο, αυξάνοντας έτσι το overhead του προγράμματος. Μια άλλη σημαντική παρατήρηση είναι πως ο αλγόριθμος αυτός μας δίνει σχετικά σταθερή αύξηση στην κλιμακωσιμότητα, πράγμα που υπονοεί πως αν είχαμε στην διάθεσή μας επεξεργαστή με περισσοτέρους πυρήνες, θα μπορούσαμε να δούμε περαιτέρω βελτίωση στα αποτελέσματα μας.



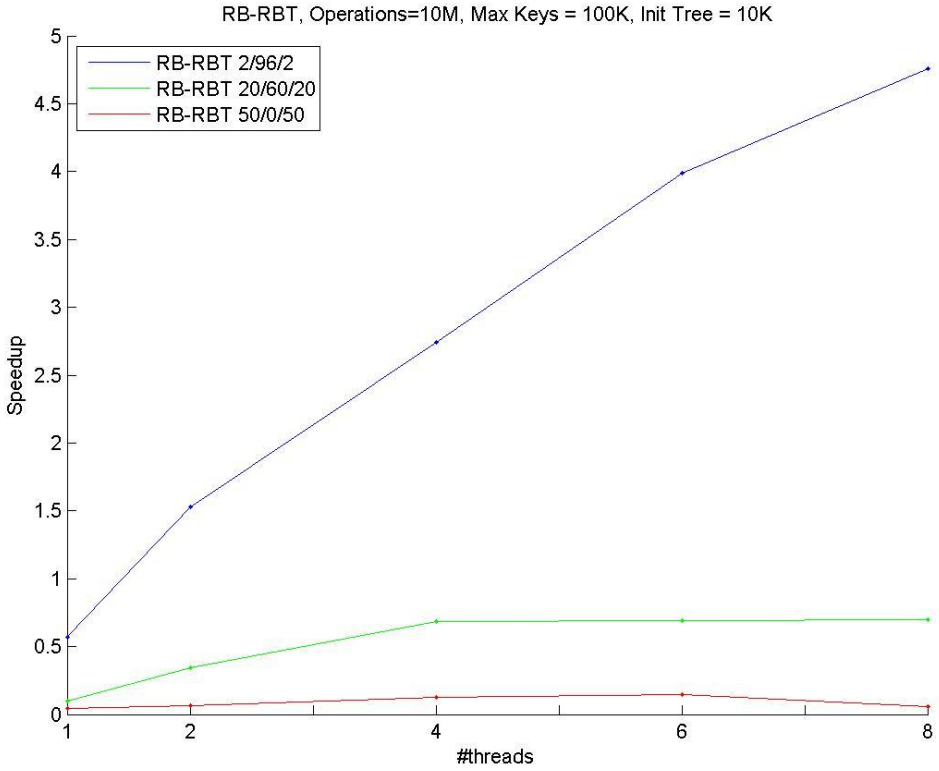
Εικόνα 4-29



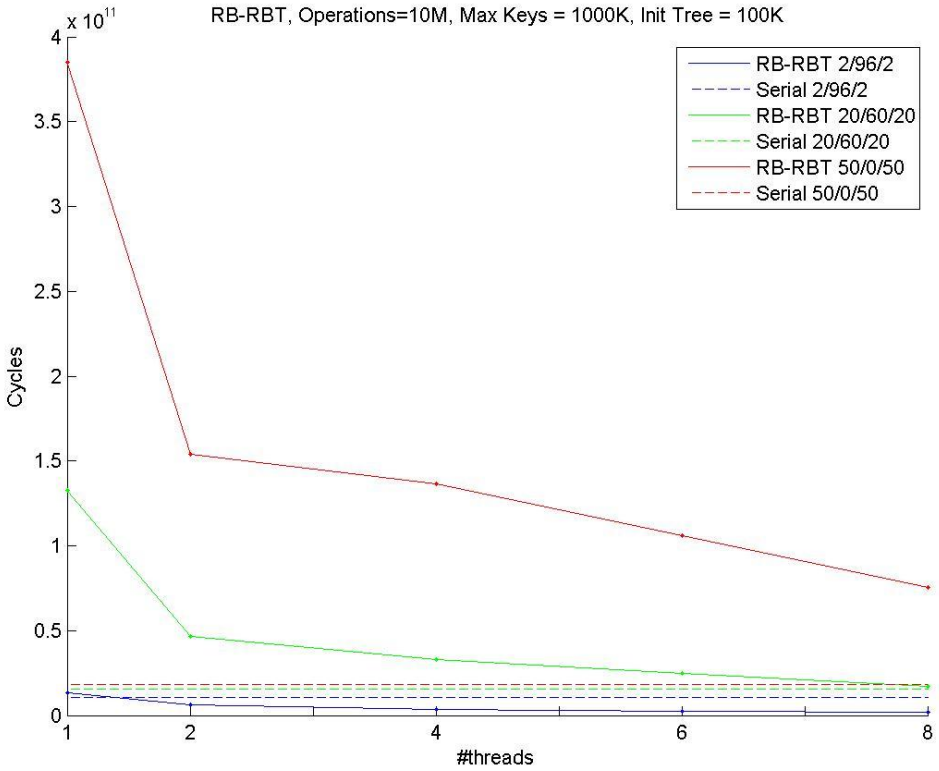
Εικόνα 4-30



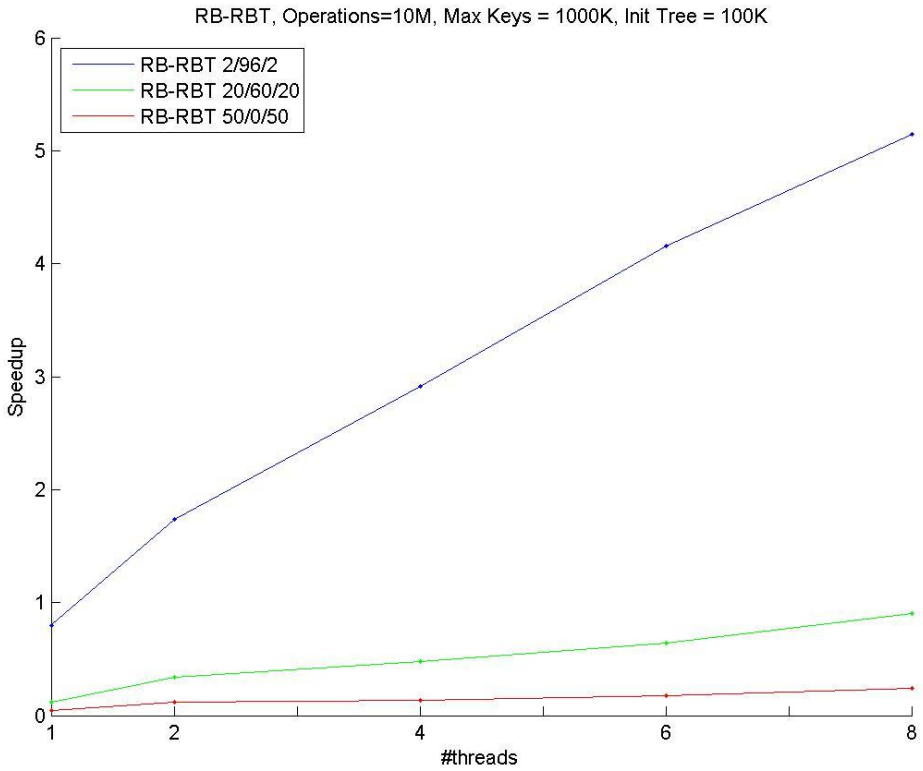
Εικόνα 4-31



Εικόνα 4-32

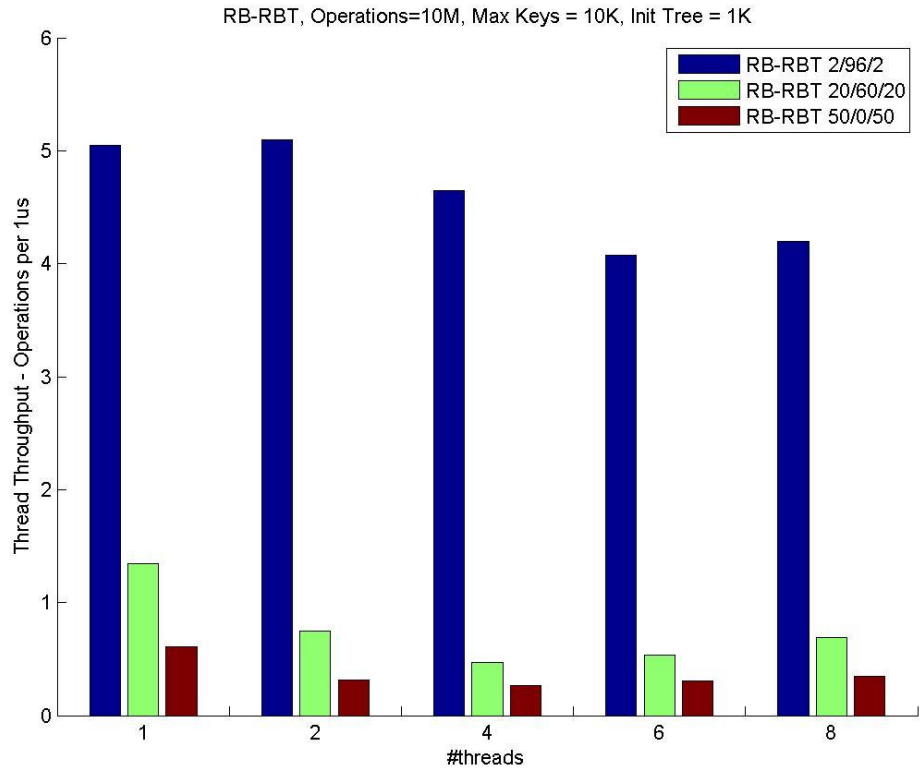


Εικόνα 4-33

