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

Μελέτη και Αξιολόγηση του Hardware Transactional Memory για Παραλληλοποίηση Skip Lists

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΑΡΙΟΣ ΚΑΡΔΑΡΑΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος, 2017

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

Μελέτη και Αξιολόγηση του Hardware Transactional Memory για Παραλληλοποίηση Skip Lists

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΜΑΡΙΟΣ ΚΑΡΔΑΡΑΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Νοεμβρίου 2017.

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Αν. Καθηγητής Ε.Μ.Π

Αθήνα, Νοέμβριος, 2017

.....
Μάριος Καρδαράς

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

Περίληψη

Τα τελευταία χρόνια η ανάγκη για εργαλεία που διευκολύνουν τον παράλληλο προγραμματισμό έχει φέρει το Transactional Memory (TM) στο προσκήνιο. Το TM απλοποιεί την παραλληλοποίηση πολύπλοκων δομών δεδομένων, όπως Skip Lists, δέντρα, πίνακες κατακερματισμού, κλπ.

Στη παρούσα διπλωματική υλοποιούμε παράλληλες δομές δεδομένων Skip List χρησιμοποιώντας Hardware Transactional Memory (HTM). Τις μελετάμε πειραματικά και τις συγκρίνουμε με υλοποιήσεις που βασίζονται σε Lock-Based και Lock-Free τεχνικές. Από τη μελέτη αυτή βγάζουμε συμπεράσματα τόσο σε σχέση με την εφαρμογή διαφόρων τεχνικών HTM σε δομές δεδομένων και ουρές προτεραιότητας Skip List, όσο και για την επίδραση που έχουν ορισμένες παράμετροι όπως το Hyper-Threading ή ένα σύστημα NUMA στην λειτουργία HTM υλοποιήσεων. Τέλος, τα αποτελέσματα αναδεικνύουν πλεονεκτήματα και μειονεκτήματα της HTM εκδοχής σε σχέση με τις άλλες τεχνικές καθώς και περιοχές εφαρμογής όπου η HTM υλοποίηση υπερέχει έως και 250% σε απόδοση συγκριτικά με τις υπόλοιπες.

Λέξεις κλειδιά

Hardware Transactional Memory, Παράλληλες Δομές Δεδομένων, Ουρές Προτεραιότητας, Skip List, Spray List

Abstract

Recently, the need for tools that simplify concurrent programming has brought Transactional Memory (TM) to the forefront of concurrency control mechanisms. TM attempts to simplify the parallelization of complex data structures, such as Skip Lists, trees, hash tables etc.

In this thesis, we utilize Hardware Transactional Memory (HTM) to implement concurrent Skip Lists. Using extensive evaluation of multiple benchmarks, we analyze these implementations and compare them with alternatives based on lock-free and lock-based synchronization mechanisms. The results of our evaluation reveal advantages and disadvantages of the HTM version as well as areas of application were it outperforms other solutions by 250%. Furthermore, we make conclusions regarding the usage of various HTM techniques used to parallelize Skip List Data Structures and Priority Queues. Finally, we study the impact of Hyper-Threading and NUMA on HTM and propose solutions.

Key Words

Hardware Transactional Memory, Concurrent Data Structures, Priority Queues, Skip List, Spray List

Ευχαριστίες

Καταρχάς, θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου Νεκτάριο Κοζύρη, καθώς μέσα από τη διδασκαλία του με εισήγαγε στις περιοχές των Παράλληλων Συστημάτων και της Αρχιτεκτονικής Υπολογιστών και μου κέντρισε τον ενδιαφέρον για να ασχοληθώ μετέπειτα με αυτές.

Επίσης θα ήθελα να ευχαριστήσω τους καθηγητές μου, κ. Γκούμα και κ. Νίκα τόσο για την ποιοτική διδασκαλία τους όσο και για τις πολύτιμες συμβουλές που μου έχουν δώσει.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Υποψήφιο Διδάκτωρ Δημήτριο Σιακαβάρα για τη καθοδήγησή του κατά τη διάρκεια εκπόνησης της διπλωματικής, την προθυμία του να με βοηθήσει οποτεδήποτε χρειαζόμουν βοήθεια και το συνολικότερο ενδιαφέρον του. Το σίγουρο είναι ότι χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δε θα ήταν εφικτή.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου, τους γονείς μου για την αγάπη τους, την εμπιστοσύνη και την απεριόριστη στήριξη τους όλα αυτά τα χρόνια και τον αδερφό μου, ο οποίος από τη πρώτη στιγμή μέχρι και σήμερα αποτελεί το μεγαλύτερο πρότυπό μου.

Η διπλωματική αυτή αφιερώνεται στον πολυαγαπημένο μου γάτο Χουχού, που έφυγε πρόωρα από τη ζωή.

Μάριος Καρδαράς,
Νοέμβριος 2017

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η εξέλιξη της τεχνολογίας έχει οδηγήσει στην καθιέρωση πολυεπεξεργαστικών υπολογιστικών συστημάτων με τον αριθμό των διαθέσιμων πυρήνων να αυξάνεται συνεχώς. Παρ' όλα αυτά αποτελεί πραγματική πρόκληση για τους προγραμματιστές να δημιουργήσουν εφαρμογές που να εκμεταλλεύονται στο έπακρον αυτήν την υπολογιστική ισχύ καθώς θα πρέπει να εξασφαλίσουν τον έγκυρο συγχρονισμό παράλληλων προσβάσεων σε κοινά δεδομένα αλλά και την σωστή επικοινωνία μεταξύ των νημάτων εκτέλεσης (threads).

Η πιο διαδεδομένη μέθοδος συγχρονισμού είναι το *κλείδωμα* (*locking*). Πιο συγκεκριμένα, όταν ένα τμήμα κώδικα (critical section) απαιτούμε να εκτελείται από το πολύ ένα νήμα εκτέλεσης για λόγους εγκυρότητας, τότε κλειδώνοντας την περιοχή αυτή επιτρέπουμε μόνο σε ένα νήμα, αυτό που ανταγωνιζόμενο με άλλα αποκτά το κλειδί, να έχει πρόσβαση σε αυτό. Η πιο απλή προσέγγιση κλειδώματος είναι το λεγόμενο *coarse-grained* κλείδωμα με το οποίο κλειδώνουμε ολόκληρη τη δομή. Αυτός ο τρόπος είναι ιδιαίτερα απλός στην υλοποίηση αλλά υστερεί σημαντικά σε απόδοση καθώς τα νήματα σειριοποιούνται για την εκτέλεση μιας λειτουργίας. Συχνά χρησιμοποιείται μια πιο εξειδικευμένη προσέγγιση κλειδώματος, το λεγόμενο *fine-grained* κλείδωμα με το οποίο προσπαθούμε να ελαχιστοποιήσουμε τα μέρη της δομής που επιχειρούμε να κλειδώσουμε. Με τον τρόπο αυτό επιτυγχάνουμε καλύτερη απόδοση καθώς αυξάνουμε τον παραλληλισμό αλλά είναι προγραμματιστικά πιο δύσκολο και χρειάζεται ιδιαίτερη προσοχή για την επίτευξη εγκυρότητας. Γενικά, ενώ η μέθοδος κλειδώματος προσφέρει αρκετές επιλογές στον προγραμματιστή για την επίτευξη παραλληλισμού θα πρέπει να λάβει υπόψη του ορισμένα προβλήματα που παρατηρούνται σε lock-based εφαρμογές όπως :

- *το αδιέξοδο* (*deadlock*), το οποίο προκύπτει όταν δύο ή περισσότερα νήματα δημιουργούν μια κυκλική αλυσίδα όπου το κάθε νήμα για να συνεχίζει περιμένει να ελευθερωθεί ένα κλείδωμα το οποίο έχει καταληφθεί από το επόμενο νήμα... π.χ. **Α**→Β→Γ→**Α**
- *τη λιμοκτονία* (*starvation*), όπου ένα ή περισσότερα νήματα περιμένουν επ' αόριστον να ελευθερωθεί ένα κλείδωμα, καθώς άλλα "άπληστα" νήματα το καταλαμβάνουν ανελλιπώς.
- *το ενεργό αδιέξοδο* (*livelock*), όπου τα νήματα εκτελούν συνεχώς το ίδιο τμήμα κώδικα και καταλήγουν διαρκώς στην ίδια κατάσταση αναστολής.
- *την έλλειψη ανθεκτικότητας* (*lack of robustness*), όπου ένα νήμα θα μπορούσε για παράδειγμα να καταργηθεί από το λειτουργικό σύστημα ενώ άλλα περιμένουν να ελευθερώσει κάποιο κλείδωμα με αποτέλεσμα να "κολλήσει" το πρόγραμμα.

- *την αντιστροφή προτεραιότητας* (*priority inversion*), όπου κάποιο νήμα υψηλότερης προτεραιότητας είναι υποχρεωμένο να περιμένει ένα άλλο χαμηλότερης να ελευθερώσει κάποιο κλείδωμα.
- *τη δημιουργία κονβόι λόγω κλειδώματος* (*lock convoying*), ένα πρόβλημα που θα παρατηρηθεί αρκετά στη συνέχεια της διπλωματικής, όπου ένα νήμα αναστέλλεται από το λειτουργικό σύστημα μέσω ενός context switch ενώ κρατάει κάποιο κλείδωμα με αποτέλεσμα άλλα που παίρνουν την θέση του και περιμένουν να το ελευθερώσει να μην μπορούν να προχωρήσουν.

Μια άλλη μέθοδος συγχρονισμού που δεν πάσχει από αυτά τα προβλήματα και είναι ιδιαίτερα αποδοτική είναι η *lock-free* μέθοδος η οποία χρησιμοποιεί εντολές υποστηριζόμενες από το hardware (π.χ. compare_and_swap), οι οποίες διασφαλίζουν ατομική πρόσβαση σε δεδομένα. Ως αποτέλεσμα τα νήματα δεν μπλοκάρουν το ένα το άλλο, συνεπώς η εκτέλεσή τους είναι χωρίς διακοπές (non-blocking). Ο προγραμματιστής που θέλει να υλοποιήσει lock free εφαρμογές θα πρέπει να λάβει υπόψη του πολλές λεπτομέρειες κυρίως ως προς τη διασφάλιση εγκυρότητας γεγονός που καθιστά τη μέθοδο αρκετά δύσκολη.

Στην διπλωματική αυτή θα πειραματιστούμε με *Transactional Memory* [1], ένα μηχανισμό συγχρονισμού που έχει προταθεί με σκοπό να λύσει ορισμένα από τα προβλήματα των lock-based και lock-free τεχνικών και χρησιμοποιεί *transactions* για τον συγχρονισμό παράλληλων προσβάσεων σε κοινή μνήμη. Συνήθως ο όρος transaction (δοσοληψία) είναι συσχετισμένος με βάσεις δεδομένων. Εδώ, ας σκεφτούμε το transaction ως μια λειτουργία ατομικής ενημέρωσης ενός συνόλου θέσεων μνήμης, η οποία εκτελείται από ένα νήμα. Πιο συγκεκριμένα, ο προγραμματιστής επισημαίνει συγκεκριμένα τμήματα κώδικα, τα οποία πρέπει να εκτελεστούν ατομικά. Το TM σύστημα αναλαμβάνει την εκτέλεση αυτών των τμημάτων ως *transactions*. Κάθε transaction έχει δύο πιθανές εκβάσεις:

- *οριστικοποίηση* (*commit*) : το transaction ολοκληρώνεται και όλες οι αλλαγές που έγιναν εντός transaction γίνονται ατομικά ορατές στα υπόλοιπα νήματα.
- *αναίρεση* (*abort*) : όλες οι αλλαγές που έγιναν εντός του transaction μέχρι εκείνη τη στιγμή απορρίπτονται. Για παράδειγμα, εάν ανιχνευτεί κάποια σύγκρουση (conflict) (Read-Write , Write-Write), τότε ένα ή περισσότερα transactions που συσχετίζονται με το conflict θα απορρίψουν την λειτουργία τους.

Σημειώνουμε ότι μέχρι να γίνει commit το transaction οι αλλαγές που γίνονται εντός του δεν είναι ορατές στα υπόλοιπα νήματα ή στη μνήμη. Για να επιτευχθεί η λειτουργία του TM κάθε transaction κρατά ένα *read* και ένα *write set* τα οποία ενημερώνονται κάθε φορά που συναντάει μια εντολή read ή write αντίστοιχα. Αυτά τα δύο set αποτελούν το *αποτύπωμα* (*footprint*) του transaction. Κύρια επιδίωξη του Transactional Memory είναι η επίτευξη προγραμματιστικής ευκολίας ανάλογης με εκείνη του coarse-grained locking και επίδοσης ανάλογης με εκείνη του fine-grained locking.

Τα συστήματα TM χωρίζονται σε *Software Transactional Memory* (*STM*) και σε *Hardware Transactional Memory* (*HTM*). Τα STM συστήματα δεν αντιμετωπίζουν περιορισμούς ως προς την ενσωμάτωσή τους στα υπάρχοντα υπολογιστικά συστήματα αλλά λόγο του ότι χρησιμοποιούν software για την επίβλεψη αλλαγών στην μνήμη και τον

εντοπισμό conflicts υστερούν σημαντικά σε επίδοση σε σχέση με τα HTM συστήματα. Αντίθετα το HTM βασίζεται στο hardware για τον εντοπισμό conflicts γεγονός που το καθιστά αποδοτικότερα. Ο προγραμματιστής που θέλει να χρησιμοποιήσει HTM θα πρέπει όμως να λάβει υπόψη του ότι το footprint του transaction είναι περιορισμένο λόγω της δεδομένης χωρητικότητας των hardware buffers που αποθηκεύουν το read και το write set καθώς επίσης ότι δεν υπάρχει καμία εγγύηση ότι ένα transaction θα ολοκληρώσει εν τέλει τη λειτουργία του καθώς μπορεί να δεχθεί κάποιο abort και σε περιπτώσεις όπως interrupt ή page fault. Κατά συνέπεια πρέπει να εξασφαλίσει μια διέξοδο εκτός transaction που να αντιμετωπίζει την περίπτωση των πολλαπλών aborts. Η διέξοδος αυτή είναι το λεγόμενο *fallback path* στο οποίο εκτελείται κώδικας διαχείρισης των aborts και συνήθως χρησιμοποιείται ένα καθολικά προσβάσιμο κλειδί, η απόκτηση του οποίου οδηγεί σε σειριοποίηση όλων τα transactions.

Υπάρχουν διάφοροι τύποι επεξεργαστών που υποστηρίζουν HTM [2],όπως

- Rock processor (ακυρώθηκε από την Oracle).
- Blue Gene/Q processor της IBM (Sequoia supercomputer).
- IBM zEnterprise EC12, ο πρώτος εμπορικός server που πρόσθεσε transactional memory εντολές στους επεξεργαστές του.
- Intel's Transactional Synchronization Extensions (TSX), διαθέσιμος σε Haswell-based επεξεργαστές και σε νεότερους [3].
- IBM POWER8 και νεότερους .

Στη διπλωματική αυτή χρησιμοποιούμε Intel Haswell-EP επεξεργαστές οι οποίοι έχουν προσθέσει στην αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture, ISA) τους εντολές για Transactional Memory, η κατανόηση των οποίων είναι σημαντική για τη συνέχεια, συνεπώς θα δώσουμε μια επεξήγηση στο επόμενο κεφάλαιο.

Μια από τις βασικότερες εφαρμογές των μεθόδων συγχρονισμού είναι οι δομές δεδομένων. Στη παρούσα διπλωματική ασχολούμαστε με τη δομή δεδομένων *Skip List* [4]. Η Skip List αποτελεί μια ενδιαφέρουσα εναλλακτική επιλογή αντί για ισοσταθμισμένα δέντρα (balanced trees) κυρίως σε εφαρμογές βάσεων δεδομένων όπως για παράδειγμα η MemSQL, καθώς προσφέρει λογαριθμική πολυπλοκότητα, διατηρεί τη διάταξη των δεδομένων της και δεν απαιτεί ανακατανομές (rebalancing) κάτι που οδηγεί σε στενωπό απόδοσης λόγω συγκρούσεων (contention bottlenecks). Λεπτομέρειες σχετικά με τη δομή δεδομένων δίνουμε στο επόμενο κεφάλαιο.

1.1 Κίνητρο

Η επιλογή χρήσης Hardware Transactional Memory υπήρξε μέχρι πρόσφατα περισσότερο επιλογή για προγραμματιστική ευκολία παρά επιλογή για επίτευξη μέγιστης απόδοσης. Επιπλέον, τα περισσότερα εμπορικά συστήματα χρησιμοποιούν lock-based και lock-free τεχνικές για την επίτευξη συγχρονισμού. Τα τελευταία χρόνια όμως, ανακαλύπτουμε δυνατότητες που παρέχει η HTM αρχιτεκτονική που μας επιτρέπουν να αναπτύξουμε αλγορίθμους οι οποίοι πολλές φορές οδηγούν σε υλοποιήσεις που υπερέχουν σε απόδοση συγκριτικά με τις αντίστοιχες lock-based και lock-free. Για παράδειγμα,

πρόσφατα αναπτύχθηκε τεχνική που συνδυάζει *HTM* και *Read-Copy-Update (RCU)* [5] για την υλοποίηση ισοσταθμισμένων Binary Search Trees όπως για παράδειγμα AVL, Red-Black-Trees, B-Trees , η οποία σε πειράματα δείχνει ότι υπερέχει από 70% έως 220% σε απόδοση συγκριτικά με τις υπάρχουσες υλοποιήσεις. Ένα άλλο παράδειγμα είναι η εφαρμογή μιας τεχνικής fine-grained HTM προγραμματισμού που χρησιμοποιεί *consistency-oblivious programming (COP)* [6], η οποία δοκιμασμένη σε BST δείχνει μεγάλη βελτίωση σε απόδοση σε σχέση με τις απλούστερες HTM τεχνικές με απαιτούμενη προγραμματιστική δυσκολία αντίστοιχης του fine-grained locking. Συνεπώς η χρήση HTM διαφαίνεται ιδιαίτερα υποσχόμενη σε μελλοντικές εφαρμογές και η μελέτη της σε διάφορες δομές δεδομένων είναι ιδιαίτερα χρήσιμη. Επιπλέον, στη παρούσα βιβλιογραφία δεν υπάρχει εκτενής μελέτη εφαρμογής HTM σε Skip Lists ανάλογης με εκείνες που υπάρχουν για lock-based και lock-free Skip Lists, γεγονός που αποτέλεσε κίνητρο ώστε να ασχοληθούμε εμείς. Τέλος, στα πλαίσια του μαθήματος Προηγμένα Θέματα Αρχιτεκτονικής Υπολογιστών η ενασχόληση με ένα project όπου εφαρμόσαμε HTM στην προσεγγιστική ουρά προτεραιότητας Spray List, είχε ως αποτέλεσμα την εξαγωγή ορισμένων αξιόλογων συμπερασμάτων και αποτέλεσε το έναυσμα για την περαιτέρω ενασχόληση με Skip Lists.