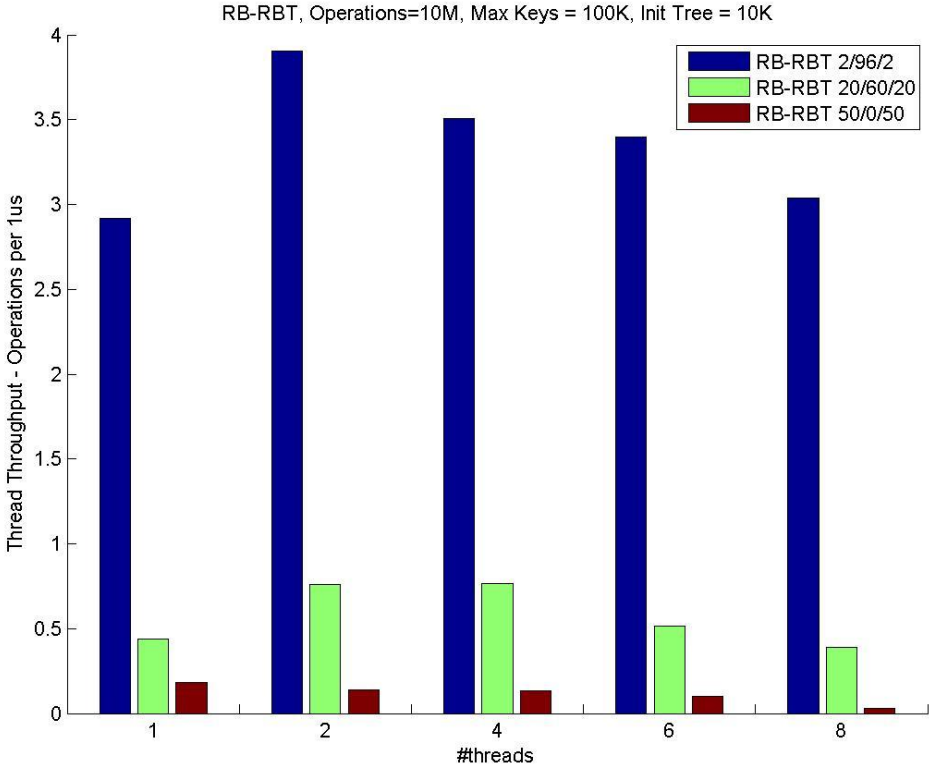


Εικόνα 4-34

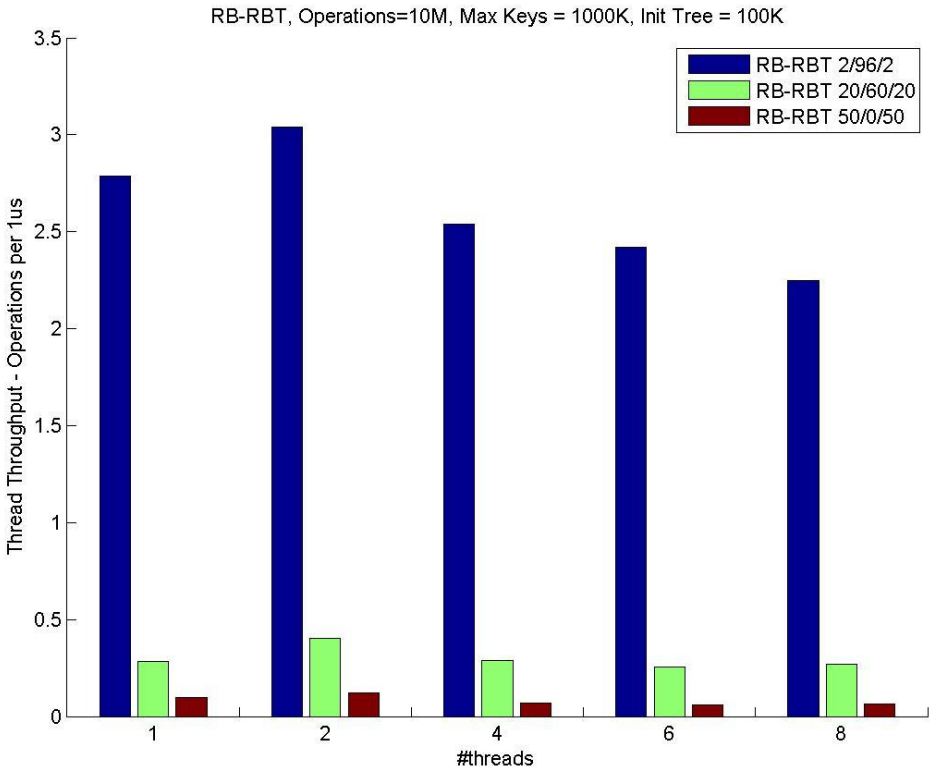
Εντυπωσιακό είναι και το throughput που παρουσιάζει αυτή η υλοποίηση, Εικόνες 4-35, 4-36, 4-37, καθώς σε όλες τις περιπτώσεις παραμένει σχετικά σταθερό. Η μεγαλύτερη μείωση στο throughput παρουσιάζεται στην περίπτωση των 8 διεργασιών όπου, σε σύγκριση με την μια διεργασία, μειώνεται κατά 0.5 εργασίες ανά microsecond. Η συμπεριφορά αυτή στο throughput ενδυναμώνει την υπόθεση πως ο αλγόριθμος αυτός θα έδινε ακόμα καλύτερα αποτελέσματα σε επεξεργαστή με μεγαλύτερο αριθμό από πυρήνες.