1.2 Στόχοι διπλωματικής

Οι στόχοι τη διπλωματικής είναι οι εξής:

1) η υλοποίηση και αξιολόγηση διαφόρων τεχνικών HTM σε Skip Lists και ουρές προτεραιότητας.
2) η μελέτη της επίδρασης που έχουν ορισμένες παράμετροι όπως το hyper-threading ή ένα σύστημα NUMA στην λειτουργία HTM υλοποιήσεων.
3) η σύγκριση των HTM υλοποιήσεων με τις state-of-the-art που βασίζονται σε Lock-Based και Lock-Free αλγορίθμους και την εξαγωγή συμπερασμάτων.

1.3 Διάθρωση κειμένου

Η δομή των κεφαλαίων είναι η ακόλουθη:

Κεφάλαιο 2 - Θεωρητικό υπόβαθρο : Στο κεφάλαιο αυτό αρχικά αναπτύσσουμε ορισμένες έννοιες γύρω από τη συμπεριφορά παράλληλων αντικειμένων. Στη συνέχεια περιγράφουμε τις εντολές που επεκτείνουν το ISA του Haswell επεξεργαστή και επιτρέπουν την χρησιμοποίηση Hardware Transactional Memory καθώς και ορισμένα χαρακτηριστικά του HTM που θεωρούμε ότι είναι σκόπιμο να γνωρίζει ο αναγνώστης. Τέλος παρουσιάζουμε τις απαραίτητες λεπτομέρειες της δομής δεδομένων Skip List.

Κεφάλαιο 3 - Lock-Based και Lock-Free Skip Lists : Στο κεφάλαιο αυτό περιγράφουμε τους αλγόριθμους που στοχεύουν στη παράλληλη προσπέλαση και επεξεργασία δεδομένων της δομής δεδομένων Skip List χρησιμοποιώντας Lock-Based και Lock-Free τεχνικές έτσι όπως αναφέρονται στην βιβλιογραφία [7].

Κεφάλαιο 4 - Υλοποίηση Skip List χρησιμοποιώντας Hardware Transactional Memory : Στο κεφάλαιο αυτό παρουσιάζουμε την HTM εκδοχή που έχουμε υλοποιήσει. Περιγράφουμε λεπτομερώς τους αλγορίθμους που χρησιμοποιήσαμε, αιτιολογούμε την εγκυρότητά τους και παρέχουμε μερικές ιδέες για βελτιστοποιήσεις.

Κεφάλαιο 5 - Αξιολόγηση αλγορίθμων : Στο κεφάλαιο αυτό αξιολογούμε τους αλγόριθμους που περιγράψαμε στα προηγούμενα κεφάλαια. Παρουσιάζουμε και σχολιάζουμε πειραματικά αποτελέσματα και βγάζουμε συμπεράσματα.

Κεφάλαιο 6 - Skip List - Ουρές προτεραιότητας : Στο κεφάλαιο αυτό αρχικά περιγράφουμε τις σημαντικότερες ουρές προτεραιότητας Skip List που υπάρχουν στην βιβλιογραφία και στη συνέχεια ασχολούμαστε με την προσεγγιστική ουρά προτεραιότητας Spray List όπου παρουσιάζουμε τον αλγόριθμο καθώς και την HTM εκδοχή που έχουμε υλοποιήσει.

Κεφάλαιο 7 - Επίλογος : Στο κεφάλαιο αυτό παρουσιάζουμε ορισμένα γενικότερα συμπεράσματα που βγάλαμε από τη διπλωματική και αναφέρουμε ορισμένες μελλοντικές επεκτάσεις που έχουν ενδιαφέρον.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτό αρχικά αναπτύσσουμε ορισμένες έννοιες γύρω από τη συμπεριφορά παράλληλων αντικειμένων, οι οποίες θα μας βοηθήσουν στη συνέχεια της διπλωματικής να κατανοήσουμε καλύτερα την συμπεριφορά των παράλληλων υλοποιήσεων που θα συναντήσουμε. Στη συνέχεια περιγράφουμε τις εντολές που επεκτείνουν το Instruction Set Architecture του Haswell επεξεργαστή και επιτρέπουν την χρησιμοποίηση Hardware Transactional Memory καθώς και ορισμένα χαρακτηριστικά του HTM που θεωρούμε ότι είναι σκόπιμο να γνωρίζει ο αναγνώστης. Τέλος, παρουσιάζουμε τη δομή δεδομένων Skip List. Για περισσότερες λεπτομέρειες σχετικά με HTM παραπέμπουμε τον αναγνώστη στη σχετική βιβλιογραφία [1], [3].

2.1 Συμπεριφορά παράλληλων αντικειμένων

Η συμπεριφορά παράλληλων αντικειμένων περιγράφεται καλύτερα μέσα από τις *ιδιότητες εγκυρότητας (correctness)* και *προόδου (progress)* που ικανοποιούν.

2.1.1 Εγκυρότητα

Είναι σχετικά εύκολο να προσδιορίσουμε τη σωστή λειτουργία ενός αντικειμένου, όπως για παράδειγμα μιας FIFO ουράς, εφόσον δεν υπάρχει παραλληλισμός. Αυτό που αρκεί να κάνουμε είναι να προσδιορίσουμε ποιές ακολουθίες εντολών είναι έγκυρες για το συγκεκριμένο αντικείμενο.

Ας δούμε το παράδειγμα της FIFO ουράς. Έστω q μια ουρά και σ_q η κατάσταση της ουράς, π.χ. σ_q={4,2,81,7} όπου head=4, tail=7 ή σ_q={} η κενή ουρά. Έστω q.enq(x) η εντολή που θέτει την κατάσταση της ουράς από σ_q σε σ_q.x και q.dec() η εντολή που θέτει την κατάσταση της ουράς από y.σ_q σε σ_q και επιστρέφει y με τη προϋπόθεση ότι η ουρά δεν είναι κενή. Με βάση τα παραπάνω ορίζουμε τους εξής δυο κανόνες που καθορίζουν ποιές ακολουθίες εντολών είναι έγκυρες για μια FIFO ουρά:

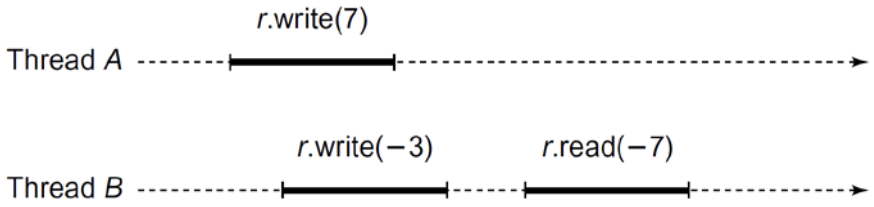
1) Για κάθε εντολή q.dec() που επιστρέφει x, θα πρέπει να έχει προηγηθεί μια αντίστοιχη εντολή q.enc(x).
2) Η ακολουθία των q.dec() εντολών πρέπει να αποτελεί πρόθεμα της ακολουθίας q.enc() εντολών.

Όταν υπεισέρχεται παραλληλισμός όμως τα πράγματα δυσκολεύουν καθώς πλέον η σειρά εκτέλεσης των εντολών δεν είναι ιδιαίτερα σαφής: Τα διαστήματα εκτέλεσης των εντολών μπορεί να επικαλύπτονται. Όταν δύο νήματα καλούν περίπου την ίδια χρονική

στιγμή εντολές που επιδρούν στην ίδια δομή τότε είναι πολύ πιθανόν οι εντολές αυτές να επικαλύπτονται χρονικά η μια με την άλλη με αποτέλεσμα να μην είναι σαφές ποιά εντολή επενέργησε πρώτη και ποιά δεύτερη στη δομή.

Συνεπώς η έννοια της εγκυρότητας είναι σχετική σε παράλληλα συστήματα και το πόσο ισχυρή ή αδύναμη είναι εξαρτάται από το πόσο απέχει η συμπεριφορά του παράλληλου συστήματος από τη συμπεριφορά του αντίστοιχου ιδανικού σειριακού συστήματος. Στη συνέχεια θα εξετάσουμε τρείς έννοιες εγκυρότητας παράλληλων αντικειμένων: *την συνέπεια ηρεμίας* (*quiescent consistency*), *την σειριακή συνέπεια* (*sequential consistency*) και *την γραμμικοποίηση* (*linearizability*).

2.1.1.1 Quiescent Consistency



Σχήμα 2.1 *Παράδειγμα εκτέλεσης εντολών ανάγνωσης και εγγραφής σε καταχωρητή. Οι διακεκομμένες γραμμές παριστάνουν τον χρόνο και οι μπάρες αντιστοιχούν στα χρονικά διαστήματα εκτέλεσης της εκάστοτε εντολής.*

Παρατηρώντας το σχήμα 2.1 βλέπουμε δύο νήματα να καταχωρούν ταυτόχρονα τις τιμές -3 και 7 στο κοινό καταχωρητή r (η εντολή r.read(x) σημαίνει ότι το νήμα διαβάζει την τιμή x από τον καταχωρητή r, αντίστοιχα γράφει για r.write(x)). Στη συνέχεια ένα νήμα διαβάζει τον r και επιστρέφει -7. Αυτή η συμπεριφορά είναι προφανώς μη αποδεχτή. Θα περιμέναμε να διαβάσουμε είτε 7 είτε -3 από τον καταχωρητή και όχι μια μίξη και των δύο τιμών. Το παράδειγμα αυτό υποδεικνύει τη σύσταση του ακόλουθου αξιώματος.

Αξίωμα 3.1. Οι λειτουργίες πρέπει να εμφανίζεται ότι συμβαίνουν με κάποια διαδοχική σειρά.

Από μόνο του αυτό το αξίωμα είναι συνήθως αρκετά αδύναμο για να είναι χρήσιμο. Για παράδειγμα επιτρέπει στα reads να επιστρέφουν πάντα την αρχική κατάσταση του αντικειμένου, ακόμα και σε σειριακές εκτελέσεις.

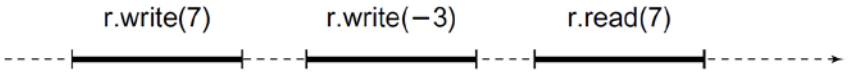
Ακολουθεί ένα ισχυρότερο αξίωμα.

Αξίωμα 3.2. Οι λειτουργίες που διαχωρίζονται από μια περίοδο ηρεμίας (period of quiescence) πρέπει να εμφανίζεται ότι λαμβάνουν χώρα σύμφωνα με τη πραγματική σειρά εμφάνισής τους στον χρόνο.

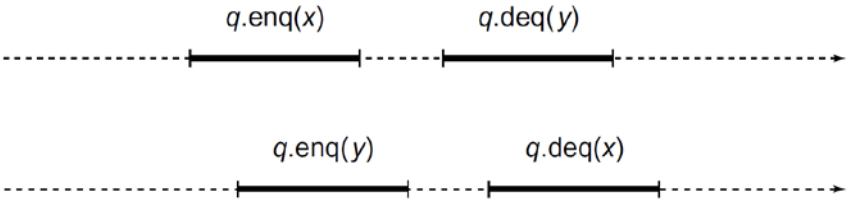
Ένα αντικείμενο είναι σε quiescent κατάσταση εάν δεν υπάρχει κάποια λειτουργία σε ισχύ. Για παράδειγμα, ας υποθέσουμε ότι τα νήματα A και B εισάγουν ταυτόχρονα τις τιμές x και y σε μια FIFO ουρά. Ύστερα η ουρά γίνεται quiescent και μετά το νήμα Γ εισάγει την τιμή z στην ουρά. Σύμφωνα με το αξίωμα 3.2 δεν μπορούμε να προβλέψουμε την σχετική σειρά των x και y στην ουρά αλλά ξέρουμε με βεβαιότητα ότι προηγούνται του z.

Μαζί, τα αξιώματα 3.1 και 3.2 ορίζουν μια ιδιότητα εγκυρότητας που ονομάζεται quiescent consistency. Με άλλα λόγια η ιδιότητα αυτή μας λέει ότι μεταξύ δύο διαδοχικών quiescent περιόδων η σειρά εκτέλεσης των εντολών ισοδυναμεί με κάποια σειριακή εκτέλεση των ίδιων εντολών.

2.1.1.2 Sequential Consistency



Σχήμα 2.2 *Παράδειγμα εκτέλεσης εντολών ανάγνωσης και εγγραφής σε καταχωρητή.*



Σχήμα 2.3 *Παράδειγμα εκτέλεσης εντολών σε FIFO ουρά.*

Στο σχήμα 2.2 ένα νήμα γράφει σε έναν καταχωρητή r την τιμή 7, ύστερα την τιμή -3 και στο τέλος διαβάζει τον καταχωρητή και επιστρέφει 7. Για μερικές εφαρμογές αυτή η συμπεριφορά μπορεί να μην είναι αποδεκτή γιατί η τιμή που διαβάστηκε δεν είναι η τελευταία τιμή που γράφτηκε. Η σειρά με την οποία ένα νήμα εκτελεί εντολές ονομάζεται ροή προγράμματος.

Σε αυτό το παράδειγμα μας φάνηκε περίεργο που οι εντολές δεν επενέργησαν σύμφωνα με τη ροή προγράμματος. Το παράδειγμα αυτό υποδεικνύει τη σύσταση του ακόλουθου αξιώματος.

Αξίωμα 3.3. Οι λειτουργίες πρέπει να εμφανίζεται ότι συμβαίνουν σύμφωνα με τη ροή προγράμματος.

Το αξίωμα αυτό μαζί με το αξίωμα 3.1 ορίζουν μια ιδιότητα εγκυρότητας που ονομάζεται sequential consistency.

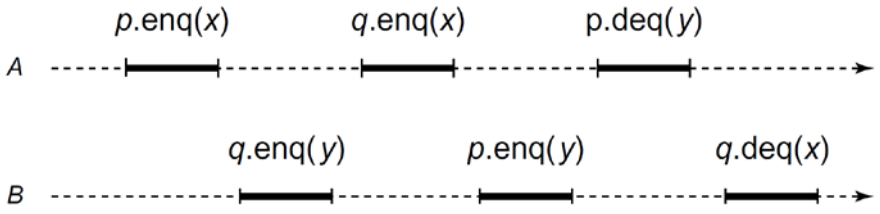
Η ιδιότητα αυτή απαιτεί ότι οι εντολές να δρουν σαν να έχουν εκτελεστεί με κάποια σειριακή σειρά που σέβεται τη ροή προγράμματος. Με άλλα λόγια σε κάθε παράλληλη εκτέλεση υπάρχει τρόπος να διατάξουμε σειριακά τις εντολές έτσι ώστε (1) να είναι συνεπείς με τη ροή προγράμματος και (2) να συμφωνούν με τις σειριακές προδιαγραφές του αντικειμένου. Μπορεί να υπάρχουν παραπάνω από μια διατάξεις εντολών που είναι σύμφωνες με την ιδιότητα αυτή. Για παράδειγμα στο σχήμα 2.3 το νήμα A εισάγει σε μια FIFO ουρά q την τιμή x την ίδια χρονική περίοδο που ο B εισάγει το y στην q. Στη συνέχεια ο A επιστρέφει την τιμή y ύστερα από εκτέλεση της εντολής αφαίρεσης στοιχείου από την ουρά ο B επιστρέφει τη τιμή x. Υπάρχουν δυο πιθανές διατάξεις εντολών συμβατές με την ιδιότητα sequential consistency που θα μπορούσαν να δικαιολογήσουν την συμπεριφορά αυτή:

1) $\langle q.enq(x)\ A\rangle \rightarrow \langle q.enq(y)\ B\rangle \rightarrow \langle q.dec(x)\ B\rangle \rightarrow \langle q.dec(y)\ A\rangle$

2) $\langle q.enq(y)\ B\rangle \rightarrow \langle q.enq(x)\ A\rangle \rightarrow \langle q.dec(y)\ A\rangle \rightarrow \langle q.dec(x)\ B\rangle$

Στο σημείο αυτό να τονίσουμε ότι οι ιδιότητες sequential consistency και quiescent consistency είναι ασυσχέτιστες. Η ιδιότητα quiescent consistency δεν διατηρεί κατ' ανάγκην την ροή του προγράμματος και η ιδιότητα sequential consistency δεν επηρεάζεται από quiescent περιόδους. Αντίθετα η ιδιότητα linearizability που θα δούμε στη συνέχεια είναι ισχυρότερη και των δύο.

Ένα αρνητικό στοιχείο του sequential consistency είναι ότι δεν είναι compositional[1]. Μια ιδιότητα εγκυρότητας $\mathcal{P}$ είναι compositional όταν κάθε αντικείμενο του συστήματος ικανοποιεί την ιδιότητα $\mathcal{P} \Rightarrow$ το σύστημα συνολικά την ικανοποιεί.



Σχήμα 2.4 *Παράδειγμα εκτέλεσης εντολών σε δύο FIFO ουρές.*

Στο σχήμα 2.4 βλέπουμε ένα παράδειγμα που παρόλο οι FIFO ουρές p και q ικανοποιούν την ιδιότητα εγκυρότητας sequential consistency το σύστημα εξολοκλήρου δεν την ικανοποιεί. Ας δούμε γιατί:

Η ουρά p είναι FIFO και το νήμα A εξάγει το y από την ουρά p συνεπώς το y θα πρέπει να έχει εισαχθεί πριν το x.

$$\langle p.enq(y)\ B\rangle \rightarrow \langle p.enq(x)\ A\rangle$$

Αντίστοιχα,

$$\langle q.enq(x)\ A\rangle \rightarrow \langle q.enq(y)\ B\rangle$$

Η ροή προγράμματος όμως απαιτεί ότι

$$\langle p.enq(x)\ A\rangle \rightarrow \langle q.enq(x)\ A\rangle \text{ και } \langle q.enq(y)\ B\rangle \rightarrow \langle p.enq(y)\ B\rangle$$

Οι διατάξεις αυτές δημιουργούν τον κύκλο

$$\langle p.enq(x)\ A\rangle \rightarrow \langle q.enq(x)\ A\rangle \rightarrow \langle q.enq(y)\ B\rangle \rightarrow \langle p.enq(y)\ B\rangle \rightarrow \langle p.enq(x)\ A\rangle$$

συνεπώς το σύστημα εξολοκλήρου δεν είναι sequential consistent.

[1] Αντίθετα η ιδιότητα quiescent consistency είναι compositional. Για την απόδειξη παραπέμπουμε τον αναγνώστη στη σχετική βιβλιογραφία [7].

2.1.1.3 Linearizability

Προκειμένου να κάνουμε πιο αυστηρές τις συνθήκες εγκυρότητας και να απαιτήσουμε compositionality ας αντικαταστήσουμε το αξίωμα 3.3 με το ισχυρότερο:

Αξίωμα 3.4. Οι λειτουργίες πρέπει να εμφανίζεται ότι λαμβάνουν χώρα στιγμιαία κάποια στιγμή ανάμεσα στην κλήση και στην επιστροφή τους.

Αυτό το αξίωμα μας λέει ότι αν μια λειτουργία o_1 επιδρά στο σύστημα πριν από μια λειτουργία o_2 τότε η o_1 προηγείται της o_2 σε πραγματικό χρόνο. Ονομάζουμε αυτή την ιδιότητα εγκυρότητας linearizability.