Εικόνα 4-35

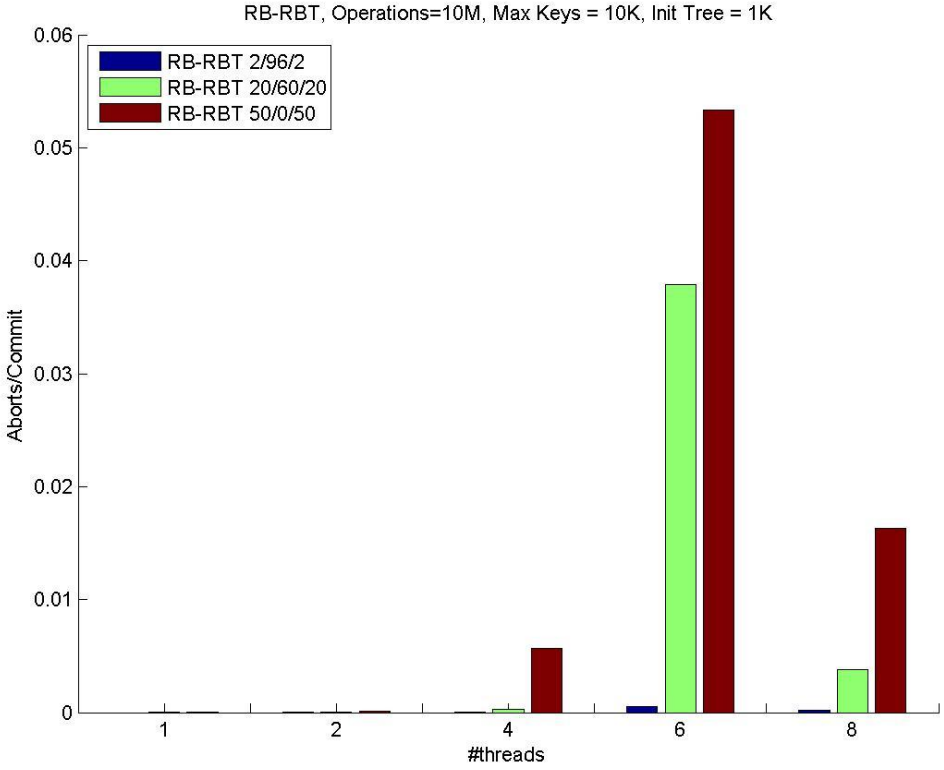


Εικόνα 4-36

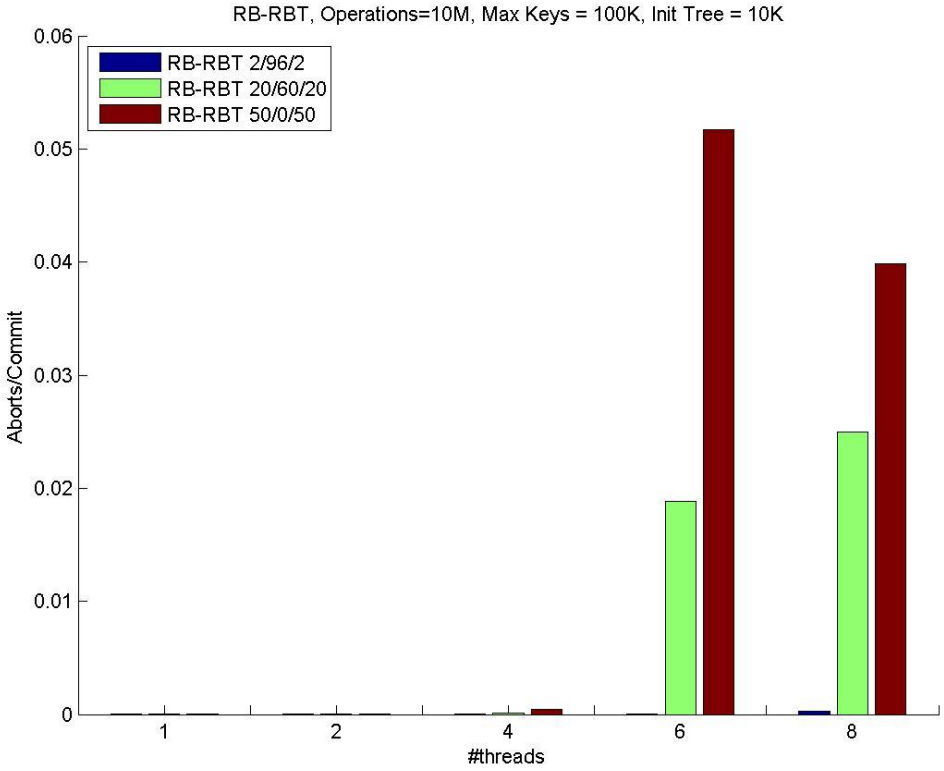


Εικόνα 4-37

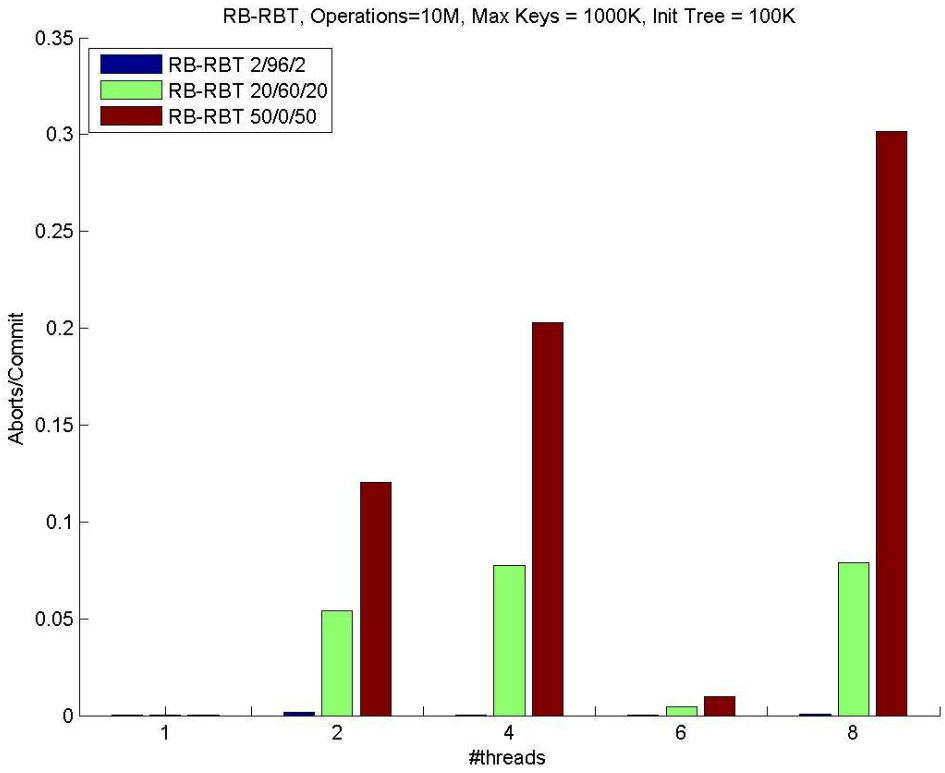
Επίσης εντυπωσιακά είναι και τα αποτελέσματα όσο αφορά το αριθμό των aborts. Στην υλοποίηση αυτή, όπως φαίνεται στις Εικόνες 4-38, 4-39, 4-40, τα aborts είναι ελάχιστα πράγμα που μας οδηγεί στο συμπέρασμα πως δεν έχουν ουσιαστική επίπτωση στην απόδοση του αλγορίθμου. Επίσης παρατηρούμε πως η εικόνα αυτή δεν αλλάζει με μεταβολές στο μέγεθος του δέντρου καθώς το abort rate παραμένει πάντα μικρότερο της μονάδας. Ο λόγος που συμβαίνει αυτό είναι επειδή το μέγεθος του transaction είναι πολύ μικρό πράγμα που δίνει την δυνατότητα σε πολλαπλές διεργασίες να τρέχουν παράλληλα χωρίς να προκύπτει ανάγκη για συγχρονισμό.



Εικόνα 4-38

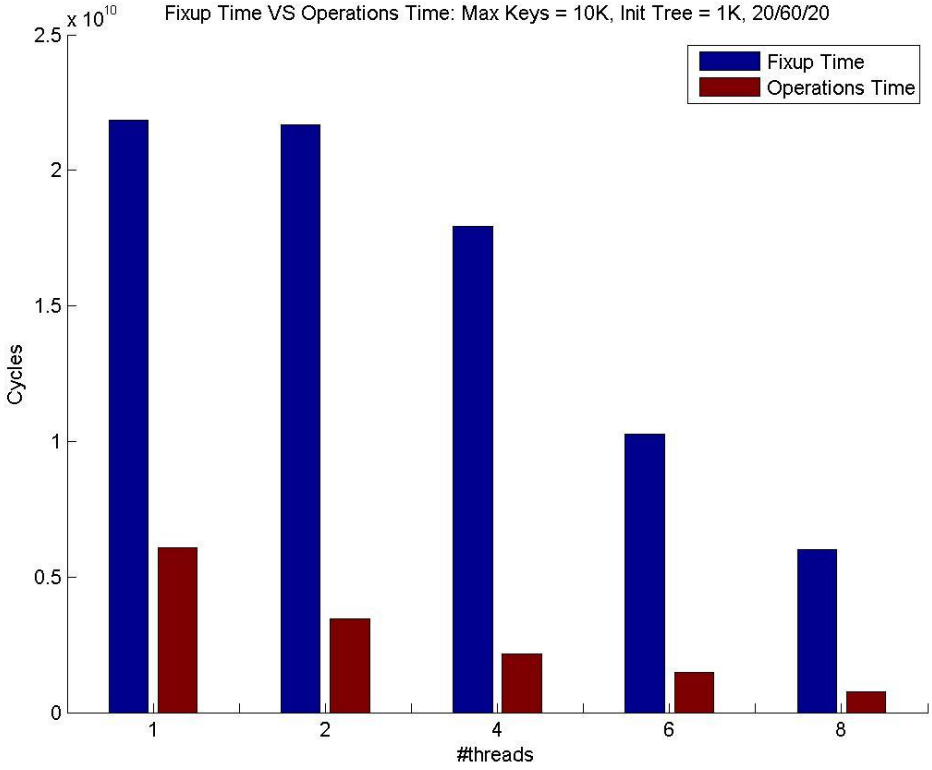


Εικόνα 4-39



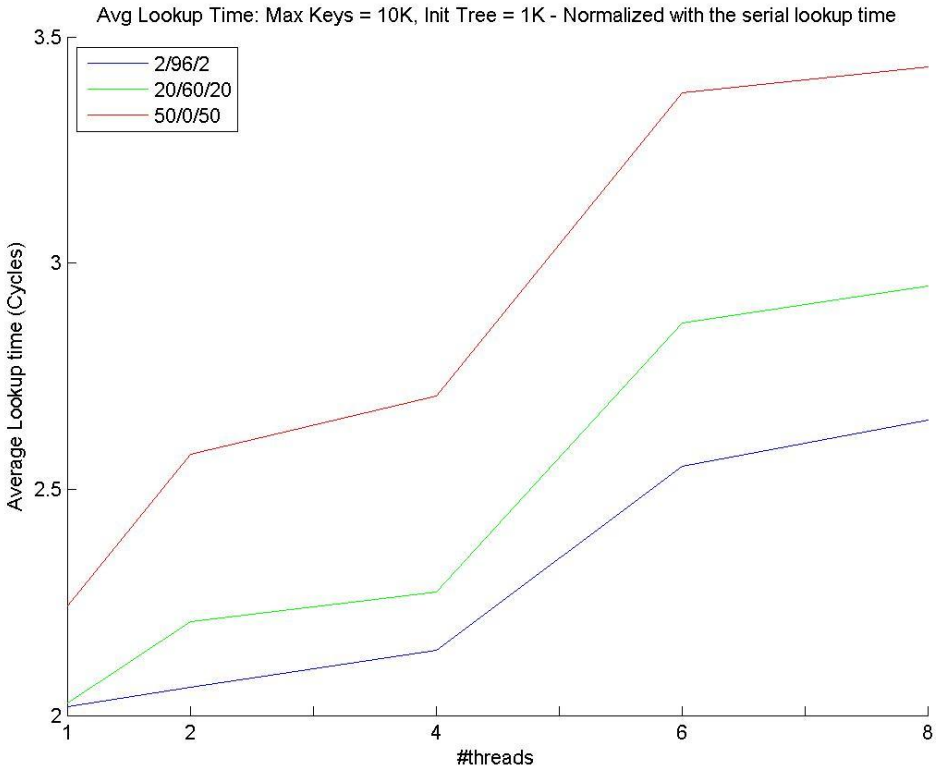
Εικόνα 4-40

Καθώς όμως το abort rate, παραμένει τόσο μικρό σημαίνει πως πρέπει να ερευνήσουμε την αιτία που κρατά την απόδοση της υλοποίησης αυτής περιορισμένη, ειδικότερα στην περίπτωση μεγάλου φόρτου εργασίας. Το ποια είναι η αιτία αυτή φαίνεται πολύ καθαρά στην Εικόνα 4-41, όπου συγκρίνουμε το χρόνο που αναλώνει το πρόγραμμά μας για να εκτελεί εργασίες (εισαγωγές, αφαιρέσεις, αναζητήσεις) πάνω στο δέντρο και το χρόνο που σπαταλά αναδιατάσσοντας το δέντρο και εξυπηρετώντας τα λογικά αιτήματα που έχουν μαζευτεί. Αντιλαμβάνεται εύκολα κανείς πως η εξυπηρέτηση των λογικών αιτημάτων είναι αυτή που περιορίζει την επίδοση της υλοποίησης αυτής, καθώς αποτελεί και το μεγαλύτερο μέρος της εκτέλεσής της. Θυμίζουμε πως η διαδικασία αυτή εκτελείται σειριακά και απαιτεί την διακοπή της διεκπεραίωσης εργασιών από τις υπόλοιπες διεργασίες.