Ο συνήθης τρόπος για να δείξουμε ότι η υλοποίηση ενός παράλληλου αντικειμένου είναι linearizable είναι να εντοπίσουμε σε κάθε λειτουργία ένα linearization point, ένα σημείο δηλαδή στο οποίο γίνεται εμφανής η επίδραση της λειτουργίας στο σύστημα. Για παράδειγμα, σε lock-based υλοποιήσεις τα linearization points μπορεί να είναι τα κρίσιμα τμήματα. Σε lock-free υλοποιήσεις τα linearization points είναι συνήθως κάποιο συγκεκριμένο τμήμα κώδικα, π.χ. μια ατομική εντολή, όπου η δράση της λειτουργίας γίνεται φανερή στις υπόλοιπες λειτουργίες.

Υπάρχουν και πιο "επίσημοι" τρόποι να αποδειχθεί το linearizability τους οποίους όμως δεν θα αναλύσουμε εδώ καθώς δεν θα χρειαστούν στην παρούσα διπλωματική, παραπέμπουμε πάντως τον αναγνώστη στη σχετική βιβλιογραφία [7].

2.1.2 Πρόοδος

Όπως είπαμε στην αρχή της ενότητας η συμπεριφορά παράλληλων αντικειμένων περιγράφεται καλύτερα μέσα από τις ιδιότητες εγκυρότητας (correctness) και προόδου (progress) που ικανοποιούν.

Ας δούμε μερικές ιδιότητες προόδου που είναι χρήσιμο να γνωρίζουμε για την συνέχεια.

- ιδιότητες **blocking** και **nonblocking**: Μια υλοποίηση είναι blocking εάν η καθυστέρηση ενός νήματος μπορεί να οδηγήσει σε καθυστέρηση και άλλων νημάτων ενώ nonblocking είναι οι υλοποιήσεις στις οποίες η καθυστέρηση ενός νήματος δεν μπορεί να οδηγήσει σε καθυστέρηση κάποιο άλλο νήμα. Χαρακτηριστικό παράδειγμα blocking υλοποιήσεων είναι οι lock-based υλοποιήσεις.
- Μια λειτουργία είναι **wait-free** εάν εγγυάται ότι κάθε κλήση θα ολοκληρώσει την εκτέλεσή της σε πεπερασμένο αριθμό βημάτων. Εάν υπάρχει κάποιο ανώτατο όριο στον αριθμό βημάτων τότε λέμε ότι η λειτουργία είναι **bounded wait-free**. Μια υλοποίηση είναι wait-free αν όλες οι λειτουργίες της είναι wait-free. Ισχύει ότι μια wait-free υλοποίηση είναι non-blocking ενώ το αντίστροφο δεν ισχύει. Για παράδειγμα σε lock-based υλοποιήσεις ένα νήμα μπορεί εκτελέσει έναν απροσδιόριστο αριθμό ανεπιτυχών προσπαθειών ούτως ώστε να καταλάβει ένα κλείδωμα.
- Μια λειτουργία είναι **lock-free** εάν εγγυάται ότι απείρως συχνά κλήσεις της λειτουργίας ολοκληρώνονται σε πεπερασμένο αριθμό βημάτων.

Ισχύει ότι wait-free ⇒ lock-free, αλλά όχι το αντίστροφο. Οι lock-free αλγόριθμοι ενέχουν την πιθανότητα ορισμένα νήματα να λιμοκτονήσουν.

- Μια λειτουργία είναι **obstruction-free** εάν από οποιοδήποτε σημείο, μετά το οποίο εκτελείται απομονωμένη, τερματίζει σε πεπερασμένο αριθμό βημάτων. Η ιδιότητα αυτή είναι πιο αδύναμη της lock-free ιδιότητας. Lock-free ⇒ obstruction free, αλλά όχι το αντίστροφο.

Ποιές ιδιότητες προόδου και ποιές ιδιότητες εγκυρότητας είναι κατάλληλες για ένα παράλληλο σύστημα? Η απάντηση είναι ότι εξαρτάται από τις ανάγκες και τη φύση του συστήματος. Για παράδειγμα η ιδιότητα quiescent consistency είναι κατάλληλη για εφαρμογές που απαιτούν υψηλή επίδοση με αντάλλαγμα τους αδύναμους περιορισμούς στη συμπεριφορά των διαφόρων αντικειμένων που απαρτίζουν το σύστημα. Η ιδιότητα sequential consistency είναι συχνά χρήσιμη για να περιγράψει χαμηλότερου επιπέδου συστήματα όπως διασυνδέσεις σε μνήμη υλικού. Η ιδιότητα linearizability είναι χρήσιμη για την περιγραφή υψηλότερου επιπέδου συστημάτων, τα οποία αποτελούνται από linearizable επιμέρους υποσυστήματα.

2.2 Hardware Transactional Memory

2.2.1 Intel Transactional Synchronization Extension

Η επέκταση Intel Transactional Synchronization Extension (TSX) προσφέρει τα παρακάτω δυο σύνολα εντολών :

- *To Hardware Lock Elision (HLE)* που είναι μια επέκταση εντολών συμβατή με παλαιότερα συστήματα (αποτελούμενη από τα προθέματα *XACQUIRE* και *XRELEASE*).
- *To Restricted Transactional Memory (RTM)* που είναι μια νέα διεπαφή εντολών (αποτελούμενη από τις εντολές *XBEGIN*, *XEND* και *XABORT*).

Και στις δυο διεπαφές όταν ένα νήμα ξεκινάει να εκτελεί κώδικα κάποιου transaction αρχικά διαβάζει ένα καθολικά προσβάσιμο κλειδί. Αν κάποια στιγμή κάποιο άλλο νήμα καταλάβει το κλειδί, δηλαδή γράψει στη διεύθυνση μνήμης που είναι αποθηκευμένο το κλειδί τότε, όπως αναφέραμε στην εισαγωγή, λόγω του read-write conflict το πρώτο νήμα θα απορρίψει τη λειτουργία του. Το HLE επιτρέπει την εκτέλεση του critical section μία φορά με speculation ,δηλαδή διαβάζει το κλειδί χωρίς να το καταλαμβάνει και αν αποτύχει εκτελείται τη δεύτερη φορά αφού πρώτα καταλάβει το κλειδί. Ένα πρόγραμμα το οποίο χρησιμοποιεί το HLE είναι συμβατό και μπορεί να εκτελεστεί και σε επεξεργαστές που δεν υποστηρίζουν TSX. Σε αυτή την περίπτωση το critical section εκτελείται κατευθείαν σε lock mode. Το RTM δεν προσφέρει αυτή την συμβατότητα με προηγούμενες γενιές επεξεργαστών αλλά προσφέρει μεγαλύτερη ευελιξία ως προς τις ενέργειες που μπορούν να γίνουν ύστερα από κάποιο transactional abort. Ο προγραμματιστής ορίζει μία διεύθυνση μνήμης στην οποία βρίσκεται ο κώδικας ο οποίος θα εκτελεστεί σε περίπτωση abort (fallback handler).

Στη συνέχεια επικεντρωνόμαστε στην RTM διεπαφή όπως περιγράφεται από τα intrinsics του gcc καθώς είναι αυτή που χρησιμοποιούμε στην παρούσα διπλωματική.
Οι εντολές που χρησιμοποιούμε είναι οι εξής :

- **int _xbegin(void)** :

  Δηλώνει την αρχή ενός transaction. Ως διεύθυνση του fallback handler ορίζεται η διεύθυνση ακριβώς κάτω από την εντολή _xbegin(). Η τιμή επιστροφής χρησιμοποιείται για να ξέρουμε αν είμαστε μέσα σε transaction ή έχει γίνει κάποιο abort. Η τιμή επιστροφής είναι το περιεχόμενο του EAX καταχωρητή (που περιέχει το status του transaction). Μπορούμε να ελέγξουμε αν το transaction έχει ξεκινήσει ή αν έχει γίνει abort κάνοντας ένα λογικό and (τελεστής & στη C) ανάμεσα στην τιμή επιστροφής και σε κάθε μία από τις παρακάτω σταθερές:
  - _XBEGIN_STARTED το transaction ξεκίνησε επιτυχώς.
  - _XABORT_{EXPLICIT,RETRY,CONFLICT,CAPACITY,DEBUG,NESTED} το transaction έγινε abort για τον αντίστοιχο λόγο.
- **void _xend(void)** :

  Δηλώνει το τέλος ενός transaction. Γίνεται commit όλων των αλλαγών που κάναμε στην μνήμη.
- **void _xabort(const unsigned int status)** :

  Explicit abort ενός transaction. Το status χρησιμοποιείται για να δείχνουμε τον λόγο για τον οποίο έγινε το abort.
- **int _xtest()** :

  Επιστρέφει 1 αν τη στιγμή της κλήσης της εκτελείται κάποιο transaction, 0 διαφορετικά.

Παράδειγμα:

```
1:  #include "rtm.h"
2:
3:  int num_retries = 10;
4:  int aborts = 0;
5:  int status;
6:  some_kind_of_lock lock; //> e.g.
7:  pthread_spinlock_t lock;
8:
9:  while (1) {
10:
11:   /* Avoid lemming effect. [37] */
12:   while ((!lock_is_free(lock));
13:
14:   int status = _xbegin();
15:   if (status == _XBEGIN_STARTED) {
16:     //> Transaction started successfully
17:
18:     //> Check if lock is free, else  abort explicitly.
19:     //> This also adds the lock to the read set so
20:     //> we abort if another thread acquires(writes to)
21:     //> the lock.
22:     if (!lock_is_free(lock))
23:       _xabort(0xff);
24:
25:   } else {
26:     //> Transaction was aborted
27:     aborts++;
28:
29:   //> Abort reason was...
30:     if (status & _XBEGIN_CONFLICT)
31:       //> data conflict
32:
33:     else if (status&_XBEGIN_CAPACITY)
34:       //> capacity (transaction read or write set
35:       //> overflow)
36:
37:  ...  //> Check for other abort reasons the
38:       //> same way.
39:
40:       //> If we exceeded the number of retries
41:       //> acquire the lock and move forward to
42:       //> execute the critical section. Else retry in
43:       //> speculative mode (e.g. go back to the
44:       //> while loop).
45:
46:     if (aborts >= num_retries)
47:       acquire_lock(lock);
48:     else
49:       continue;
50:   }
51:
52:   //> Critical section goes here...
53:
54:   if (_xtest())
55:     _xend();
56:   else
57:     release_lock(lock);
58:   break;
59:  }
```

2.2.2 Χαρακτηριστικά HTM στους Haswell επεξεργαστές

Τα βασικά χαρακτηριστικά του Hardware Transactional Memory στους Haswell επεξεργαστές είναι :

- **Ατομικότητα (Atomicity)** : Το αποτέλεσμα ενός transaction είναι είτε να γίνουν όλες οι εγγραφές στη μνήμη μονομιάς είτε να μην γίνει καμία. Όταν έχουμε transaction commit τότε όλες οι εγγραφές περνάνε στην μνήμη και όταν έχουμε transaction abort τότε δεν γίνεται καμία εγγραφή, σαν να μην συνέβη ποτέ το transaction.
- **Lazy versioning και isolation** : Κάθε φορά που ένα thread κάνει κάποια εγγραφή ενώ εκτελεί κώδικα transaction, την αποθηκεύει στον write-set του μέχρις ότου κάνει transaction commit, οπότε τότε ενημερώνει και τη μνήμη. Συνεπώς, κανένα άλλο thread δεν μπορεί να παρατηρήσει τις αλλαγές που έχουν γίνει από ένα transaction πριν αυτό κάνει commit (isolation).
- **Pessimistic (eager) conflict detection** : Γίνεται έλεγχος για conflicts σε κάθε load ή store και μόλις ανιχνευθεί κάποιο conflict τότε το transaction θα κάνει αμέσως abort.
- **Σειριοποίηση (Serializability)** : Τα αποτελέσματα των transactions είναι συνεπή, δηλαδή είναι ίδια με αυτά μιας αντίστοιχης σειριακής εκτέλεσης. Επιπλέον τα transactions φαίνονται ότι κάνουν commit σειριακά χωρίς όμως να υπάρχει κάποια εγγύηση για το ποια θα είναι η συγκεκριμένη σειρά τους.
- **False conflicts και strong isolation** : Στη πράξη το TM εντοπίζει το read και το write set ενός transaction μέσα από cache lines των 64 bytes. Πιο συγκεκριμένα, το read και το write set αποτελούνται από όλες τις cache lines στις οποίες το transaction έχει διαβάσει και έχει γράψει κατά τη διάρκεια της εκτέλεσής του. Θα ανιχνευθεί ένα conflict σε transaction αν κάποιο άλλο thread γράψει σε κάποια cache line του read-set ή διαβάσει ή γράψει σε κάποια cache line του write-set του transaction. Συνεπώς, υπάρχει πιθανότητα να δημιουργηθούν false conflicts, δηλαδή conflicts σε δεδομένα που δεν διαβάστηκαν ή γράφτηκαν από κάποιο transaction αλλά τυχαίνει να βρίσκονται στο ίδιο cache line με άλλα που έχουν σχέση με το transaction. Τέλος ισχύει η ιδιότητα strong isolation όπου το thread που προκαλεί το conflict δεν χρειάζεται κατ' ανάγκη να έχει εκτελέσει κώδικα transaction.

2.2.3 Τύποι Transactional Abort

Οι λόγοι για τους οποίους ένα transaction μπορεί να απορρίψει τη λειτουργία του είναι οι εξής :

- **Data conflict** : Όταν ένα thread το οποίο δεν εκτελεί κατ' ανάγκη κώδικα transaction γράψει σε δεδομένο που βρίσκεται σε read ή write set κάποιου άλλου transaction ή διαβάσει δεδομένο που βρίσκεται σε κάποιο write set άλλου transaction.
- **Capacity abort** : Όταν το αποτύπωμα (footprint) του transaction ξεπεράσει τα όρια χωρητικότητας των read ή write buffers.
- **Explicit abort** : Όταν ο προγραμματιστής επιτηδευμένα προκαλεί abort.
- **Other** : Πρόκληση abort από άλλους λόγους όπως interrupts, system calls κ.α.

2.3 Η δομή δεδομένων Skip List

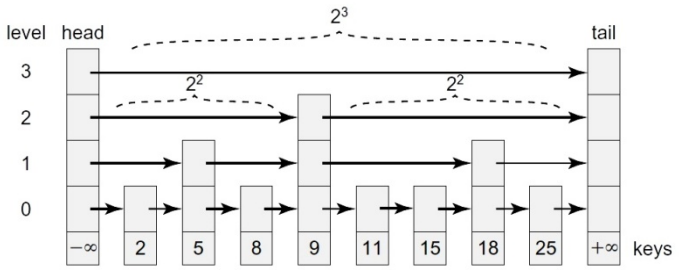
Η δομή δεδομένων Skip List [4] αποτελεί μια ενδιαφέρουσα εναλλακτική επιλογή αντί για ισοσταθμισμένα δέντρα (balanced trees), καθώς προσφέρει λογαριθμική πολυπλοκότητα, διατηρεί τη διάταξη των δεδομένων της και δεν απαιτεί ανακατανομές (rebalancing) κάτι που οδηγεί σε στενωπό απόδοσης λόγω συγκρούσεων (contention bottlenecks).

Για να καταλάβουμε διαισθητικά τη δομή δεδομένων Skip List ας παρατηρήσουμε αρχικά το σχήμα 2.5 παρακάτω χωρίς να εστιάσουμε σε λεπτομέρειες. Παρατηρούμε ότι η δομή αποτελείται από πολλές απλά συνδεδεμένες λίστες διατεταγμένες ανά επίπεδο. Η λίστα στο κατώτερο επίπεδο περιέχει όλους τους κόμβους της δομής ταξινομημένους σύμφωνα με κάποιο κλειδί. Στα ανώτερα επίπεδα κάθε λίστα περιέχει ένα υποσύνολο στοιχείων της λίστας που βρίσκεται στο αμέσως κατώτερο επίπεδο. Πως διαμορφώνεται όμως αυτή η διάταξη?

Η δομή Skip List αποτελεί μια πιθανοκρατική δομή δεδομένων. Όταν δημιουργείται ένας νέος κόμβος τότε αυτός προστίθεται στις λίστες ως εξής: Στο κατώτερο επίπεδο προστίθεται πάντα. Στη συνέχεια ρίχνουμε ένα δίκαιο νόμισμα. Αν τύχει κορώνα τότε προσθέτουμε τον κόμβο και στο επόμενο επίπεδο και επαναλαμβάνουμε τη διαδικασία έως ότου τύχουμε γράμματα οπότε και σταματάμε. Το αποτέλεσμα αυτής της ενέργειας είναι περίπου το 1/2 των στοιχείων να εμφανίζονται και στο δεύτερο επίπεδο, το 1/4 έως και στο τρίτο, το 1/8 και έως και στο τέταρτο κ.ο.κ.

Αρχικά αρχικοποιούμε την δομή με δυο κόμβους (head,tail) με τιμές κλειδιού $+\infty$ και $-\infty$ και ύψος το μέγιστο επιτρεπτό. Στη συνέχεια εισαγωγές κόμβων διαμορφώνουν την Skip List όπως για παράδειγμα φαίνεται στο σχήμα 2.5.

Στο σημείο αυτό θα επεξηγήσουμε τους σειριακούς αλγορίθμους προσπέλασης και επεξεργασίας δεδομένων της δομής Skip List. Όλοι οι αλγόριθμοι έχουν αναμενόμενη πολυπλοκότητα O(log n). Για απόδειξη παραπέμπουμε στη σχετική βιβλιογραφία [4], [8]. Κάθε κόμβος έχει ένα όρισμα next στον οποίο αποθηκεύεται ένας πίνακας δεικτών σε διάδοχους κόμβους (successors), ένας για κάθε λίστα που περιέχει τον κόμβο(σχήμα 2.7 (α)).