Εικόνα 4-41

Ενδιαφέρον έχει επίσης να δούμε πως διαμορφώνεται ο μέσος χρόνος αναζήτησης κόμβου στο δέντρο, καθώς στην υλοποίηση αυτή όλες οι εργασίες καταλήγουν τελικά σε αναζητήσεις κόμβων. Στην περίπτωση της αφαίρεσης ή της εισαγωγής κόμβου έχουμε επιπλέον της αναζήτησης και την λογική υποσημείωση των κόμβων με ανάλογο αίτημα, που όμως ουσιαστικά δεν εισάγει αξιοσημείωτη χρονική καθυστέρηση. Στην Εικόνα 4-42 παρουσιάζεται η μεταβολή του χρόνου αναζήτησης καθώς αυξάνουμε των αριθμό των διεργασιών που τρέχουν παράλληλα. Ο χρόνος αυτός είναι κανονικοποιημένος ως προς το σειριακό χρόνο αναζήτησης. Παρατηρούμε πως γενικότερα έχουμε μια άνοδο του χρόνου αυτού με αύξηση των διεργασιών. Αν και δεν φαίνεται να υπάρχει κάποια προφανής αιτία για να συμβαίνει αυτό, ίσως έχει να κάνει με την μικρή αύξηση των conflicts με αύξηση των διεργασιών. Ένας άλλος λόγος που μπορεί να προκαλεί την αύξηση αυτή είναι η πιθανότητα αναζήτηση ενός κόμβου, σε ένα μονοπάτι που δυναμικά μεγαλώνει καθώς άλλες διεργασίες, που προηγούνται χρονικά, εντάσσουν νέους κόμβους στο μονοπάτι αυτό. Αυτό το σενάριο έχει περισσότερες πιθανότητες να συμβαίνει όσο αυξάνουμε τον αριθμό των διεργασιών. Επίσης διαπιστώνουμε πως για τους διαφορετικούς φόρτους εργασίας έχουμε μια σημαντική διαφορά στους χρόνους. Ο λόγος είναι πως αυξάνοντας τις λογικές εισαγωγές και διαγραφές, το δέντρο γίνεται πιο έντονα και πιο συχνά μη-ισοζυγισμένο, προκαλώντας έντονη αύξηση στο χρόνο αναζήτησης.



Εικόνα 4-42

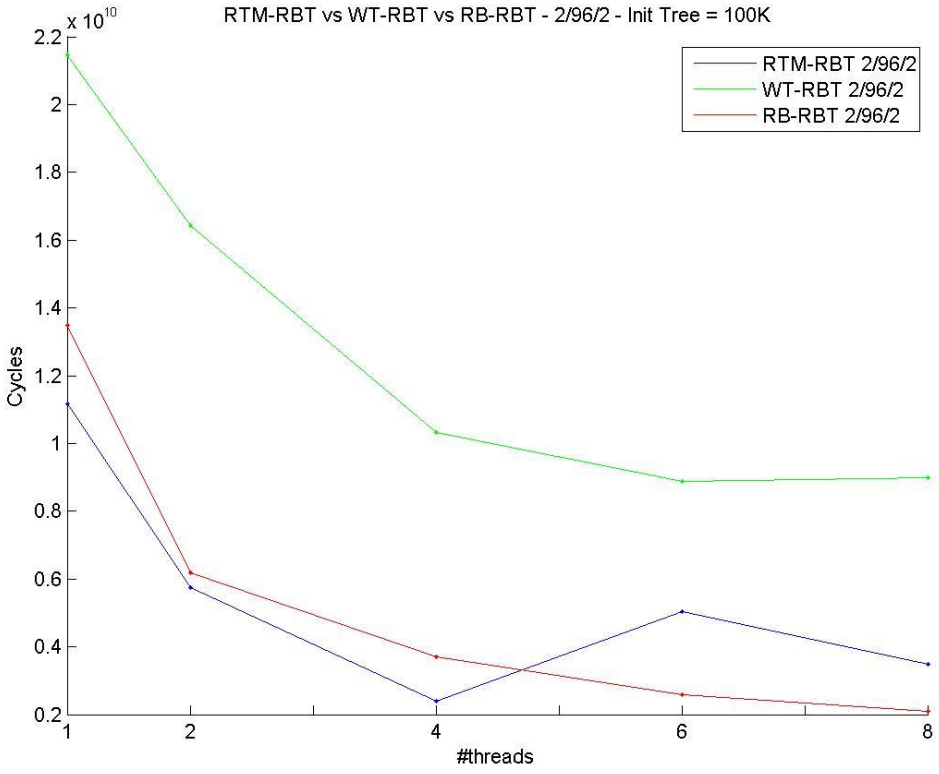
Με την υλοποίηση αυτή πετύχαμε τελικά να έχουμε πολύ καλή απόδοση για τις περιπτώσεις μικρού φόρτου εργασίας. Επίσης πετύχαμε να κρατάμε σχετικά σταθερό throughput ανεξάρτητα από τον αριθμό των διεργασιών που τρέχουν παράλληλα και σταθερή κλιμακωσιμότητα. Ακόμα τα aborts της υλοποίησης αυτής είναι ελάχιστα δίνοντας της έτσι το πλεονέκτημα να έχει ίδια συμπεριφορά για διαφορετικά μεγέθη δέντρων.

Το μεγαλύτερό της ωστόσο μειονέκτημα είναι το μεγάλο overhead που εντάσσει η διαδικασία αναδιάταξης του δέντρου και η απαίτηση συγχρονισμού των διεργασιών στην περίπτωση αυτή. Αυτό έχει και ως αποτέλεσμα να παίρνουμε πολύ κακή επίδοση για μεγάλους φόρτους εργασίας με αυξημένο αριθμό εισαγωγών κόμβων.

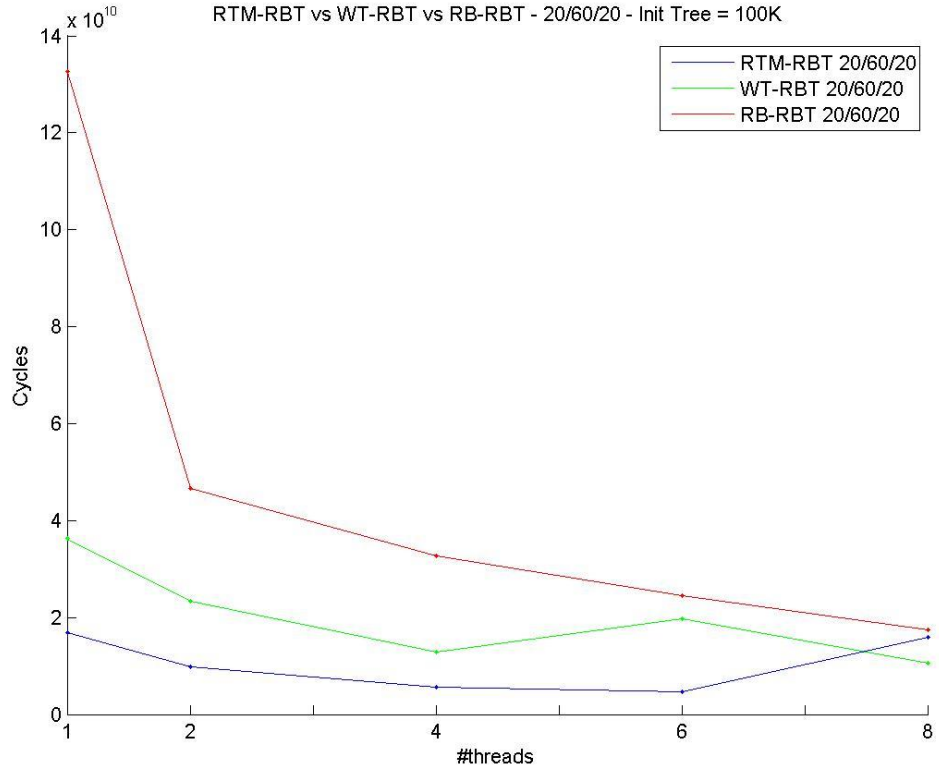
4.5 Σύγκριση Αλγορίθμων - Συμπεράσματα

Κλείνοντας το κεφάλαιο των πειραματικών μετρήσεων θεωρούμε σημαντικό να συγκρίνουμε μεταξύ τους, τις τρεις υλοποιήσεις παράλληλου RBT. Για να το κάνουμε αυτό επιλέξαμε να δείξουμε σε μια γραφική την απόδοση και των τριών υλοποιήσεων. Πιο συγκεκριμένα στην Εικόνα 4-43 παρουσιάζουμε το χρόνο εκτέλεσης και των τριών υλοποιήσεων για 2% εισαγωγές/αφαιρέσεις και 96% αναζητήσεις. Αντίστοιχα στην Εικόνα 4-44 έχουμε τους