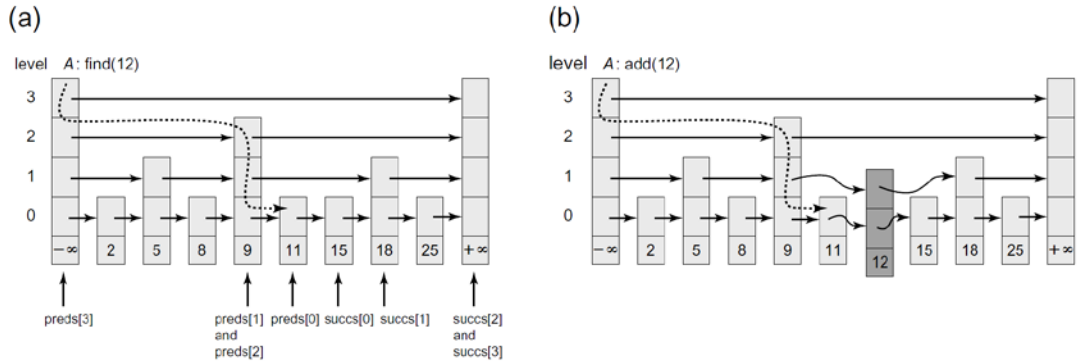


Σχήμα 2.5 *Η δομή δεδομένων Skip List : Σε αυτό το παράδειγμα έχουμε τέσσερα επίπεδα λιστών. Κάθε κόμβος έχει ένα κλειδί και οι κόμβοι head και tail έχουν κλειδιά $\pm\infty$. Η λίστα στο επίπεδο i αποτελεί συντόμευση (shortcut) όπου κάθε μετάβαση στο επίπεδο αυτό προσπερνάει 2^i κόμβους. Για παράδειγμα, στο επίπεδο 3 κάθε μετάβαση προσπερνά 2^3 κόμβους, στο επίπεδο 2 2^2 κόμβους , κ.ο.κ .*

Η συνάρτηση *find()* (σχήματα 2.6(α), 2.6(β)) προσπελαύνει τη δομή ξεκινώντας από το head και ύψος το μέγιστο επιτρεπτό και διασχίζει τα στοιχεία της λίστας αυτού του επιπέδου μέχρις ότου βρει στοιχείο με κλειδί μεγαλύτερο ή ίσο από το κλειδί που ψάχνει. Όταν το βρει αποθηκεύει τον προκάτοχο (predecessor) και τον διάδοχο (successor) κόμβο σε δύο πίνακες προκατόχων και διαδόχων αντίστοιχα. Στη συνέχεια συνεχίζει την ίδια διαδικασία στη λίστα του επόμενου επιπέδου ξεκινώντας αυτή τη φορά από τον προκάτοχο κόμβο που μόλις βρήκε πριν αλλάξει επίπεδο. Η διαδικασία ολοκληρώνεται όταν έχουν ενημερωθεί οι πίνακας προκατόχων και διαδόχων για όλα τα επίπεδα και επιστέφει τιμή ανάλογη με τον αν βρήκε ή όχι τον κόμβο με το κλειδί που έψαχνε.

Για την *εισαγωγή* ενός κόμβου στη δομή αρχικά καλούμε τη συνάρτηση find() η οποία ενημερώνει τους πίνακες προκατόχων και διαδόχων του στοιχείου που θέλουμε να εισάγουμε. Στην συνέχεια δημιουργούμε τον νέο κόμβο και τον τοποθετούμε ανάμεσα στους προκατόχους και τους διαδόχους ενημερώνοντας τους δείκτες (σχήματα 2.6(β), 2.7(γ)).

Για την *διαγραφή* ενός κόμβου καλούμε πάλι την find() η οποία βρίσκει τους προκατόχους και διαδόχους του στοιχείου που θέλουμε να διαγράψουμε. Στη συνέχεια πραγματοποιούμε την διαγραφή ενημερώνοντας τους δείκτες των προκατόχων να δείχνουν στους διαδόχους (σχήμα 2.7(δ)).



Σχήμα 2.6 *Η δομή δεδομένων Skip List : συναρτήσεις add() και find() . Στο τμήμα (α), η find() προσπελαύνει τη δομή ξεκινώντας από το head και ύψος το μέγιστο επιτρεπτό, 3 εδώ, και διασχίζει τα στοιχεία της λίστας αυτού του επιπέδου μέχρις ότου βρει στοιχείο με κλειδί μεγαλύτερο ή ίσο του 12. Όταν το βρεί, εδώ είναι το tail, αποθηκεύει τον προκάτοχο (head) και τον διάδοχο (tail) στους πίνακες preds και succs αντίστοιχα. Έπειτα συνεχίζει στο επόμενο επίπεδο κ.ο.κ ξεκινώντας από τον προκάτοχο του προηγούμενου επιπέδου. Ως αποτέλεσμα στο παράδειγμα αυτό preds[3]=head, preds[2]=preds[1]=9, preds[0]=11, succs[0]=15, succs[1]=18, succs[2]=succs[3]=tail. Η find() επιστέφει false καθώς δεν βρέθηκε κόμβος με κλειδί 12 οπότε μια κλήση add(12) στο τμήμα (b) μπορεί να προχωρήσει. Στο τμήμα (b) δημιουργείται ένας νέος κόμβος με τυχαίο toplevel=2. Το όρισμα next του κόμβου ενημερώνεται με τους αντίστοιχους succs[] κόμβους και κάθε προκάτοχος ενημερώνει το next όρισμά του ώστε να δείχνει στον νέο κόμβο στα κατάλληλα επίπεδα.*

(α)

```
1.  struct  Node{
2.    int key;
3.    Node [] next;
4.    int topLevel;
5.    state state;
6.  }
```

(β)

```
1.  int find(int x, Node [] preds, Node [] succs) {
2.    int key = x;
3.    Node pred = head;
4.    for (int level = MAX_LEVEL; level >= 0; level--)
5.    {
6.       Node curr = pred.next[level];
7.       while (key > curr.key) {
8.          pred = curr; curr = pred.next[level];
9.       }
10.      preds[level] = pred;
11.      succs[level] = curr;
12.   }
13.   if (key == curr.key) {
14.    return true;
15.   }
16.   return false;
17. }
```

(γ)

```
1.  boolean add(int x) {
2.    int topLevel = randomLevel();
3.    Node [] preds = (Node []) new Node[MAX_LEVEL + 1];
4.    Node [] succs = (Nod []) new Node[MAX_LEVEL + 1];
5.    int Found = find(x, preds, succs);
6.    if (Found == false ) {
7.       Node pred, succ;
8.       Node newNode = new Node(x, topLevel);
9.       newNode.state = INITIAL;
10.      for (int level = 0; level <= topLevel; level++)
11.         newNode.next[level] = succs[level];
12.      for (int level = 0; level <= topLevel; level++)
13.         preds[level].next[level] = newNode;
14.      newNode.state = INSERTED;
15.      return true;
16.   }
17.   return false;
18. }
```

(δ)

```
1.  boolean remove(int x) {
2.     Node [] preds = (Node []) new Node[MAX_LEVEL + 1];
3.     Node [] succs = (Node []) new Node[MAX_LEVEL + 1];
4.     int Found = find(x, preds, succs);
5.     if (Found == true ) {
6.           Node  pred, succ;
7.           Node   victim= pred[0].next[0];
8.           victim.state = DELETED;
9.            for (int level = victim.topLevel; level >= 0; level--)
10.              pred[level].next[level] = succs[level];
11.          return true;
12.    }
13.    return false;
14. }
```

Σχήμα 2.7 *Η δομή δεδομένων Skip List : ψευδοκώδικες για (α) ορίσματα δομής, (β) αναζήτηση στοιχείου, (γ) προσθήκη στοιχείου, (δ) διαγραφή στοιχείου .*

Κεφάλαιο 3

Lock-Based και Lock-Free Skip Lists

Στο παρόν κεφάλαιο παρουσιάζουμε τους αλγόριθμους που στοχεύουν στη παράλληλη προσπέλαση και επεξεργασία δεδομένων της δομής δεδομένων Skip List χρησιμοποιώντας Lock-Based και Lock-Free τεχνικές έτσι όπως αναφέρονται στην βιβλιογραφία [7].

3.1 Lock-Based Skip List

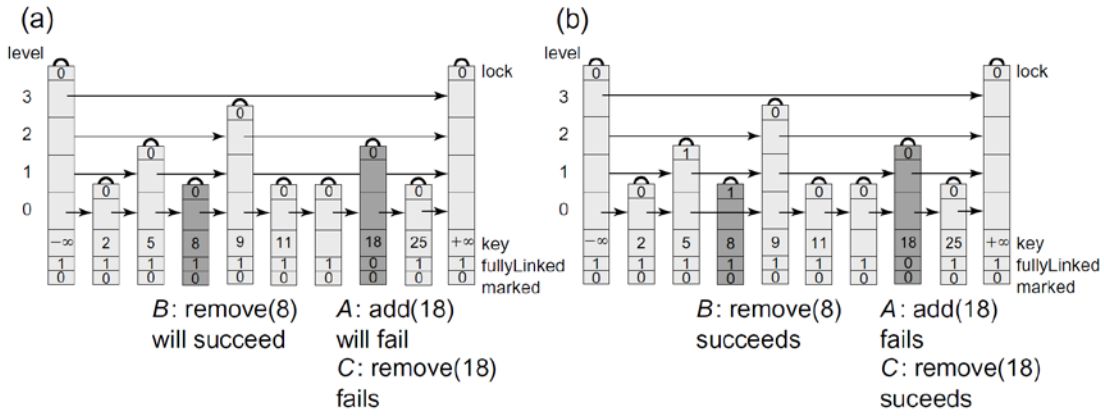
3.1.1 Περιγραφή αλγορίθμου

Ας ξεκινήσουμε παρατηρώντας εν τάχει το σχήμα 3.1 εστιάζοντας στη δομή των κόμβων. Κάθε κόμβος έχει ένα κλείδωμα και ένα πεδίο μαρκαρίσματος το οποίο μας δείχνει αν ο κόμβος ανήκει στο σύνολο δεδομένων ή έχει αφαιρεθεί λογικά (*logically removed*). Η ιδιότητα της Skip List (*skiplist property*) σύμφωνα με την οποία κάθε λίστα περιέχει ένα υποσύνολο στοιχείων της λίστας που βρίσκεται στο αμέσως κατώτερο επίπεδο διατηρείται και δεν επηρεάζεται από τον αλγόριθμο.

Αυτό συμβαίνει γιατί τα κλειδώματα αποτρέπουν αλλαγές γύρω από κόμβους που προσθέτονται ή διαγράφονται καθώς και καθυστερούν την πρόσβαση σε οποιονδήποτε κόμβο εισάγεται μέχρις ότου ολοκληρωθεί η διαδικασία εισαγωγής σε όλα τα επίπεδα. Στη συνέχεια επεξηγούμε τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης στοιχείων.

Λειτουργία Εισαγωγής

Για την εισαγωγή ενός νέου κόμβου αρχικά καλούμε την find() που είδαμε στην ενότητα 2.3, η οποία προσπελαύνει τη δομή και επιστρέφει τους προκατόχους και τους διαδόχους του νέου κόμβου για όλα τα επίπεδα. Στη συνέχεια, για να αποτρέψουμε αλλαγές στους προκατόχους τους κλειδώνουμε ξεκινώντας από το επίπεδο 0 μέχρι το ανώτερο επίπεδο του νέου κόμβου ενώ ταυτόχρονα ελέγχουμε 1) αν ο προκάτοχος είναι μαρκαρισμένος (logically deleted), 2) αν ο διάδοχος είναι μαρκαρισμένος και 3) αν ο προκάτοχος δείχνει στον διάδοχο. Σε περίπτωση που ο έλεγχος αποτύχει τότε ελευθερώνουμε τα κλειδώματα και επαναλαμβάνουμε την διαδικασία ξανακαλώντας την find(). Αν ο έλεγχος επιτύχει συνεχίζουμε με την προσθήκη του κόμβου ενημερώνοντας τους δείκτες όπως ακριβώς έγινε με την σειριακή add που είδαμε στην ενότητα 2.3. Προκειμένου να διασφαλίσουμε το skiplist property θεωρούμε ότι ο κόμβος δεν ανήκει λογικά στο σετ μέχρις ότου ενημερωθούν όλες οι αναφορές σε αυτόν σε όλα τα επίπεδα. Για το σκοπό αυτό κρατάμε ένα πεδίο *fullyLinked* το οποίο λαμβάνει την τιμή αληθές όταν ο



Σχήμα 3.1 *Lock-Based Skip List : αποτυχημένες και επιτυχημένες κλήσεις add() και remove(). Στο τμήμα (a) η κλήση add(18) εντοπίζει τον κόμβο με κλειδί 18 αμαρκάριστο αλλά όχι πλήρως συνδεδεμένο. Σπινιάρει μέχρι να συνδεθεί πλήρως στο τμήμα (b) όπου και επιστρέφει την τιμή false. Στο τμήμα (a) η κλήση remove(8) εντοπίζει τον κόμβο με κλειδί 8 αμαρκάριστο και πλήρως συνδεδεμένο, το οποίο σημαίνει ότι μπορεί να καταλάβει το κλείδωμα του κόμβου στο τμήμα (b). Έπειτα θέτει το bit μαρκαρίσματος και συνεχίζει κλειδώνοντας τους προκατόχους που στη συγκεκριμένη περίπτωση είναι ο είναι ο κόμβος με κλειδί 5. Μόλις κλειδωθεί ο προκάτοχος, πραγματοποιεί την φυσική διαγραφή του κόμβου από τη λίστα αλλάζοντας τον δείκτη στο κατώτερο επίπεδο του κόμβου με κλειδί 5 ολοκληρώνοντας έτσι επιτυχώς την κλήση remove(). Στο τμήμα (a) η remove(18) αποτυγχάνει, καθώς ο ζητούμενος κόμβος δεν έχει συνδεθεί πλήρως. Η ίδια κλήση επιτυγχάνει στο τμήμα (b) γιατί εδώ ο ζητούμενος κόμβος έχει εισαχθεί πλήρως.*

κόμβος συνδεθεί σε όλα τα επίπεδα. Δεν επιτρέπουμε πρόσβαση στον κόμβο μέχρις ότου εισαχθεί πλήρως στη δομή. Για παράδειγμα η add() όταν προσπαθεί να καταλάβει αν ο κόμβος που θέλει να εισάγει βρίσκεται ήδη στη δομή πρέπει να περιμένει (σπινιάρει) μέχρις ότου συνδεθεί πλήρως. Στο σχήμα 3.1 βλέπουμε μια κλήση add(18) που σπινιάρει περιμένοντας τον κόμβο με κλειδί 18 να συνδεθεί πλήρως. Όταν ο κόμβος συνδεθεί πλήρως τότε ολοκληρώνεται η διαδικασία εισαγωγής και ελευθερώνονται όλα τα κλειδώματα.

Λειτουργία Διαγραφής

Για την διαγραφή ενός κόμβου καλούμε την find() ώστε να εντοπίσουμε τον κόμβο με το ζητούμενο κλειδί. Αν βρεθεί τότε εξετάζουμε αν ο κόμβος αυτός έχει συνδεθεί πλήρως και δεν είναι μαρκαρισμένος. Για παράδειγμα, στο τμήμα (a) του σχήματος 3.1 η remove(8) βρίσκει τον κόμβο με κλειδί 8 αμαρκάριστο και πλήρως συνδεδεμένο που σημαίνει ότι μπορεί να τον διαγράψει. Αντίθετα η κλήση remove(18) αποτυγχάνει γιατί διαπιστώνει ότι ο ζητούμενος κόμβος δεν έχει εισαχθεί πλήρως. Η ίδια κλήση επιτυγχάνει στο τμήμα (b) γιατί εδώ ο ζητούμενος κόμβος έχει συνδεθεί πλήρως. Αν ο έλεγχος επιτύχει τότε κλειδώνουμε τον κόμβο και τον μαρκάρουμε.

Στη συνέχεια ξεκινώντας από το επίπεδο 0 μέχρι το ανώτερο επίπεδο του νέου κόμβου κλειδώνουμε τους προκατόχους και ελέγχουμε αν δείχνουν στον ζητούμενο κόμβο καθώς και αν είναι αμαρκάριστοι. Αν σε οποιοδήποτε σημείο ο έλεγχος αποτύχει ελευθερώνουμε τα κλειδώματα και επαναλαμβάνουμε την διαδικασία από την αρχή

ξανακαλώντας την find(). Αν όλοι οι έλεγχοι επιτύχουν τότε ξεκινώντας από το ανώτερο επίπεδο αυτή τη φορά ώστε να διασφαλίσουμε το skiplist property ενημερώνουμε τους δείκτες των προκατόχων να δείχνουν στους διαδόχους πραγματοποιώντας έτσι τη φυσική διαγραφή του ζητούμενου κόμβου.

Για παράδειγμα, στο τμήμα (b) η remove(8) κλειδώνει τον προκάτοχο με κλειδί 5. Μόλις κλειδωθεί ο προκάτοχος, η remove πραγματοποιεί την φυσική διαγραφή του κόμβου από τη λίστα αλλάζοντας τον δείκτη στο κατώτερο επίπεδο του κόμβου με κλειδί 5 να δείχνει στο κόμβο με κλειδί 9.

Λειτουργία Αναζήτησης

Για την αναζήτηση ενός κόμβου καλούμε την find() για να εντοπίσουμε τον κόμβο με το ζητούμενο κλειδί. Αν βρεθεί ο κόμβος, τότε θεωρούμε ότι ο κόμβος ανήκει στην δομή αν είναι αμαρκάριστος και πλήρως συνδεδεμένος. Η λειτουργία αυτή είναι wait-free καθώς αγνοεί τα κλειδώματα και αλλαγές που πιθανόν να γίνονται ταυτόχρονα.

3.2 Lock-Free Skip List

3.2.1 Περιγραφή αλγορίθμου

Στη περίπτωση του Lock-Free αλγορίθμου επειδή δεν χρησιμοποιούνται κλειδώματα δεν μπορεί να διασφαλιστεί το skiplist property, ότι δηλαδή κάθε λίστα περιέχει ένα υποσύνολο στοιχείων της λίστας που βρίσκεται στο αμέσως κατώτερο επίπεδο. Εφόσον λοιπόν δεν μπορούμε να διασφαλίσουμε την ιδιότητα αυτή, θα θεωρήσουμε ότι το abstract set θα ορίζεται μόνο από τη λίστα στο κατώτερο επίπεδο: ένα κλειδί ανήκει στο σετ αν και μόνο αν υπάρχει κόμβος με αυτό το κλειδί στη λίστα του κατώτερου επιπέδου. Οι κόμβοι στα ανώτερα επίπεδα λειτουργούν μόνο βοηθητικά για να φτάσουμε γρηγορότερα σε κόμβους του χαμηλότερου επιπέδου. Ως αποτέλεσμα ο lock-free αλγόριθμος δεν χρησιμοποιεί το πεδίο fullyLinked. Επίσης δεν έχουμε ένα marked bit για κάθε κόμβο αλλά ένα για κάθε επίπεδο του κόμβου. Πριν δούμε τις λειτουργίες εισαγωγής, διαγραφής και αναζήτησης ας δούμε πρώτα την συνάρτηση find() καθώς είναι λίγο διαφορετική από αυτήν που έχουμε δει μέχρι τώρα.

find()

Ο σκοπός της find() είναι να διαπιστώσει αν υπάρχει στοιχείο με το κλειδί που αναζητεί καθώς και να επιστρέψει ένα πίνακα προκατόχων και ένα πίνακα διαδόχων όπως έκανε μέχρι τώρα. Η διαφορά είναι ότι τώρα αναλαμβάνει και ένα επιπλέον ρόλο αυτό της φυσικής απομάκρυνσης κόμβων που έχουν διαγραφεί λογικά (έχουν μαρκαριστεί). Ξεκινώντας από το πρώτο στοιχείο και μέγιστο ύψος διασχίζει οριζόντια τους κόμβους κοιτάζοντας αν είναι μαρκαρισμένοι ή όχι. Αν βρεθεί μαρκαρισμένος κόμβος τότε προχωράει χωρίς να κοιτάει τιμές κλειδιών μέχρις ότου συναντήσει τον πρώτο αμαρκάριστο κόμβο και τον ενώνει με τον προηγούμενο αμαρκάριστο κόμβο χρησιμοποιώντας την ατομική εντολή *compareAndSet*. Αν αποτύχει η εντολή είτε διότι ο κόμβος του οποίου

θέλουμε να αλλάξουμε τον δείκτη είναι μαρκαρισμένος, είτε γιατί πλέον δεν δείχνει εκεί που έδειχνε όταν εξετάστηκε, τότε επαναλαμβάνεται όλη η διαδικασία από την αρχή. Αν επιτύχει τότε συνεχίζει τη προσπέλαση με παρόμοιο τρόπο μέχρι να βρει κόμβο με τιμή κλειδιού μεγαλύτερη ή ίση του ζητούμενου κλειδιού. Ύστερα κατεβαίνει επίπεδο και επαναλαμβάνει την διαδικασία έως και το κατώτερο επίπεδο.

Λειτουργία Εισαγωγής

Η λειτουργία εισαγωγής αρχικά καλεί την find() η οποία επιστρέφει τους πίνακες προκατόχων και διαδόχων. Αν δεν υπάρχει ήδη το κλειδί στο σετ τότε προχωράμε στην δημιουργία ενός νέου κόμβου και συνδέουμε τις αναφορές του με τους διαδόχους. Στη συνέχεια ενώνουμε τον προκάτοχο στο κατώτερο επίπεδο με τον νέο κόμβο μέσω της ατομικής εντολής compareAndSet, γεγονός που σηματοδοτεί την εισαγωγή του νέου κόμβου στο σετ. Αν η ατομική εντολή αποτύχει, είτε γιατί ο προκάτοχος είναι μαρκαρισμένος είτε γιατί δεν δείχνει στον διάδοχο, τότε επαναλαμβάνουμε την διαδικασία από την αρχή. Αν επιτύχει, τότε συνεχίζουμε στο επόμενο επίπεδο ενώνοντας τον προκάτοχο με τον νέο κόμβο μέσω εντολής compareAndSet και προχωράμε προς τα πάνω κάνοντας το ίδιο. Αν σε οποιονδήποτε επίπεδο η ατομική εντολή αποτύχει για τους ίδιους λόγους που εξηγήσαμε τότε καλούμε την find() ώστε να ανανεώσουμε τους πίνακες προκατόχων και διαδόχων και συνεχίζουμε από το επίπεδο που σταματήσαμε.

Λειτουργία Διαγραφής

Όπως και με την εισαγωγή έτσι και με την διαγραφή στοιχείου αρχικά καλούμε την find(). Εφόσον βρεθεί το στοιχείο με το ζητούμενο κλειδί τότε ξεκινώντας από το ανώτερο επίπεδο του στοιχείου και για κάθε επίπεδο μέχρι να φτάσουμε στο προτελευταίο από κάτω, διαγράφουμε λογικά τις αναφορές του μαρκάροντας τες μέσω ατομικών εντολών compareAndSet. Αν κάποια ατομική εντολή αποτύχει, αυτό έχει συμβεί για κάποιον/ους από τους ακόλουθους λόγους: Κάποιο άλλο νήμα ταυτόχρονα διαγράφει το στοιχείο (έχει θέσει το marked bit στο επίπεδο αυτό) όπου σε αυτή τη περίπτωση συνεχίζουμε στο επόμενο επίπεδο ή ο δείκτης σε επόμενο κόμβο έχει αλλάξει οπότε τον ξαναδιαβάζουμε και επαναλαμβάνουμε την ατομική εντολή [2]. Στη συνέχεια διαγράφει λογικά τον κόμβο από την κατώτερη λίστα γεγονός που σηματοδοτεί την λογική διαγραφή του κόμβου από το σετ. Προκειμένου να διαγραφεί φυσικά ο κόμβος καλεί την find να αναζητήσει το στοιχείο που μόλις διέγραψε λογικά, η οποία όπως εξηγήσαμε διαγράφει φυσικά όλες τις μαρκαρισμένες αναφορές που συναντάει. Συνεπώς ο κόμβος διαγράφεται φυσικά τόσο από την find() που καλούμε εμείς όσο και από κλήσεις find() που καλούνται από άλλα νήματα.

---

[2] *Η ατομική εντολή που χρησιμοποιείται είναι η εξής :*

NodeToRemove.next[Level].compareAndSet(succ, succ, false, true);

Το πρώτο και το δεύτερο όρισμα της παρενθέσεως αναφέρονται στον διάδοχο κόμβο, ενώ το τρίτο και το τέταρτο όρισμα αναφέρονται στο bit μαρκαρίσματος. Στη πράξη το bit μαρκαρίσματος αποτελεί το πρώτο bit του δείκτη προς το διάδοχο. Άρα ο δείκτης περιέχει τη διεύθυνση succ του διαδόχου και το bit μαρκαρίσματος το οποίο ενσωματώθηκε μέσω της λογικής πράξης OR με τη διεύθυνση (το πρώτο bit της διεύθυνσης είναι πάντα 0). Συνεπώς το πρώτο και το τρίτο όρισμα αποτελούν την παλιά τιμή του δείκτη η οποία εφόσον είναι έγκυρη μετατρέπεται στη νέα που ορίζεται από το δεύτερο και τέταρτο όρισμα.

Λειτουργία Αναζήτησης

Στον lock-free αλγόριθμο η λειτουργία αναζήτησης δεν μπορεί να λειτουργήσει με τον κλασσικό τρόπο όπου προσπελαύνει τη δομή εξετάζοντας τα κλειδιά όλων των προσβάσιμων κόμβων ανεξαρτήτως εάν είναι μαρκαρισμένοι ή όχι. Επειδή οι εισαγωγές και οι διαγραφές παραβιάζουν το skiplist property μπορεί ένας μαρκαρισμένος κόμβος να είναι προσπελάσιμος σε κάποιο ανώτερο επίπεδο αλλά να έχει διαγραφεί φυσικά στο κατώτερο. Εάν αγνοηθεί η τιμή μαρκαρίσματος τότε η λειτουργία αναζήτησης μπορεί να αγνοήσει κόμβους που είναι προσβάσιμοι στο κατώτερο επίπεδο.

Θα μπορούσε να χρησιμοποιηθεί η find() για αναζήτηση η οποία αντιμετωπίζει αυτό το πρόβλημα. Στις περισσότερες εφαρμογές οι αναζητήσεις είναι πιο συχνές από τις ενημερώσεις. Συνεπώς η επιλογή της find() θα δημιουργούσε πρόβλημα καθώς πολλαπλές ταυτόχρονες find() που προσπαθούν να "καθαρίσουν" τους ίδιους κόμβους οδηγούν σε συνωστισμό. Η λύση που προτείνεται είναι μια λειτουργία αναζήτησης που εκτελεί ακριβώς τα ίδια βήματα με τη λειτουργία find() χωρίς όμως να προχωράει σε φυσική διαγραφή κόμβων.

Τόσο ο lock-free αλγόριθμος όσο και ο lock-based είναι *linearizable*. Για τις σχετικές αποδείξεις παραπέμπουμε τον αναγνώστη στο πρωτότυπο κείμενο [7].

Κεφάλαιο 4

Υλοποίηση Skip List χρησιμοποιώντας Hardware Transactional Memory

Στο κεφάλαιο αυτό παρουσιάζουμε την HTM εκδοχή που έχουμε υλοποιήσει. Περιγράφουμε λεπτομερώς τους αλγορίθμους που χρησιμοποιήσαμε, αιτιολογούμε την εγκυρότητά τους και παρέχουμε μερικές ιδέες για βελτιστοποιήσεις.

4.1 Εισαγωγή

Στο προηγούμενο κεφάλαια είδαμε τους lock based και lock free αλγορίθμους . Και οι δύο τρόποι υλοποίησης συνιστούν αξιόλογες επιλογές για τους προγραμματιστές που επιθυμούν να υλοποιήσουν δομές Skip List. Η ενασχόληση μας στη διπλωματική με HTM έχει στόχο να διευρύνει τις επιλογές προσφέροντας εναλλακτικές σε περιπτώσεις όπου οι άλλες μέθοδοι αντιμετωπίζουν προβλήματα. Για παράδειγμα το lock convoying που εξηγήσαμε στο πρώτο κεφάλαιο αποτελεί πρόβλημα για τις lock based υλοποιήσεις ενώ η προγραμματιστική πολυπλοκότητα και το κόστος των ατομικών εντολών [9] ιδιαίτερα σε πολύ μεγάλα σετ δεδομένων όπου δεν απαιτείται αρκετός συγχρονισμός αποτελούν προβλήματα στις lock free υλοποιήσεις. Αναμένουμε οι HTM υλοποιήσεις να αντιμετωπίζουν τέτοιου είδους προβλήματα. Αυτό θα διαπιστωθεί στο επόμενο κεφάλαιο.

Σε αυτό το κεφάλαιο παρουσιάζουμε δυο διαφορετικές εκδοχές υλοποίησης Skip List χρησιμοποιώντας Hardware Transactional Memory. Αρχικά περιγράφουμε την coarse-grained HTM εκδοχή όπου περικλείουμε τους σειριακούς αλγορίθμους επεξεργασίας δεδομένων με transactions και στην συνέχεια την fine-grained HTM εκδοχή όπου προσπαθούμε να ελαχιστοποιήσουμε το μέγεθος των transactions όπως κάναμε αντίστοιχα στην fine-grained lock based εκδοχή. Με την coarse-grained HTM εκδοχή επιδιώκουμε να ελαχιστοποιήσουμε την προγραμματιστική δυσκολία που απαιτείται ενώ ταυτόχρονα να πετύχουμε καλές αποδόσεις. Με την fine-grained HTM εκδοχή εφαρμόζοντας τεχνικές που ελαχιστοποιούν το μέγεθος των transactions σκοπεύουμε να μεγιστοποιήσουμε την απόδοση.

4.2 Coarse-Grained αλγόριθμος

4.2.1 Περιγραφή αλγορίθμου

Ο coarse-grained HTM skip list αλγόριθμος είναι ιδιαίτερα απλός. Για να εισάγουμε ή να διαγράψουμε ένα στοιχείο από τη δομή καλούμε τους σειριακούς αλγορίθμους επεξεργασίας δεδομένων μέσα σε transaction όπως φαίνεται στο σχήμα 4.1 (α). Λόγω του μεγάλου footprint του transaction υπάρχει μεγάλη πιθανότητα να δεχτούμε επανειλημμένα aborts κυρίως λόγω data conflict ή capacity overflows. Στο σχήμα 4.1(γ),(δ) φαίνεται ο αλγόριθμος που διαχειρίζεται τα transactions και ακολουθεί το παράδειγμα που είδαμε στην ενότητα 2.2.1. Αν ένα transaction δεχτεί abort τότε εκτελείται το fallback path (σχ. 4.1 (γ) γραμμές 17-22) όπου είτε ξαναεκτελούμε την λειτουργία σε transactional mode, είτε αν έχουμε υπερβεί το ανώτερο όριο επαναλήψεων καταλαμβάνουμε το καθολικά προσβάσιμο κλειδί και εκτελούμε τη λειτουργία σε non-transactional mode. Σε αυτή τη περίπτωση έχουμε έναν updater και πολλούς readers.

(α)

```
1.  boolean update(int x,operation op) {
2.    boolean result;
3.    tx_start(max_tx_retries, global_lock);
4.    if (op == INSERT)
5.         result = add(x);
6.    elseif (op == REMOVE)
7.         result = remove(x);
8.     tx_end(global_lock);
9.     return result;
10. }
```

(β)

```
1.  boolean contains(int  key){
2.    boolean result=false;
3.    Node curr, pred;
4.    pred= head;
5.    for ( int h = levelmax-1; h >= 0; h--)  {
6.         curr = pred.next[h];
7.         while (key > curr.key)  {
8.              pred=curr;
9.              curr = pred.next[h];
10.        }
11.   }
12.   if (key == curr.key) {
13.        while(curr.state == INITIAL);
14.        if (curr.state != DELETED) result=true;
15.   }
16.   return result;
17. }
```

(γ)

```
1.  int tx_start(int num_retries, lock fallback_lock)
2.  {
3.     int status = 0;
4.     int aborts = num_retries;
5.
6.     while (1) {
7.          /* Avoid lemming effect. */
8.          while (!lock_is_free(fallback_lock));
9.
10.         status = _xbegin();
11.         if (status == _XBEGIN_STARTED) {
12.              if (!lock_is_free(fallback_lock))
13.                   _xabort(0xff);
14.              return num_retries - aborts;
15.         }
16.
17.         /* Abort comes here. */
18.
19.         if (--aborts <= 0) {
20.              acquire_lock(fallback_lock);
21.              return num_retries - aborts;
22.         }
23.    }
24.
25.         /* Unreachable. */
26.         return -1;
27. }
```

(δ)

```
1.  void tx_end(lock fallback_lock)
2.  {
3.       if (lock_is_free(fallback_lock))
4.            _xend();
5.       else
6.            release_lock (fallback_lock);
7.       return;
8.  }
```

Σχήμα 4.1 *Coarse-Grained HTM Skip List : ψευδοκώδικες για (α) ενημέρωση δομής, (β) αναζήτηση στοιχείου, (γ),(δ) διαχείριση transactions .*

Για την αναζήτηση ενός στοιχείου καλούμε τη συνάρτηση contains όπως φαίνεται στο σχήμα 4.1(β). Σε αντίθεση με τη συνάρτηση update η contains δεν απαιτεί transactions για την διασφάλιση συγχρονισμού. Για να κατανοήσουμε το λόγο για τον οποίο ισχύει αυτό, θα πρέπει πρώτα να θέσουμε τα linearization points για τις διάφορες λειτουργίες.

4.2.2 Linearization Points

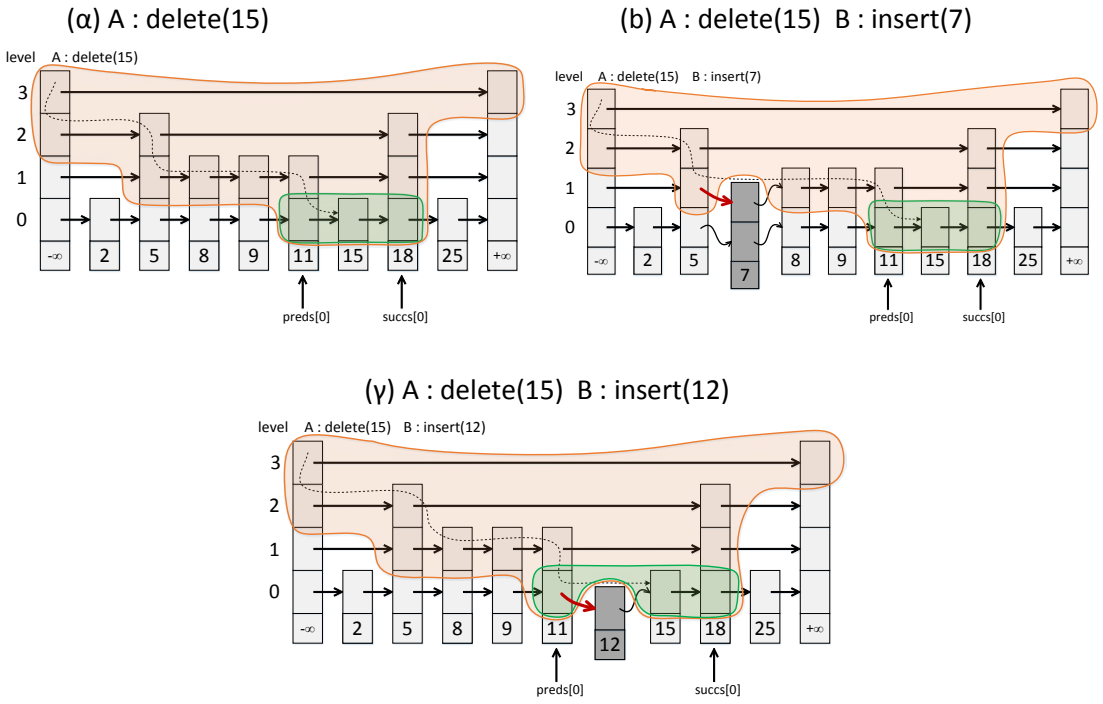
Το linearization point για update σε transactional mode είναι το σημείο όπου το transaction κάνει commit. Όταν είμαστε σε non-transactional mode, δηλαδή έχουμε καταλάβει το κλείδωμα και είμαστε οι μοναδικοί updaters, τότε το linearization point για επιτυχημένο update είναι το σημείο όπου θέτουμε τη κατάσταση του ζητούμενου κόμβου σε INSERTED (γραμμή 14 σχ. 2.7 (γ)) για τη περίπτωση εισαγωγής στοιχείου και σε DELETED (γραμμή 8 σχ. 2.7 (δ)) για τη περίπτωση διαγραφής στοιχείου. Το linearization point ενός αποτυχημένου update είναι το σημείο όπου διαπιστώνουμε ότι το κλειδί δεν υπάρχει (για add) ή αντίστοιχα υπάρχει (για remove) στη δομή (γραμμή 6 σχ. 2.7 (γ) και γραμμή 5 σχ. 2.7 (δ)). Το linearization point μιας κλήσης contains() που βρίσκει τον κόμβο με το ζητούμενο κλειδί είναι το σημείο όπου διαπιστώνει ότι ο κόμβος ανήκει στη δομή (γραμμή 14 σχ. 4.1 (β)). Αν η αναζήτηση δεν βρει στοιχείο με το ζητούμενο κλειδί ή το βρει και είναι σε κατάσταση DELETED υπάρχει περίπτωση στο μεταξύ να έχει εισαχθεί νέος κόμβος με ίδιο κλειδί. Όπως όμως και στη lock based περίπτωση έτσι και εδώ έχει υπάρξει κάποια στιγμή κατά την διάρκεια της εκτέλεσης της συνάρτησης contains όπου η δομή δεν περιελάμβανε στοιχείο με το ζητούμενο κλειδί. Επομένως, με τα linearization points που θέσαμε έχουμε διασφαλίσει έγκυρο συγχρονισμό με non-transactional αναζητήσεις.