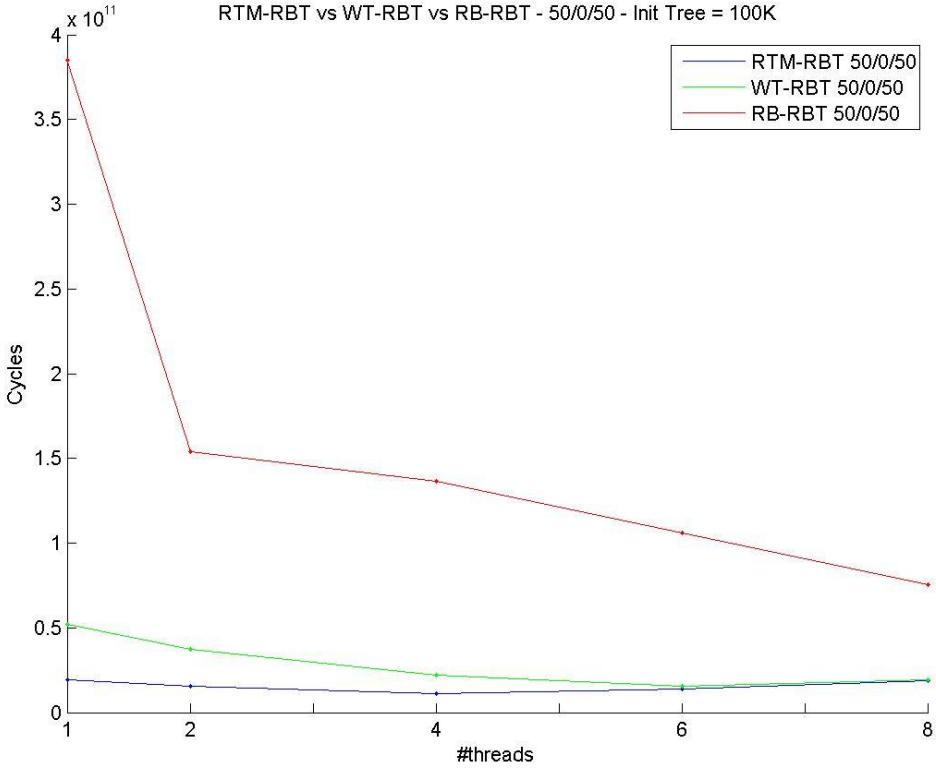
χρόνους για 20% εισαγωγές/αφαιρέσεις και 60% αναζητήσεις. ενώ στην Εικόνα 4-45 για 50% εισαγωγές/αφαιρέσεις και 0% αναζητήσεις.



Εικόνα 4-43



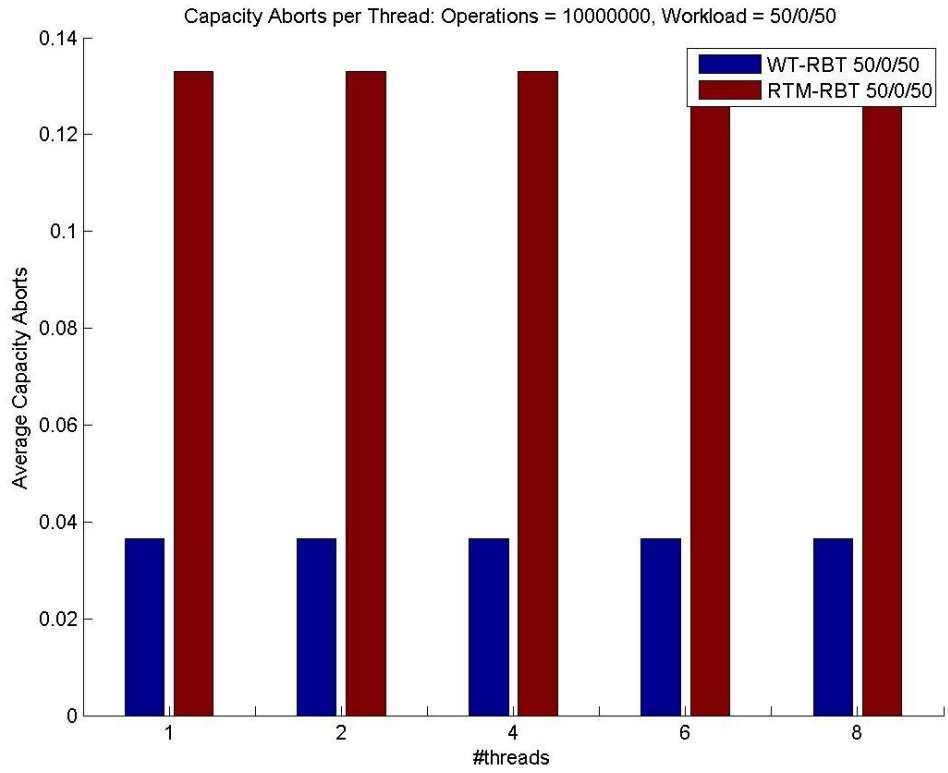
Εικόνα 4-44



Εικόνα 4-45

Και στις τρεις αυτές εικόνες παρουσιάζουμε τα αποτελέσματα για δέντρο μεγέθους 10^5 κόμβων. Παρατηρούμε πως στην περίπτωση του μικρού φόρτου εργασίας, Εικόνα 4-43, καλύτερο χρόνο πετυχαίνει η υλοποίηση RB-RBT για 8 διεργασίες ενώ φαίνεται έντονα πως ο χρόνος αυτός θα μειωνόταν περαιτέρω σε σύστημα με περισσότερους πυρήνες. Αντίθετα στην περίπτωση μέτριου φόρτου εργασίας, Εικόνα 4-44, το καλύτερο χρόνο το πετυχαίνει η υλοποίηση RTM-RBT στις 6 διεργασίες. Ωστόσο λόγω της κακής κλιμακωσιμότητας παρατηρούμε πως στις 8 διεργασίες καλύτερο χρόνο έχει η WT-RTM υλοποίηση. Αντίστοιχη συμπεριφορά παρατηρούμε και στη περίπτωση μεγάλου φόρτου εργασίας, Εικόνα 4-45. Αυτή τη φορά όμως η υλοποίηση RTM-RBT πετυχαίνει το καλύτερο χρόνο για 4 διεργασίες καθώς στην συνέχεια η απόδοση της χειροτερεύει. Αντίθετα η υλοποίηση WT-RTM κλιμακώνει καλύτερα με αποτέλεσμα στις 8 διεργασίες να συναντά την επίδοση της RTM-RBT υλοποίησης.

Συνοψίζοντας, μπορούμε εύκολα να πούμε πως η καλύτερη μέθοδος ως προς την απλότητα και ευκολία υλοποίησης της είναι η RTM-RBT. Ωστόσο πρόκειται για μια υλοποίηση που δεν μπορεί να κλιμακώσει για πάνω από 4 διεργασίες ενώ ταυτόχρονα περιορίζεται σημαντικά από τα capacity aborts και τους περιορισμούς που εντάσσει το HTM. Αυτό μπορούμε να το δούμε και στην Εικόνα 4-46 όπου συγκρίνεται ο μέσος όρος capacity aborts ανά διεργασία για τις υλοποιήσεις WT-RBT και RTM-RBT.