4.2.3 Προβλήματα Coarse-Grained αλγορίθμου

Ο coarse-grained αλγόριθμος είναι ιδιαίτερα απλός και αναμένουμε ικανοποιητική επίδοση σε περιπτώσεις με μικρή ανάγκη για συγχρονισμό. Τι γίνεται όμως όταν έχουμε μεγαλύτερη συμφόρηση ή όταν το μέγεθος της πληροφορίας είναι αρκετά μεγάλο? Έχουμε διαπιστώσει τα εξής δύο προβλήματα :

Πρόβλημα 1 - Ανεπιθύμητα Aborts. Όπως έχουμε προαναφέρει ο σκοπός των read και write sets είναι να εντοπίζουν περιπτώσεις όπου νήματα γράφουν σε δεδομένα που έχουμε διαβάσει εντός transaction ή διαβάζουν/γράφουν σε δεδομένα που έχουμε γράψει (εντός transaction) έτσι ώστε να εξασφαλιστεί η εγκυρότητα της λειτουργίας που εκτελούμε. Για παράδειγμα, όταν εκτελούμε μια διαγραφή στοιχείου το footprint του transaction περιέχει τους πίνακες προκατόχων και διαδόχων του στοιχείου που θέλουμε να διαγράψουμε. Αν από τη στιγμή που διαβάσουμε έναν προκάτοχο μέχρι τη στιγμή που ξεκινήσουμε να διαγράψουμε το στοιχείο κάποιο άλλο νήμα εισάγει ένα κόμβο ανάμεσα στον προκάτοχο και το στοιχείο τότε το HTM σύστημα θα εντοπίσει την αλλαγή δείκτη του "πρώην" προκατόχου και θα απορρίψει το transaction προφυλάσσοντας μας από μια λανθασμένη λειτουργία διαγραφής (θα ενώναμε τον "πρώην" προκάτοχο με τον διάδοχο). Δεν είναι όμως όλες οι περιπτώσεις που δεχόμαστε abort λόγω conflict απαραίτητες για να

εξασφαλισθεί η εγκυρότητα της λειτουργίας. Η τροποποίηση από άλλο νήμα ενός δεδομένου που βρίσκεται στο read set μας και δεν έχει άμεση σχέση με την λειτουργία ενημέρωσης που εκτελούμε θα οδηγήσει σε περιττό abort. Για να το καταλάβουμε καλύτερα ας κοιτάξουμε το σχήμα 4.2 (α). Το νήμα Α εκτελεί την coarse-grained λειτουργία διαγραφής delete(15). Με διακεκομμένη φαίνεται η πορεία προσπέλασης της δομής που ακολουθήθηκε και με πορτοκαλί πλαίσιο το footprint του transaction. Στο σημείο αυτό έχουμε ολοκληρώσει την προσπέλαση αλλά δεν έχουμε ξεκινήσει την ενημέρωση της δομής. Στη συνέχεια στο σχήμα (β) ένα άλλο νήμα Β εισάγει το στοιχείο με κλειδί 7 στη δομή. Η αλλαγή του δείκτη του στοιχείου με κλειδί 5 που φαίνεται με κόκκινο χρώμα θα οδηγήσει σε abort το transaction που εκτελείται από το νήμα Α καθώς ο δείκτης αυτός είναι στο read set του Α. Το abort αυτό είναι ανεπιθύμητο καθώς ο δείκτης αυτός σε καμία περίπτωση δεν επηρεάζει την διαγραφή του στοιχείου με κλειδί 15. Στη πραγματικότητα μόνο αλλαγές εντός του πλαισίου με πράσινο χρώμα επηρεάζουν την εγκυρότητα και ορθά τότε δεχόμαστε abort. Οποιαδήποτε αλλαγή εκτός του πράσινου πλαισίου θα επιφέρει ανεπιθύμητο abort. Συνεπώς παρατηρούμε ότι το footprint του transaction στην coarse-grained εκδοχή είναι αρκετά μεγαλύτερο από τις ανάγκες μας για συγχρονισμό κάτι που σε περιπτώσεις συμφόρησης οδηγεί σε πάρα πολλά ανεπιθύμητα aborts και άρα περιορισμένη απόδοση.



Σχήμα 4.2 Πρόβλημα 1 της *Coarse-Grained HTM Skip List : Στο τμήμα (α) προσπελαύνουμε τη δομή για να διαγράψουμε το στοιχείου με κλειδί 15. Με διακεκομμένη φαίνεται η πορεία προσπέλασης της δομής. Με πορτοκαλί πλαίσιο είναι το footprint του coarse-grained HTM transaction ενώ το πράσινο είναι το ιδανικό footprint για τη συγκεκριμένη λειτουργία. Στο τμήμα (β) εισάγουμε το κλειδί 7 με αποτέλεσμα ο Α να δεχθεί ανεπιθύμητο abort. Στο τμήμα (γ) εισάγουμε το κλειδί 12 και δημιουργείται σωστό abort καθώς χάρη σε αυτό διασφαλίζεται η συνέπεια της δομής.*

Πρόβλημα 2 - Υπερχείλιση. Όταν το σετ δεδομένων αποτελείται από πάρα πολλά δεδομένα (π.χ. 100 εκατ. στοιχεία) ή το μέγεθος της πληροφορίας είναι αρκετά μεγάλο σε bytes τότε το footprint του transaction μπορεί να υπερβεί τα ανώτερα όρια των read και write sets με αποτέλεσμα να αυξηθούν αρκετά τα capacity aborts.

4.3 Fine-Grained αλγόριθμος

Ένας τρόπος για να αντιμετωπίζαμε τα προβλήματα που προαναφέραμε θα ήταν να σπάγαμε τα transactions σε μικρότερα, για παράδειγμα όταν προσπελαύνουμε τη δομή να έχουμε ένα transaction για κάθε επίπεδο. Παρόλο που θα αυξάναμε την απόδοση δεν θα πλησιάζαμε εκείνης των lock based και lock free τεχνικών καθώς και πάλι αναμένουμε ένα μεγάλο αριθμό από aborts, κάτι που οδηγεί σε αλλεπάλληλες επαναπροσπέλασεις της δομής, γεγονός που αποφεύγεται στις άλλες υλοποιήσεις. Γενικά έχει διαπιστωθεί [10] ότι θα πρέπει να ισχύουν οι παρακάτω τέσσερεις συνθήκες για να επιτύχουμε υψηλή κλιμακωσιμότητα σε παράλληλες δομές δεδομένων :

- Η λειτουργία αναζήτησης δεν πρέπει να περιλαμβάνει αναμονές (waiting), επαναδοκιμές (retries) ή αποθηκεύσεις δεδομένων (stores).
- Η διαδικασία προσπέλασης δεδομένων κατά τη διάρκεια μιας λειτουργίας ενημέρωσης δεν πρέπει να προβεί σε αποθηκεύσεις δεδομένων εκτός εάν πρόκειται για cleaning σκοπούς καθώς και δεν θα πρέπει να περιλαμβάνει αναμονές και επαναπροσπάθειες.
- Σε μια λειτουργία ενημέρωσης στην οποία η διαδικασία προσπέλασης αποτυγχάνει, π.χ. δεν βρέθηκε στοιχείο με το κλειδί που ψάχνουμε στη περίπτωση διαγραφής ή το κλειδί υπάρχει είδη σε στοιχείο στη περίπτωση εισαγωγής, τότε δεν πρέπει να έχουμε εκτελέσει αποθηκεύσεις (stores) εκτός από αυτές που χρησιμοποιήθηκαν στη διαδικασία προσπέλασης για cleaning.
- Ο αριθμός των αποθηκεύσεων και η περιοχή της μνήμης που τροποποιείται θα πρέπει να είναι παραπλήσιες με εκείνες της σειριακής εκδοχής.

Στόχος μας λοιπόν είναι η δημιουργία ενός αλγορίθμου HTM που να ικανοποιεί τις παραπάνω συνθήκες. Για να το πετύχουμε αυτό χρησιμοποιήσαμε μια τεχνική προγραμματισμού, τη λεγόμενη *Consistency-Oblivious Programming (COP)* [6], η οποία έχει δοκιμαστεί μέχρι στιγμής σε HTM υλοποιήσεις παράλληλων δενδρικών δομών δεδομένων. Γενικά, με την COP τεχνική διακρίνουμε δύο φάσεις : μια επιλήσμονα φάση που τρέχουμε χωρίς transactions ή locks και μια ατομική φάση που ελέγχουμε αν η επιλήσμονα φάση εκτελέστηκε σωστά. Σε περίπτωση που διαπιστωθεί κάποιο λάθος ξαναδοκιμάζουμε από την αρχή. Η τεχνική αυτή απαιτεί ιδιαίτερη προσοχή για τη διασφάλιση εγκυρότητας καθώς και για την διαχείριση των transactions στο fallback path στη περίπτωση που υλοποιούμε την ατομική φάση με transactions. Στην επόμενη ενότητα εξηγούμε ακριβώς πως εφαρμόζουμε την τεχνική αυτή σε skip lists.

4.3.1 Λεπτομερής περιγραφή αλγορίθμου

(α)

```
1.  int insert_TM(int key, info info)
2.  {
3.    int h , ret , status;
4.    Node preds[levelmax];
5.    Node succs[levelmax];
6.    Node curr , pred;
7.    int retry = -1;
8.
9.  start_from_scratch_i:
10.   retry++;
11.   if (retry >= max_invalidation_abort_retries) {
12.     acquire_lock(global_lock);
13.     ret=insert_sec(key,info);
14.     release_lock(global_lock);
15.     return ret;
16.   }

17. pred = head;
18.   for ( h = levelmax-1; h >= 0; h--) {
19.     curr = pred.next[h];
20.     while (key > curr.key) {
21.       pred = curr;
22.       curr = pred.next[h];
23.     }
24.     preds[h] = pred;
25.     succs[h] = curr;
26.   }
27.   if (key == curr.key) return 0; // if node exists, we return
                                     // Linearization point in
                                     // fallback lock mode for
                                     // unsuccessful insertion
28.
29.   int nodeHeight = get_rand_level(); // find the height of
                                         // the new node
30.   Node newNode = createNewNode(key,info,nodeHeight);
31.
32.   status = tx_start(max_tx_retries, global_lock);
33.   if (status == INVALIDATION_ABORT){
34.     goto start_from_scratch_i;
35.   }
36.
37.   //check consistency
38.   for(h = 0; h < nodeHeight; h++ )  {
39.     if (preds[h].next[h] != succs[h]
            || preds[h].state ==  DELETED
            || succs[h].state == DELETED){
40.
41.       deleteNode(newNode);
42.       if (_xtest())
43.         _xabort(0xaa); //force an abort
44.       else {
45.         release_lock (global_lock));
46.         goto start_from_scratch_i;
```

(β)

```
1.  int delete_TM(int key, info info)
2.  {
3.    int h, nodeHeight, status, ret;
4.    Node preds[levelmax];
5.    Node curr;
6.    Node pred;
7.    int retry=-1;
8.
9.  start_from_scratch_d:
10.   retry++;
11.   if (retry >= max_invalidation_abort_retries) {
12.     acquire_lock (global_lock);
13.     ret=delete_sec(key,info);
14.     release_lock(global_lock);
15.     return ret;
16.   }

17. pred=set.head;
18.   for ( h = levelmax-1; h >= 0; h--) {
19.     curr = pred.next[h];
20.     while (key > curr.key) {
21.       pred=curr;
22.       curr = pred.next[h];
23.     }
24.     preds[h] = pred;
25.   }
26.
27.   status=tx_start(max_tx_retries, set.global_lock); //begin
                                                        //transaction
28.   if (status == INVALIDATION_ABORT){
29.     goto start_from_scratch_d;
30.   }
31.   else {
32.     if (curr.key == key) {
33.       nodeHeight=curr.height;
34.       //check consistency
35.       for(h = 0; h < nodeHeight; h++) {
36.         if (preds[h].next[h] != curr
                || preds[h].state == DELETED ){
37.
38.           if (_xtest())
39.             _xabort(0xaa);       //force an abort
40.           else {
41.             release_lock(global_lock);
42.             goto start_from_scratch_d;
43.           }
44.         }
45.       }
46.
47.       //update fields
48.       curr.state = DELETED;  // Linearization point in
                                 // fallback lock mode for
                                 // successful deletion
```

```
47.      }
48.    }
49.  }
50.
51.  //update fields
52.  for(h = 0; h < nodeHeight; h++ )
53.    newNode.next[h] = succs[h];
54.
55.  for(h = 0; h < nodeHeight; h++ )
56.    preds[h].next[h] = newNode;
57.
58.  newNode.state = INSERTED; // Linearization point in
                               // fallback lock mode
                               // for successful insertion
59.  //commit
60.  tx_end(global_lock);     // Linearization point in
                               // transactional mode
61.  return 1;
62. }
```

```
49.    for(h = nodeHeight-1; h ≥ 0; h--) {
50.      preds[h].next[h] = curr.next[h];
51.    }
52.
53.    //commit
54.    tx_end(sglobal_lock);   // Linearization point in
                                 // transactional mode
                                 // successful deletion
55.    return 1;
56.  }
57.  else {          // Linearization point (transactional/
                     // fallback mode) for unsuccessful
                     // deletion
58.    tx_end(global_lock);
59.    return 0;
60.  }
61. }
62. }
```

(γ)

```
1.  int tx_start(int num_retries, lock fallback_lock)
2.  {
3.     int status = 0;
4.     int aborts = num_retries;
5.
6.     while (1) {
7.        /* Avoid lemming effect. */
8.        while ((!lock_is_free(fallback_lock));
9.
10.       status = _xbegin();
11.       if (status == _XBEGIN_STARTED) {
12.             if ((!lock_is_free(fallback_lock))
13.                   _xabort(0xff);
14.             return num_retries - aborts;
15.       }
16.
17.       /* Abort comes here. */
18.
19.       if (_XABORT_CODE(status) == 0xaa)
20.          return INVALIDATION_ABORT;
21.
22.       if (--aborts <= 0) {
23.             acquire_lock(fallback_lock);
24.             return num_retries - aborts;
25.       }
26.    }
27.
28.       /* Unreachable. */
29.       return -1;
30. }
```

(δ)

```
1.  void tx_end(lock fallback_lock)
2.  {
3.     if (lock_is_free(fallback_lock))
4.        _xend();
5.     else
6.        release_lock (fallback_lock);
7.     return;
8.  }
```

(ε)

```
1.  boolean contains(int  key){
2.    boolean result=false;
3.    Node curr, pred;
4.    pred= head;
5.    for ( int h = levelmax-1; h >= 0; h--)  {
6.          curr = pred.next[h];
7.          while (key > curr.key)  {
8.                pred=curr;
9.                curr = pred.next[h];
10.         }
11.   }
12.   if (key == curr.key) {
13.         while(curr.state == INITIAL);
14.         if (curr.state != DELETED) result=true;
15.   }
16.   return result;
17.  }
```

Σχήμα 4.3 *Fine-Grained HTM Skip List : ψευδοκώδικες για (α),(β) ενημέρωση δομής (γ),(δ) διαχείριση transactions, (ε) αναζήτηση στοιχείου.*

Λειτουργίες Εισαγωγής και Διαγραφής

Στο σχήμα 4.3 παρουσιάζουμε τους αλγορίθμους επεξεργασίας και αναζήτησης δεδομένων καθώς και τους αλγόριθμους επεξεργασίας transaction. Όταν θέλουμε να ενημερώσουμε τη δομή, είτε με insert είτε με delete, ακολουθούμε μια παρόμοια διαδικασία. Αρχικά προσπελαύνουμε τη δομή εκτός transaction ενημερώνοντας τους πίνακες προκατόχων και διαδόχων η μόνο προκατόχων για delete μέχρι να φτάσουμε στο κατώτερο επίπεδο με στόχο να βρούμε το στοιχείο με το ζητούμενο κλειδί που θέλουμε να διαγράψουμε ή το σημείο όπου θέλουμε να πραγματοποιήσουμε εισαγωγή (γραμμές 17-26 για insert / 17-25 για delete). Στη συνέχεια εντός transaction ελέγχουμε : 1) αν οι προκάτοχοι και οι διάδοχοι βρίσκονται ακόμα στη δομή και δεν έχουν διαγραφεί, 2) αν οι προκάτοχοι δείχνουν στους διαδόχους (για insert) ή στον κόμβο που θέλουμε να διαγράψουμε (για delete) (γραμμές 39 / 36). Στη περίπτωση που ο έλεγχος επιτύχει τότε επειδή έχουμε τοποθετήσει όλη αυτήν τη πληροφορία στο read set είμαστε σίγουροι ότι έχουμε εξασφαλίσει την εγκυρότητα της λειτουργίας καθώς εάν πριν ολοκληρώσουμε το transaction κάποιο άλλο νήμα αλλάξει κάποιον δείκτη ή διαγράψει κάποιο στοιχείο που έχουμε στο read set μας τότε θα δεχθούμε conflict abort και θα απορρίψουμε την λειτουργία χωρίς να ενημερώσουμε την μνήμη για τις αλλαγές που πιθανόν να έχουμε κάνει. Αν δεν δεχθούμε abort τότε προχωράμε ενημερώνοντας τη δομή. Υπενθυμίζουμε ότι όλες οι αλλαγές που κάνουμε θα εμφανιστούν στην δομή αφότου πραγματοποιήσουμε transaction commit. Στη περίπτωση της εισαγωγής στοιχείου αρχικά δημιουργούμε έναν κόμβο με το ζητούμενο κλειδί και τυχαίο ύψος (29,30), ύστερα ενημερώνουμε τους δείκτες του νέου κόμβου να δείχνουν στους διαδόχους (52,53) και στη συνέχεια ενημερώνουμε τους δείκτες των προκατόχων να δείχνουν στο νέο κόμβο (55,56). Στη περίπτωση της διαγραφής ενημερώνουμε τους δείκτες των προκατόχων να δείχνουν στους διαδόχους του στοιχείου που διαγράφουμε (49,50). Οι αλλαγές στην εισαγωγή γίνονται από το κατώτερο προς το ανώτερο επίπεδο ενώ στη διαγραφή από το ανώτερο προς το κατώτερο.

Τι γίνεται όμως όταν διαπιστώσουμε κάποια παράβαση στον έλεγχο που κάνουμε εντός transaction ή όταν δεχτούμε κάποιο abort? Καταρχήν, υπάρχουν δύο λόγοι για τους οποίους μπορούμε να δεχτούμε conflict abort. Πρώτον επειδή κάποιο άλλο νήμα έγραψε σε δεδομένο που είναι στο read ή στο write set μας (write conflict). Αυτό μπορεί να χαρακτηριστεί θετικό abort καθώς συμβάλει στην διατήρηση εγκυρότητας. Δεύτερον επειδή κάποιο άλλο νήμα διάβασε δεδομένο που έχουμε στο write set μας (read conflict). Το write set αποτελείται από όλες τις cache lines που περιέχουν δεδομένα που έχουμε τροποποιήσει κατά τη διάρκεια ενημέρωσης της δομής, δηλαδή όταν αλλάζουμε τους δείκτες. Αυτό μπορεί να χαρακτηριστεί ως αρνητικό abort καθώς έχουμε διατήρηση εγκυρότητας είτε γίνει το abort γίνει είτε αν δεν γινόταν συνεπώς έχουμε πλεονάζον abort. Στην επόμενη ενότητα εξηγούμε γιατί ισχύει αυτό.

Ας εξηγήσουμε πρώτα τη περίπτωση που δεχόμαστε abort. Με το που δεχόμαστε abort αρχικά απορρίπτουμε ότι έχουμε στο read και στο write set χωρίς να ενημερώσουμε τη μνήμη. Στη συνέχεια τρέχουμε κώδικα του fallback path. Συγκεκριμένα πηγαίνουμε στη γραμμή 17,σχ. γ, εκτός transaction. Η τιμή της μεταβλητής status έχει τον κωδικό του είδους του abort που έγινε. Η συνθήκη στη γραμμή 19 `if (_XABORT_CODE(status) == 0xaa)` δεν ικανοποιείται καθώς η μεταβλητή status θα πάρει τη τιμή 0xaa μόνο αν έχει προκληθεί explicit abort με κωδικό 0xaa μέσω της εντολής `_xabort(0xaa)` . Συνεπώς προχωράμε στη

γραμμή 22 όπου εξετάζουμε αν ο αριθμός των aborts από τη στιγμή που καλέσαμε την tx_start() έχει υπερβεί τον μέγιστο αριθμό επαναλήψεων , π.χ 15 , που έχουμε καθορίσει. Αν δεν τον έχει υπερβεί επιστέφουμε στο while loop και ξεκινάμε νέο transaction αλλιώς καταλαμβάνουμε το κλειδί και εκτελούμε το critical section ως μοναδικοί updaters. Είτε είμαστε σε lock mode είτε είμαστε σε transactional mode δεν εκτελούμε όλη τη λειτουργία ξανά, δηλαδή δεν επαναπροσπελαύνουμε τη δομή. Αντίθετα επαναλαμβάνουμε τον έλεγχο εγκυρότητας και τη διαδικασία ενημέρωσης των δεικτών.