Εικόνα 4-46

Αυτό έχει επίπτωση στην απόδοσή της καθώς οδηγεί τα transaction σε συνεχόμενα capacity aborts, δίνοντας μια ασταθή συμπεριφορά καθώς αλλάζει το μέγεθος του δέντρου. Για το λόγω αυτό θα την εισηγούμασταν στην περίπτωση όπου θέλαμε να εφαρμόσουμε παραλληλοποίηση ενός RBT με σχετικά μικρό αριθμό κόμβων. Αντίθετα η υλοποίηση WT-RB, όπως φάνηκε, δίνει καλύτερα αποτελέσματα σε μεγαλύτερα δέντρα και για τον λόγο αυτό θα την προτιμούσαμε σε περίπτωση που θέλαμε να εφαρμόσουμε παραλληλοποίηση RBT μεγάλου μεγέθους όπου έχουμε σχετικά μεγάλο αριθμό εισαγωγών και αφαιρέσεων κόμβων. Αντίθετα σε περίπτωση μικρού αριθμού εισαγωγών και αφαιρέσεων η υλοποίηση RB-RBT προσφέρει καλύτερο χρόνο εκτέλεσης καθώς και μεγαλύτερη κλιμακωσιμότητα.

5

Επίλογος

5.1 Σύνοψη - Συμπεράσματα πάνω στο TSX

Στη διπλωματική αυτή εργασία είχαμε την ευκαιρία να πειραματιστούμε με μια πραγματική HTM υλοποίηση. Είδαμε πως μπορούμε να εκμεταλλευτούμε το HTM που προσφέρουν οι Haswell επεξεργαστές της Intel για την υλοποίηση παράλληλου κώδικα RBT. Μέσα από την διαδικασία αυτή συναντήσαμε τόσο τα πλεονεκτήματα του TSX όσο και τα μειονεκτήματά του, τους περιορισμούς και τις δυσκολίες που εισάγει η χρήση του. Για τον λόγο αυτό θα θέλαμε συνοψίζοντας να παρουσιάσουμε τις παρατηρήσεις αυτές στον αναγνώστη ώστε να παρουσιάσουμε μια πλήρη εικόνα γύρω από το TSX κάτι που θα τον βοηθήσει πολύ αν θελήσει να πειραματιστεί με το εργαλείο αυτό.

Γενικότερα το TSX δίνει την δυνατότητα στο προγραμματιστή, σχετικά εύκολα, να παραλληλοποιήσει ένα πρόγραμμα, όπως είδαμε και στην RTM-RBT υλοποίηση. Ωστόσο στην τελική, καταφέρνει να μετατρέψει την απλή περίπτωση σε ακόμα πιο απλή, αλλά να κάνει την σύνθετη περίπτωση ακόμα πιο σύνθετη. Το πρώτο όμως πράγμα που προβληματίζει κάποιον, όταν αποφασίζει να εκμεταλλευτεί τις δυνατότητες που του προσφέρει το TSX, είναι το κόστος που προσθέτει το transactional abort. Καθώς όπως αναφέραμε στην περίπτωση του RTM, πρέπει να έχουμε πάντα ένα fallback-path, πρέπει να μπορούμε να αποφασίσουμε μετά από πόσα aborts συμφέρει να εγκαταλείψουμε την προσπάθεια εκτέλεσης του transaction με το HTM και να καταφύγουμε στο fallback path. Δυστυχώς δεν υπάρχει συγκεκριμένη απάντηση στο ερώτημα αυτό. Εξαρτάται αποκλειστικά από την

επίδοση του fallback path και τι επίπτωση έχει η χρήση του σε σύγκριση με την υλοποίηση του transaction. Αυτό το είδαμε στους αλγορίθμους RTM-RBT και WT-RBT όπου το fallback path είχε τελείως διαφορετική επίδραση. Ο μόνος τρόπος να εντοπίσουμε τον ιδανικό αριθμό aborts πριν την χρήση του fallback path είναι μέσω πειραματικών δοκιμών. Επίσης η χρήση του fallback path έχει ως αποτέλεσμα ουσιαστικά να μην μπορούμε να αποφύγουμε τη χρήση των locks. Η απαλλαγή από την χρήση των locks ήταν κάτι που παρουσιαζόταν σαν μείζων πλεονέκτημα του TM.

Κατά την χρήση του TSX παρατηρήσαμε και δύο πολύ μεγάλα μειονεκτήματα του, που δυσκόλευαν πολύ το προγραμματισμό. Αν από προγραμματιστικό λάθος το transaction κατά την εκτέλεσή του οδηγείται σε 'Segmentation Fault', κατά την χρήση του HTM αυτό δεν γίνεται ορατό στον προγραμματιστή. Αντί να τερματιστεί το πρόγραμμα δίνοντας αντίστοιχο μήνυμα, το transaction οδηγείται σε unknown abort και η εκτέλεση του προγράμματος συνεχίζεται κανονικά. Επίσης όλες οι κλήσεις συστήματος μέσα στο transaction οδηγούν και πάλι σε unknown aborts. Οι δύο αυτές συμπεριφορές κάνουν πολύ δύσκολη την αποσφαλμάτωση ενός transactional κώδικα.

Τέλος πρέπει να αναφέρουμε δύο τεχνικές που παρατηρήσαμε να βελτιώνουν κατά πολύ την απόδοση ενός προγράμματος που εκμεταλλεύεται το HTM. Η πρώτη τεχνική είναι η αποφυγή αχρείαστων προσπαθειών εκτέλεσης του transaction. Αυτό μπορούμε να το πετύχουμε διαβάζοντας το είδος του abort που προκύπτει από το καταχωρητή EAX και αναλόγως να αποφασίζουμε αν αξίζει ή όχι να ξαναπροσπαθήσουμε να εκτελέσουμε το transaction. Για παράδειγμα σε περίπτωση συνεχόμενων capacity aborts, ή aborts για τα οποία το υλικό δεν μπορεί να μας καθορίσει την αιτία, συμφέρει να οδηγηθούμε άμεσα στο fallback path καθώς τέτοιες περιπτώσεις μάλλον δεν πρόκειται να μπορέσουν να εκτελεστούν ποτέ με την χρήση του HTM. Η δεύτερη τεχνική αφορά την εφαρμογή padding στις μεταβλητές που χρησιμοποιούμε στο transaction και στις οποίες έχουν πρόσβαση όλες οι διεργασίες. Το padding γίνεται έτσι ώστε κάθε μεταβλητή να καταλαμβάνει μια ολόκληρη γραμμή της cache. Αυτό βοηθά να αποφύγουμε aborts που οφείλονται σε false sharing. Θυμίζουμε πως τα conflicts ανιχνεύονται με την βοήθεια του πρωτοκόλλου συνάφειας που ενημερώνει για το διάβασμα ή εγγραφή ανά γραμμή της cache και όχι για κάθε θέση μνήμης ξεχωριστά.

Σαν γενικότερο συμπέρασμα πρέπει να σημειωθεί πως το TM δεν είναι η ιδανική λύση για όλα τα προβλήματα συγχρονισμού. Σε περιπτώσεις όπου όντως χρειάζεται έντονος συγχρονισμός τα συστήματα TM ουσιαστικά αδυνατούν να προσφέρουν ποιοτικές λύσεις. Επίσης είδαμε πως παρόλο που το HTM φημίζεται για την επίδοση του ως προς το STM, εντάσσει τους δικούς τους περιορισμούς και δυσκολίες. Χαρακτηριστική είναι η αδυναμία του να εκτελεί transactions πολύ μεγάλου μεγέθους.