Η περίπτωση που βρίσκουμε παράβαση στον έλεγχο που κάνουμε εντός transaction αντιμετωπίζεται διαφορετικά από την περίπτωση που δεχόμαστε abort. Εάν λοιπόν εντοπίσουμε κάποια παράβαση σημαίνει ότι οι πίνακες προκατόχων και διαδόχων δεν είναι ενημερωμένοι σωστά καθώς θα συνέβη κάποια αλλαγή από κάποιο άλλο νήμα πριν πραγματοποιήσουμε τον έλεγχο, συνεπώς δεν μπορούμε να προχωρήσουμε στην ενημέρωση της δομής. Θα πρέπει επομένως να επαναπροσπελάσουμε τη δομή ώστε να ανανεώσουμε τους πίνακες προκατόχων και διαδόχων. Για να το πετύχουμε αυτό τερματίζουμε με explicit abort `_xabort(0xaa)` το transaction (43/39) ή ελευθερώνουμε το κλειδί (45/41) αν είμαστε σε lock mode. Ελέγχουμε σε τι mode είμαστε με την εντολή `_xtest()` (42/38) η οποία επιστρέφει 1 αν είμαστε σε transactional mode. Αν είμαστε σε transactional mode τότε μέσω του ελέγχου που κάνουμε στη γραμμή 19 σχ. γ , `if (_XABORT_CODE(status) == 0xaa)` , διαπιστώνουμε ότι κάναμε explicit abort και επιστέφουμε τον κωδικό INVALIDATION ABORT. Συνεπώς επιστέφοντας από το fallback path βρισκόμαστε στη γραμμή 32/27 σε non-transactional και non-locking mode, διαπιστώνουμε ότι έγινε invalidation abort και πηγαίνουμε μέσω της εντολής goto στη γραμμή 9/9 . Αν είμαστε σε lock mode ελευθερώνουμε το κλειδί (45,46 / 41,42) και πηγαίνουμε στη γραμμή 9/9. Όπως μετράγαμε τον αριθμό των aborts στο fallback path έτσι και εδώ μετράμε τον αριθμό των invalidation aborts. Εάν υπερβούμε το ανώτερο όριο που έχουμε θέσει , π.χ 10 invalidation aborts, τότε καταλαμβάνουμε το κλειδί και εκτελούμε τη λειτουργία χρησιμοποιώντας τους σειριακούς αλγορίθμους (9-16 / 9-16). Σε αυτή τη περίπτωση δεν χρειάζεται να πραγματοποιήσουμε ελέγχους καθώς είμαστε οι μοναδικοί updaters. Επίσης σύμφωνα με τα linearization points που έχουμε θέσει και θα εξηγήσουμε στην επόμενη ενότητα επιτυγχάνουμε συγχρονισμό με παράλληλες εκτελέσεις contains(). Εάν δεν έχουμε υπερβεί το ανώτερο όριο των invalidation aborts τότε επαναλαμβάνουμε την διαδικασία προσπέλασης της δομής σε non-transactional και non-locking mode.

Λειτουργία Αναζήτησης

Η συνάρτηση contains() είναι ακριβώς η ίδια με την contains() που είδαμε στον coarse-grained αλγόριθμο.

4.3.2 Linearization Points

- **(Linearization points για αποτυχημένα updates)** : Ο έλεγχος όταν κάνουμε insert για το αν ο κόμβος ήδη υπάρχει γίνεται πριν το critical section (μόλις τελειώσει το traverse, γραμμή 27) και αν βρεθεί τότε το σημείο αυτό αποτελεί το linearization point ενός αποτυχημένου insert. Στη περίπτωση του removal ο έλεγχος γίνεται μέσα στο critical section (γραμμή 32) και αν διαπιστωθεί ότι δεν υπάρχει τότε το σημείο αυτό είναι το linearization point για το αποτυχημένο removal .
- **(Linearization points για πετυχημένα updates)** : Όταν ένα update κάνει transaction commit τότε το σημείο αυτό αποτελεί linearization point για το αντίστοιχο update (γρ. 60 / 54). Όταν ένα update εκτελείται με lock τότε αν το update είναι delete θέτουμε το status_flag του κόμβου με DELETED πριν πραγματοποιήσουμε το update (δηλαδή πριν ενημερώσουμε τους pointers) (γρ. 48) και το σημείο αυτό αποτελεί linearization point για το delete με lock. Για το insert θέτουμε το status_flag με INSERTED μετά την ολοκλήρωση του update και πριν τη απελευθέρωση του lock (γρ 58) . Αυτό αποτελεί το linearization point για το insert με lock.
- **(Linearization points για contains)** : Αν ένα thread εκτελεί contains τότε αν ο κόμβος που ψάχνει γίνεται delete από κάποιο άλλο thread (status_flag =DELETED) τότε επιστρέφει false . Αν ο ζητούμενος κόμβος γίνεται insert τότε περιμένει (σπινιάρει) μέχρι το status_flag να γίνει INSERTED (γρ. 13) . Τα Linearization points είναι ίδια όπως στη coarse-grained HTM περίπτωση.

Σημείωση1 (parsing). Όταν ένα thread εκτελεί update με lock και ένα άλλο κάνει parsing , τότε υπάρχει περίπτωση το δεύτερο thread να προσπεράσει ένα κόμβο που δεν έχει γίνει ακόμα insert η να προσπελάσει ένα κόμβο που γίνεται delete . Αυτές οι περιπτώσεις δεν αποτελούν πρόβλημα εγκυρότητας σύμφωνα με τα linearization points που έχουμε θέσει. Αν το parsing thread εκτελεί update τότε οι περιπτώσεις αυτές μπορούν να δημιουργήσουν inconsistencies τα οποία όμως θα βρεθούν στον έλεγχο που θα γίνει στη συνέχεια.

Σημείωση2 (γιατί τα read conflict προκαλούν αρνητικά abort). Όπως είδαμε στην προηγούμενη ενότητα όταν κάποιο άλλο νήμα διαβάσει δεδομένο που έχουμε στο write set μας, το οποίο αποτελείται από τις αλλαγές στους δείκτες που επιχειρήσαμε, τότε θα δεχτούμε conflict abort και θα απορρίψουμε τη λειτουργία μας. Αυτό είναι ένα αρνητικό abort το οποίο δεν συνεισφέρει στην διατήρηση εγκυρότητας της λειτουργίας. Ο λόγος που ισχύει αυτό είναι ότι τα υπόλοιπα νήματα που διαβάζουν στοιχεία του write set στη πραγματικότητα διαβάζουν τις προηγούμενες εκδόσεις των στοιχείων και όχι τις τροποποιημένες. Συνεπώς αν το transaction έκανε commit και όχι abort τότε όποια αλλαγή έγινε λόγω commit 1) δεν επηρεάζει κάποιο νήμα που εκτελούσε contains() καθώς σύμφωνα με τα linearization points που θέσαμε το αποτέλεσμα της αναζήτησης είναι ακριβώς το ίδιο με το να είχαμε πολλούς readers και έναν updater και 2) δεν επηρεάζει την εγκυρότητα της λειτουργίας των νημάτων που εκτελούν update, καθώς αν οι αλλαγές που έγιναν επηρεάσουν τους πίνακες διαδόχων και προκατόχων αυτό θα διαπιστωθεί είτε με τον έλεγχο που θα γίνει είτε μέσω write conflict -> abort.

4.3.3 Βελτιστοποιήσεις

Παρατηρώντας τους αλγορίθμους εισαγωγής και διαγραφής δεδομένων διαπιστώσαμε δυο αλλαγές που θεωρητικά θα βελτίωναν την γενική απόδοση.

- Όταν εκτελούμε εισαγωγή δεδομένου αυτό που κάνουμε σύμφωνα με τον αλγόριθμο του σχήματος 4.3 είναι να δημιουργούμε νέο κόμβο με νέο ύψος κάθε φορά που βρίσκουμε inconsistency και επαναπροσπελαύνουμε τη δομή. Αυτό θα μπορούσε να αποφευχθεί εάν χρησιμοποιούσαμε ένα flag που θα μας επέτρεπε να ελέγχουμε αν είναι η πρώτη φορά που προσπελαύνουμε τη δομή οπότε και να δημιουργούσαμε έναν νέο κόμβο, ή αν δεν είναι η πρώτη φορά να χρησιμοποιούσαμε τον ήδη υπάρχον. Αυτή η αλλαγή έχει δοκιμαστεί προγραμματιστικά αλλά για αδιευκρίνιστους λόγους δεν βελτιώθηκε η απόδοση, μάλιστα χειροτέρευσε.
- Όταν εκτελούμε update και βρίσκουμε inconsistency, αυτό που κάνουμε σύμφωνα με τον αλγόριθμο του σχήματος 4.3 είναι να επαναπροσπελάσουμε τη δομή από το ανώτερο επίπεδο ώστε να ενημερώσουμε τους πίνακες προκατόχων και διαδόχων από την αρχή. Μια βελτιστοποίηση θα ήταν αντί να επαναπροσπελάσουμε τη δομή από το ανώτερο επίπεδο να ξεκινάγαμε από το επίπεδο που διαπιστώθηκε το inconsistency. Μέχρι στιγμής δεν έχουμε καταφέρει να υλοποιήσουμε προγραμματιστικά αυτή τη βελτιστοποίηση καθώς οι εντολές του RTM δεν παρέχουν τρόπο να διασώζουμε τιμές που δημιουργήθηκαν εντός ενός transaction που στη συνέχεια απορρίπτεται. Η βελτιστοποίηση αυτή αποτελεί μελλοντική επέκταση.

Κεφάλαιο 5

Αξιολόγηση αλγορίθμων

Σκοπός του παρόντος κεφαλαίου είναι να κατανοήσουμε καλύτερα τους αλγορίθμους που έχουμε δει μέχρι τώρα μέσα από μια πιο πρακτική σκοπιά. Εκτελώντας πειράματα στοιχειοθετούμε τα επιχειρήματά μας και εντοπίζουμε τα προτερήματα και μειονεκτήματα κάθε τεχνικής.

Συγκεκριμένα, στην ενότητα 5.1 αρχικά παρουσιάζουμε το υπολογιστικό περιβάλλον όπου διεξάγουμε μετρήσεις και στη συνέχεια παρουσιάζουμε και σχολιάζουμε τα αποτελέσματα δυο πειραμάτων που πραγματοποιήσαμε. Ύστερα, στην ενότητα 5.2 αξιολογούμε την coarse-grained HTM εκδοχή. Στην ενότητα 5.3 εξετάζουμε κατά πόσο ένα σύστημα NUMA επηρεάζει την λειτουργία των HTM υλοποιήσεων καθώς και προτείνουμε ορισμένες λύσεις. Στην ενότητα 5.4 εξετάζουμε τι συμβαίνει όταν εκτελούνται παραπάνω νήματα από τον συνολικό αριθμό των πυρήνων του συστήματος και τέλος στην ενότητα 5.5 συνοψίζουμε, παρουσιάζοντας τα συμπεράσματά μας σχετικά με τα δυνατά και αδύναμα σημεία κάθε τεχνικής.

5.1 Πειραματική αξιολόγηση

5.1.1 Υπολογιστικό περιβάλλον

Στον παρακάτω πίνακα παρουσιάζουμε τα χαρακτηριστικά του μηχανήματος στο οποίο εκτελέσαμε τα πειράματά μας.

Όνομα		**Haswell-EP**
Επεξεργαστές		2 x Intel Xeon E5-2697 v3
# Πυρήνων		2 x 14
# Νημάτων		2 x 28
Συχνότητα Επεξεργαστή		2.6 GHz
L1 (Data)		32 KB, 8-way, 64B block size
L2		256 KB, 8-way, 64B block size
L3		35 MB, 20-way, 64B block size (shared per die)
Μνήμη		64 GB
Λειτουργικό Σύστημα		Debian 8.1
Πυρήνας Linux		4.0.4
GCC		4.9.2 with -O3 optimization

Πίνακας 5.1 *Στοιχεία υπολογιστικής πλατφόρμας.*

Όπως φαίνεται στον Πίνακα 1 ο server που χρησιμοποιούμε αποτελείται από δυο επεξεργαστές Intel Xeon E5-2697 v3. Η μνήμη είναι διαμοιρασμένη στους δυο επεξεργαστές σύμφωνα με το μοντέλο NUMA (Non-uniform memory access). Σύμφωνα με το μοντέλο αυτό ο χρόνος πρόσβασης στη μνήμη εξαρτάται από τη σχετική θέση που έχουν η τοποθεσία μνήμης και ο επεξεργαστής. Κάθε επεξεργαστής έχει τη δική του τοπική μνήμη και η πρόσβαση σε αυτήν είναι σημαντικά ταχύτερη από τη πρόσβαση σε τοπική μνήμη άλλου επεξεργαστή.

Επιπλέον, το παραπάνω σύστημα χρησιμοποιεί τεχνολογία Hyper-threading σύμφωνα με την οποία κάθε φυσικός πυρήνας εμφανίζεται στο λειτουργικό σύστημα ως δυο με αποτέλεσμα να επιτρέπεται ταυτόχρονη δρομολόγηση δυο διεργασιών ανά φυσικό πυρήνα. Έτσι ο αριθμός των Hardware threads είναι στην ουσία διπλάσιος του αριθμού των πυρήνων. Κάθε ζευγάρι hyper-threaded νημάτων μοιράζεται τους διαθέσιμους επεξεργαστικούς πόρους του πυρήνα στο οποίο ανήκει.

Οι επεξεργαστές υποστηρίζουν HTM μέσω της επέκτασης εντολών TSX. Τα μεγέθη των transactional buffers είναι τα ακόλουθα :

Read set (Total / Per HW thread)		4 MB / 2 MB
Write set (Total / Per HW thread)		22 KB / 11 KB

Πίνακας 5.2 *Μεγέθη transactional buffers του Haswell-EP.*

5.1.2 Γενικές πληροφορίες

Προκειμένου να δοκιμάσουμε πειραματικά τους αλγορίθμους των προηγούμενων κεφαλαίων υλοποιήσαμε την coarse-grained και την fine-grained HTM skip list στη γλώσσα προγραμματισμού C και τις συγκρίναμε με τις lock-based και lock-free υλοποιήσεις που παρέχονται από την βιβλιοθήκη ASCYLIB [11]. Συγκεκριμένα οι υλοποιήσεις που εξετάζουμε είναι οι εξής :

- **seq** : υλοποίηση σειριακής skip list χωρίς συγχρονισμό. Χρησιμεύει μόνο για σύγκριση απόδοσης με τις υπόλοιπες υλοποιήσεις καθώς παράγει λάθος αποτελέσματα.
- **lb** : Herlihy et al. skip list (lock-based) [12]
- **lf** : Fraser skip list με βελτιστοποιήσεις του Herlihy (lock-free) [13]
- **tx_fg** : Fine-Grained HTM skip list
- **tx_cg** : Coarse-Grained HTM skip list

Γενικότερα στις μετρήσεις και στα αποτελέσματα που ακολουθούν ισχύουν τα ακόλουθα εκτός εάν αναφέρεται διαφορετικά :

- Κάθε μέτρηση διαρκεί ένα δευτερόλεπτο, το οποίο αποτελεί σύνηθες επιλογή χρονικού διαστήματος για τέτοιου είδους πειράματα και στο οποίο εκτελείται αρκετά μεγάλος αριθμός λειτουργιών. Τα αποτελέσματα προκύπτουν ως μέσοι όροι τεσσάρων με έξι επαναλαμβανόμενων δοκιμών.
- Κάθε software thread προσδένεται (γίνεται pinned) σε ένα hardware thread.
- Για την αποφυγή false conflicts χρησιμοποιούμε ευθυγράμμιση (alignment) έτσι ώστε κάθε κόμβος της skip list να καταλαμβάνει ακριβώς τρεις cache lines.
- Καμία υλοποίηση δεν χρησιμοποιεί ανάκτηση μνήμης.
- Η δομή αρχικοποιείται από ένα νήμα στην αρχή πριν ξεκινήσει η μέτρηση.
- Οι λειτουργίες ενημέρωσης είναι πάντα 50% insert, 50% delete, δηλαδή μετά από κάθε πετυχημένο insert ακολουθεί delete και αντίστροφα αλλιώς επαναλαμβάνουμε το insert μέχρι να επιτύχει. Με τον τρόπο αυτό διατηρούμε το μέγεθος της δομής σταθερό.
- Έχουμε θέσει τον αριθμό των φορών που ξαναεκτελείται ένα transaction πριν καταληφθεί το κλείδωμα σε 30 ενώ τον αριθμό επαναπροσπελάσεων της δομής όταν βρεθεί inconsistency σε 15.

5.1.3 Πείραμα 1 : Alternate

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα του πρώτου πειράματος που εκτελέσαμε. Για τη πραγματοποίησή του θέτουμε το εύρος κλειδιών στην τιμή 100.000.000 και αρχικοποιούμε τη δομή με 1.000.000 , 100.000 , 10.000 ή 1.000 κόμβους ανάλογα με τη δοκιμή που θέλουμε να τρέξουμε. Εισάγουμε στοιχεία με τυχαία τιμή στο εύρος κλειδιών που προαναφέραμε οπότε περίπου το 99% των εισαγωγών θα είναι πετυχημένο (δεν θα βρουν στοιχείο στη δομή με το ίδιο κλειδί). Κάθε λειτουργία διαγραφής διαγράφει το στοιχείο που προστέθηκε από τη πιο πρόσφατη λειτουργία εισαγωγής από το ίδιο νήμα και όχι σύμφωνα με κάποιο τυχαίο κλειδί. Έτσι αναμένουμε το 100% των διαγραφών να βρουν το κλειδί που ψάχνουν. Τέλος, οι αναζητήσεις ψάχνουν κάποιο τυχαίο κλειδί στη δομή αν η τελευταία λειτουργία ενημέρωσης από το ίδιο νήμα ήταν διαγραφή ή ψάχνουν το στοιχείο που μόλις εισήχθη αν η τελευταία λειτουργία ενημέρωσης ήταν εισαγωγή. Συνεπώς αναμένουμε περίπου το 50% των αναζητήσεων να βρουν το κλειδί που ψάχνουν.