5.2 Μελλοντικές Επεκτάσεις

Κλείνοντας θα θέλαμε να προτείνουμε κάποιες επεκτάσεις της εργασίας αυτής που θα μπορούσαν μελλοντικά να συνεισφέρουν περαιτέρω.

Όσο αφορά τους αλγορίθμους που υλοποιήθηκαν, θα ήταν χρήσιμο να δούμε στην περίπτωση του WT-RBT πως μεταβάλλεται η συμπεριφορά του για διαφορετικά μεγέθη του παραθύρου στο οποίο εκτελείται το transaction. Αυξάνοντας το μέγεθος του παραθύρου μπορούμε να μειώσουμε τον αριθμό των transactions που χρειάζεται να εκτελεστούν για μια εργασία, κάτι που ίσως βοηθήσει να πετύχουμε καλύτερες επιδόσεις. Επίσης αναφορικά με την υλοποίηση του RB-RBT αξίζει να γίνει μια προσπάθεια παραλληλοποίησης της διαδικασίας αναδιάταξης του δέντρου. Όπως είδαμε η διαδικασία αυτή αποτελεί και το μεγαλύτερο χρόνο εκτέλεσης του προγράμματος και άρα κάτι τέτοιο θα βοηθούσε να δούμε πολύ καλύτερους χρόνους.

Τέλος θα ήταν πολύ ενδιαφέρον να δούμε τις επιδόσεις των αλγορίθμων αυτών σε μελλοντικά συστήματα που θα υποστηρίζουν HTM, μελετώντας την εξέλιξή του καθώς η χρήση του θα γίνεται ολοένα και πιο δημοφιλής.

Η σελίδα αυτή είναι σκόπιμα λευκή.

6

Βιβλιογραφία

[1] T. Knight, «An Architecture for Mostly Functional Languages», *In Proceedings of the 1986 ACM conference on LISP and functional programming (LFP '86). ACM, New York, NY, USA, 105-112. DOI=10.1145/319838.319854 http://doi.acm.org/10.1145/319838.319854,* 1986.

[2] N. Shavit και D. Touitou, «Software Transactional Memory», *In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95). ACM, New York, NY, USA, 204-213. DOI=10.1145/224964.224987 http://doi.acm.org/10.1145/224964.224987,* 1995.

[3] V. J. Marathe, W. N. S. III και M. L. Scott, «Design Tradeoffs in Modern Software Transactional Memory Systems», *IN PROCEEDINGS OF THE 7TH WORKSHOP ON LANGUAGES, COMPILERS, AND RUN-TIME SUPPORT FOR SCALABLE SYSTEMS,* 2004.

[4] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III και M. L. Scott, «Lowering the Overhead of Nonblocking Software Transactional Memory», *DEPT. OF COMPUTER SCIENCE, UNIV. OF ROCHESTER,* 2006.

[5] M. Herlihy, J. Eliot και B. Moss, «Transactional Memory: Architectural Support for Lock-Free Data Structures», *In Proceedings of the 20th annual international symposium*

on computer architecture (ISCA '93). ACM, New York, NY, USA, 289-300. DOI=10.1145/165123.165164 http://doi.acm.org/10.1145/165123.165164, 1993.

[6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis και K. Olukotun, «Transactional Memory Coherence and Consistency», *SIGARCH Comput. Archit. News 32, 2 (March 2004), 102-. DOI=10.1145/1028176.1006711 http://doi.acm.org/10.1145/1028176.1006711,* 2004.

[7] R. Rajwar και J. R. Goudman, «Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution», *In Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34). IEEE Computer Society, Washington, DC, USA, 294-305,* 2001.

[8] D. Dice, Y. Lev, M. Moir and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation", *Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA,* 2009.

[9] R. M. Yoo, C. J. Hughes, K. Lai και R. Rajwar, «Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing», *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). ACM, New York, NY, USA, , Article 19 , 11 pages. DOI=10.1145/2503210.2503232 http://doi.acm.org/10.1145/2503210.2503232,* 2013.

[10] H. W. Cain, B. Frey, D. Williams, M. M. Michael, C. May και H. Le, «Robust Architectural Support for Transactional Memory in the Power Architecture», *In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, 225-236. DOI=10.1145/2485922.2485942 http://doi.acm.org/10.1145/2485922.2485942,* 2013.

[11] P. Damron, A. Fedorova και Y. Lev, «Hybrid Transactional Memory», *SIGPLAN Not. 41, 11 (October 2006), 336-346. DOI=10.1145/1168918.1168900 http://doi.acm.org/10.1145/1168918.1168900,* 2006.

[12] Y. Lev, M. Moir και D. Nussbaum, «PhTM: Phased Transactional Memory», *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf,* 2007.

[13] Intel, Intel® Architecture Instruction Set Extensions Programming Reference, 2012.

[14] Intel, «Using HLE and RTM with older compilers with tsx-tools», [Ηλεκτρονικό]. Available: https://software.intel.com/en-us/blogs/2013/05/20/using-hle-and-rtm-with-

older-compilers-with-tsx-tools.

[15] K. Nikas, N. Anastopoulos, G. Goumas και N. Koziris, «Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm», *In Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09). IEEE Computer Society, Washington, DC, USA, 388-395. DOI=10.1109/ICPP.2009.60 http://dx.doi.org/10.1109/ICPP.2009.60,* 2009.

[16] D. Dice, Y. Lev και V. J. Marathe, «Simplifying Concurrent Algorithms by Exploiting Hardware Transactional Memory», *In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures (SPAA '10). ACM, New York, NY, USA, 325-334. DOI=10.1145/1810479.1810537 http://doi.acm.org/10.1145/1810479.1810537,* 2010.

[17] S. Kang και D. A. Bader, «An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs», *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Raleigh, NC, February 14-18,* 2009.

[18] J. Chung, M. Dalton, H. Kannan και C. Kozyrakis, «Thread-Safe Dynamic Binary Translation using Transactional Memory», *In the Proc. of the 14th HPCA, Salt Lake City, UT, Feb,* 2008.

[19] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris και M. Valero, «Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server», *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009),* p. 25–34, 2009.

[20] M. Mehrara, J. Hao, P.-C. Hsu και S. Mahlke, «Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory», *In Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09). ACM, New York, NY, USA, 166-176. DOI=10.1145/1542476.1542495 http://doi.acm.org/10.1145/1542476.1542495,* 2009.

[21] T. Moreshet, R. I. Bahar και M. Herlihy, «Energy Reduction in Multiprocessor Systems UsingTransactional Memory», *In Proceedings of the 2005 international symposium on Low power electronics and design (ISLPED '05). ACM, New York, NY, USA, 331-334. DOI=10.1145/1077603.1077683 http://doi.acm.org/10.1145/1077603.1077683,* 2005.

[22] J. Walker, «Eternally Confuzzled», [Ηλεκτρονικό]. Available: http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx.

[23] A. Natarajan, L. H. Savoie και N. Mittal, «Concurrent Wait-Free Red Black Trees», Technical Report UTDCS-16-12, Department of Computer Science, The University of Texas at Dallas, 2012.

[24] S. Hanke, T. Ottmann και E. Soisalon-Soininen, «Relaxed Balanced Red-Black Trees», *In Proceedings of the Third Italian Conference on Algorithms and Complexity (CIAC '97), Gian Carlo Bongiovanni, Daniel P. Bovet, and Giuseppe Di Battista (Eds.). Springer-Verlag, London, UK, 193-204*, 1997.