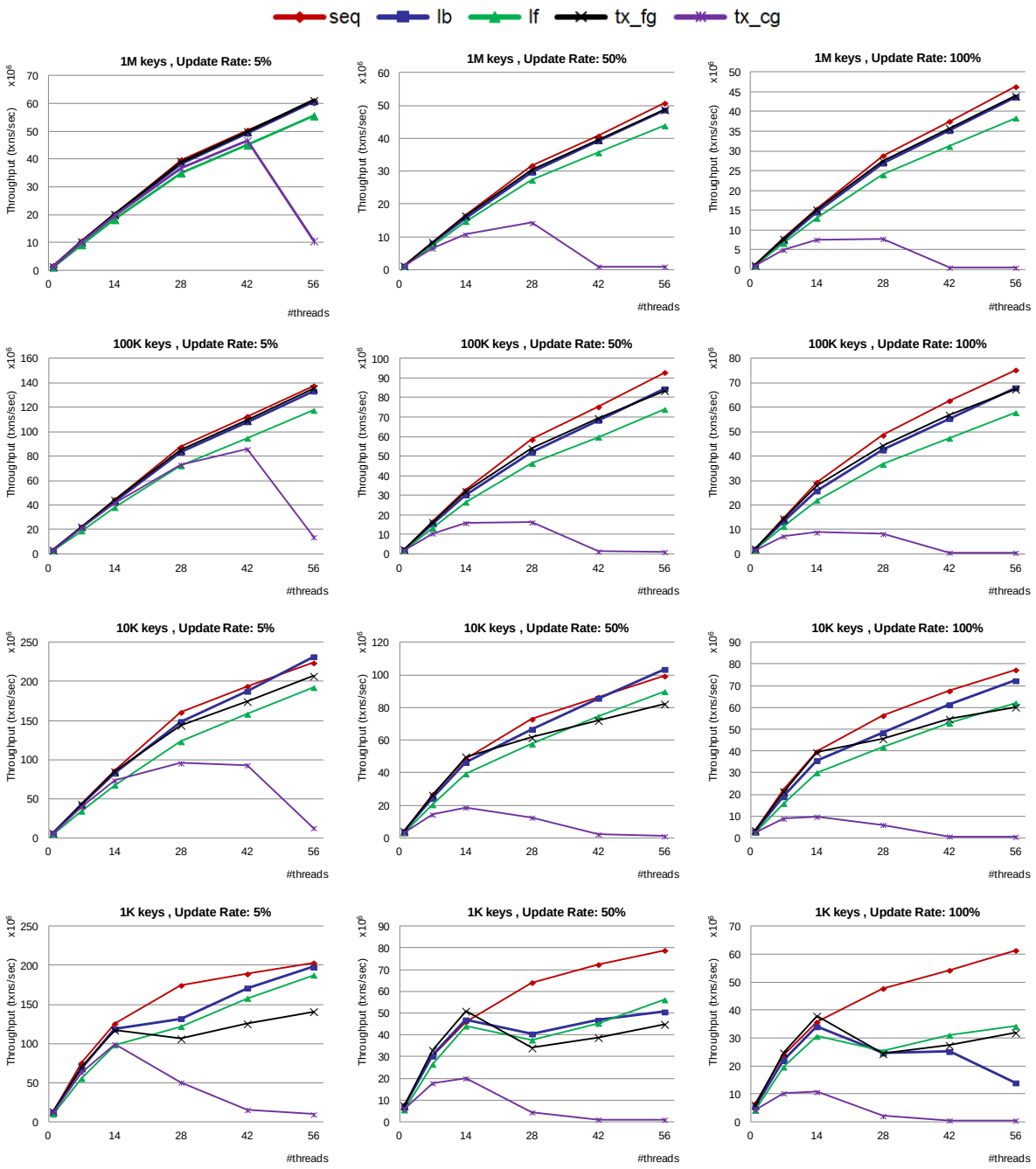
Στο σχήμα 5.1 παρουσιάζουμε τα αποτελέσματα των μετρήσεων σε σχέση με την απόδοση (throughput). Στον οριζόντιο άξονα είναι ο αριθμός των νημάτων που τρέχουνε και στον κάθετο η απόδοση που μετριέται σε λειτουργίες ανά δευτερόλεπτο. Στο σχήμα 5.2 παρουσιάζουμε τα στατιστικά σε σχέση με τα transactions (commits,aborts) για την fine-grained και την coarse-grained HTM υλοποίηση.

Για μεγάλα μεγέθη δομής (1 εκατ., 100 χιλ. κλειδιά) οι περιπτώσεις όπου δύο ή περισσότερα νήματα αναγκάζονται να συγχρονιστούν για τη πρόσβαση σε κοινά δεδομένα είναι λίγες. Με άλλα λόγια, τα νήματα εκτελούν εντολές συγχρονισμού (κλειδώματα, ατομικές εντολές, εντολές transactions) εκ των οποίων η πλειοψηφία είναι χαμένος χρόνος. Λόγω του συγκριτικά μεγαλύτερου κόστους των ατομικών εντολών η lock free υλοποίηση υστερεί σε απόδοση σε σχέση με τις υπόλοιπες στα μεγάλα μεγέθη της δομής, κάτι που φαίνεται στις γραφικές του σχήματος 5.1 .

Όσο το μέγεθος της δομής μικραίνει (10 χιλ. κλειδιά , 1000 κλειδιά) και ο ρυθμός ενημερώσεων μεγαλώνει (50% , 100%) τόσο αυξάνονται οι συγκρούσεις και η ανάγκη για συγχρονισμό μεγαλώνει. Στη περίπτωση όπου μεγιστοποιούνται οι συγκρούσεις (1000 κλειδιά, 100% updates) παρατηρούμε ότι η lock free υλοποίηση υπερέχει ελαφρώς. Σε ορισμένες περιπτώσεις παρατηρούμε ότι η lock based περίπτωση υπερέχει της σειριακής. Αυτό οφείλεται στο ότι η σειριακή υλοποίηση δεν έχει ορισμένες από τις βελτιστοποιήσεις που έχουν οι άλλες υλοποιήσεις, ιδιαίτερα στις λειτουργίες contains ούτως ώστε να αποφευχθούν περιπτώσεις όπως segmentation fault.

Η coarse-grained HTM υλοποίηση έχει ανταγωνιστική επίδοση όταν ο ρυθμός ενημερώσεων είναι χαμηλός (5%) και ο αριθμός των threads είναι μικρότερος ή ίσος με 28. Παρατηρούμε ότι όσο ο ρυθμός ενημερώσεων αυξάνεται τόσο μειώνεται η παρατηρούμενη κλιμακωσιμότητα λόγω των αυξημένων aborts όπως βλέπουμε και στο σχήμα 5.2. Μάλιστα για 42 και 56 νήματα το ποσοστό των committed transactions πέφτει κάτω από 2%.

Αντίθετα η fine-grained HTM υλοποίηση έχει παραπλήσια επίδοση με τις lock-based και lock-free υλοποιήσεις σε όλα τα μεγέθη δομής και σε όλους τους ρυθμούς ενημερώσεων. Επίσης για μεγέθη δομής μεγαλύτερα των 10 χιλ. το ποσοστό των commits είναι μεγαλύτερο του 50%.



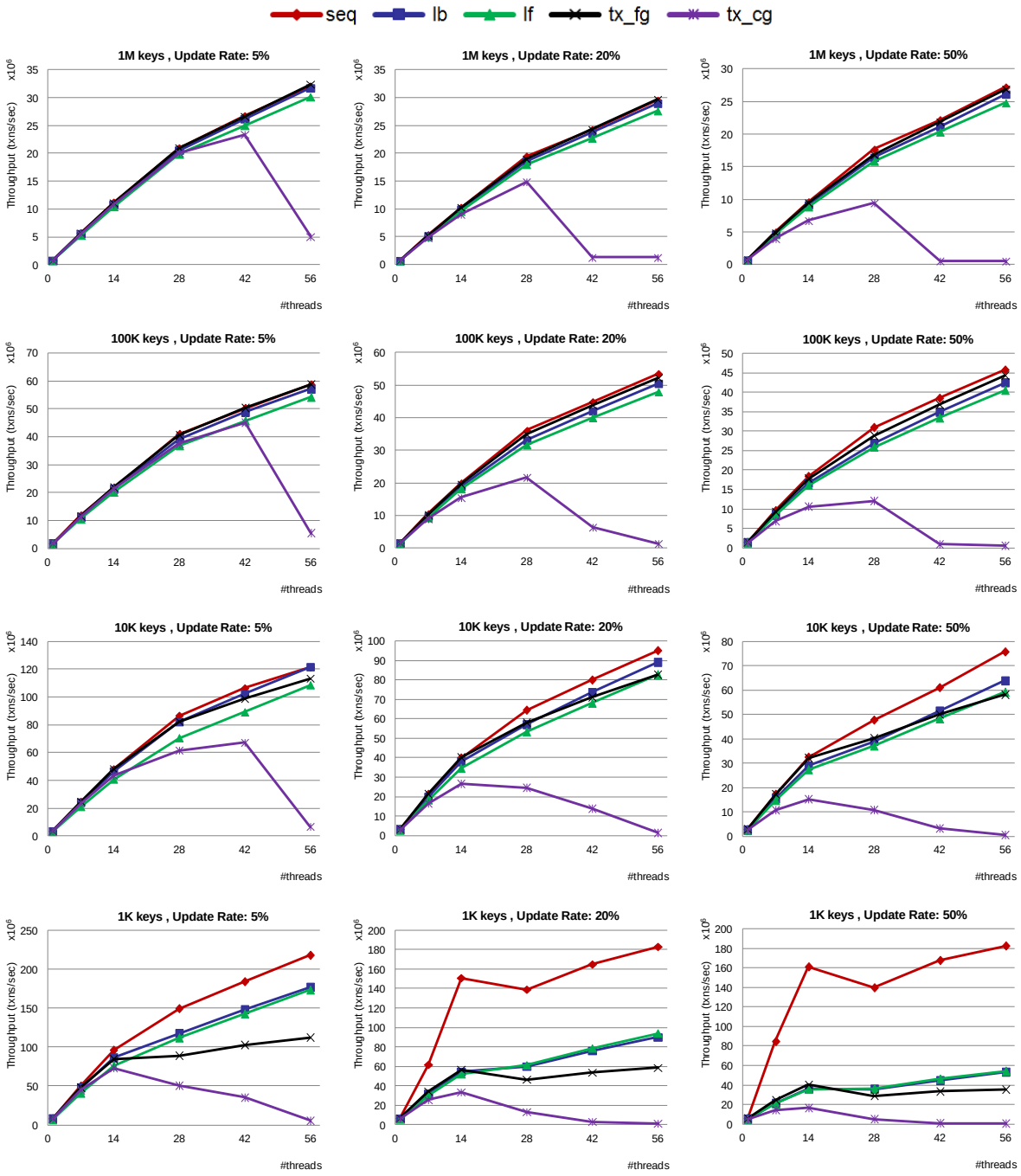
Σχήμα 5.1 *Επίδοση παράλληλων Skip List υλοποιήσεων, πείραμα 1. Στον οριζόντιο άξονα είναι ο αριθμός των νημάτων που τρέχουμε και στον κάθετο είναι η απόδοση (throughput) σε εκατομμύρια λειτουργίες το δευτερόλεπτο.*

Σχήμα 5.2 *Αριθμός committed και aborted transactions, πείραμα 1.*

5.1.4 Πείραμα 2 : Random

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα του δεύτερου πειράματος που εκτελέσαμε. Και εδώ , αρχικοποιούμε τη δομή με 1.000.000 , 100.000 , 10.000 ή 1.000 κόμβους αλλά θέτουμε το εύρος κλειδιών να είναι ίσο με εύρος = 2×αρχικό μέγεθος. Οι εισαγωγές και οι διαγραφές γίνονται με βάση τυχαία επιλογή κλειδιών στο εύρος που ισχύει για την τρέχουσα δοκιμή με αποτέλεσμα περίπου το 50% των εισαγωγών και το 50% των διαγραφών να επιτυγχάνουν. Οι αναζητήσεις επίσης ψάχνουν σύμφωνα με κάποιο τυχαίο κλειδί , συνεπώς αναμένουμε περίπου το 50% των αναζητήσεων να βρουν το κλειδί που ψάχνουν.

Σύμφωνα με τις γραφικές που ακολουθούν τα συμπεράσματα που βγάζουμε από αυτό το πείραμα είναι τα ίδια με αυτά που βγάλαμε από το προηγούμενο.

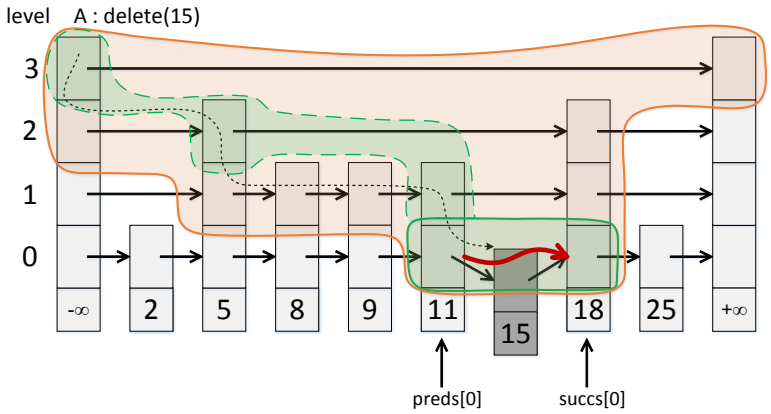


Σχήμα 5.3 *Επίδοση παράλληλων Skip List υλοποιήσεων, πείραμα 2.*

Σχήμα 5.4 *Αριθμός committed και aborted transactions, πείραμα 2.*

5.2 Πόσο πολύ νόημα έχει η Coarse-Grained εκδοχή ?

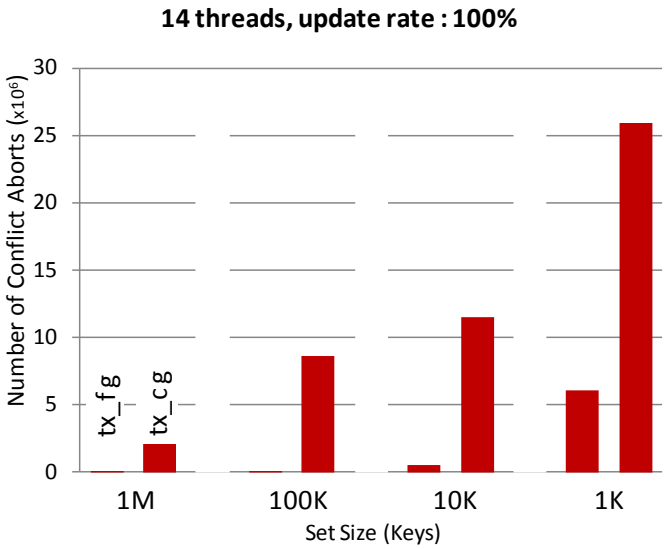
Αν ανατρέξουμε στην ενότητα 4.2.3 του προηγούμενου κεφαλαίου θα θυμηθούμε ότι το μεγάλο πρόβλημα της coarse-grained HTM εκδοχής είναι το υπερβολικά μεγάλο της footprint που οδηγεί σε ανεπιθύμητα conflict aborts και capacity overflows. Αντίθετα, το footprint της fine-grained υλοποίησης θεωρητικά ταυτίζεται με το ιδανικό footprint που απαιτείται για την διασφάλιση εγκυρότητας. Στην πράξη τα footprints και των δύο υλοποιήσεων είναι λίγο μεγαλύτερα καθώς όταν διαβάζουμε ή γράφουμε δεδομένα τοποθετούμε στα read και write sets ολόκληρες τις cache lines που περιέχουν αυτά τα δεδομένα. Συνεπώς στη περίπτωση της fine-grained υλοποίησης το footprint είναι λίγο μεγαλύτερο από το ιδανικό με αποτέλεσμα να παρατηρούνται μερικά ανεπιθύμητα aborts λόγω false sharing conflicts.



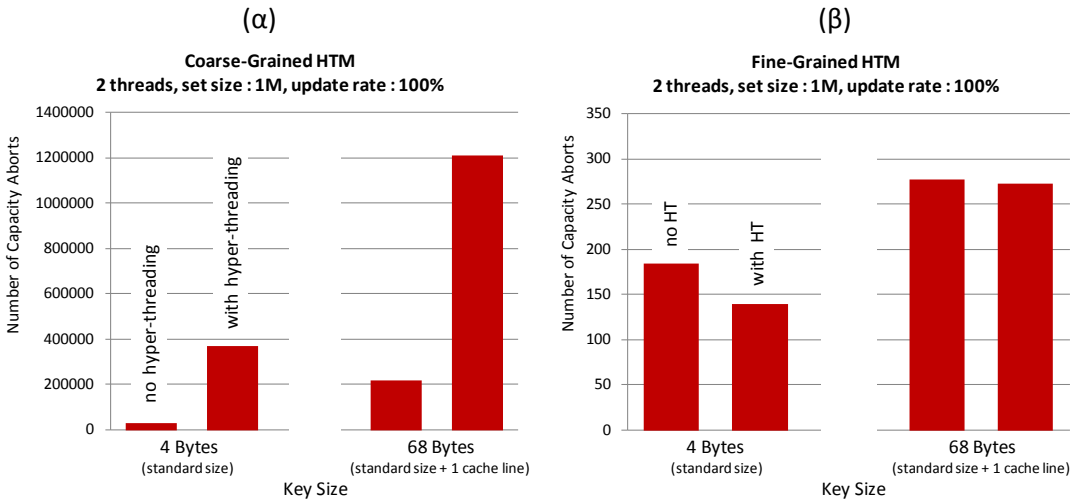
Σχήμα 5.5 *Footprints των coarse-grained και fine-grained HTM transactions. Με μαύρη διακεκομμένη φαίνεται η πορεία προσπέλασης της δομής. Με πορτοκαλί πλαίσιο φαίνεται το footprint του coarse-grained HTM transaction ενώ το "θεωρητικό" footprint του fine-grained HTM transaction είναι το πράσινο πλαίσιο που περικλείεται από την συνεχόμενη γραμμή και ταυτίζεται με το ιδανικό footprint. Το πράσινο πλαίσιο που περικλείεται από τη διακεκομμένη είναι μια πιθανή επέκταση του fine-grained HTM footprint λόγω επιπλέον δεδομένων που μπορεί να βρίσκονται στις ίδιες cache lines με δεδομένα που περιέχονται στο ιδανικό footprint.*

Η υπόθεσή μας σχετικά με τα ανεπιθύμητα conflict aborts επιβεβαιώνεται πειραματικά στο σχήμα 5.6 παρακάτω. Μετράμε τα conflict aborts για μεγέθη δομής 1 εκατ., 100 χιλ., 10 χιλ. και 1000. Επιλέγουμε ρυθμό ενημερώσεων 100% και αριθμό νημάτων ίσο με 14. Επιλέξαμε αυτόν τον αριθμό νημάτων γιατί 1) ο αριθμός εκκινήσεων των transactions είναι παρόμοιος στις δυο υλοποιήσεις (λίγες περισσότερες η fg HTM), 2) είναι απενεργοποιημένο το hyper-threading και το NUMA και άρα δεν επηρεάζουν τις μετρήσεις.

Ας δούμε τώρα τη συμπεριφορά των HTM υλοποιήσεων για μεταβλητό μέγεθος κλειδιού σε περιπτώσεις όπου το hyper-threading είναι ενεργό ή ανενεργό. Τα αποτελέσματα φαίνονται στο σχήμα 5.7.



Σχήμα 5.6 *Αριθμός conflict aborts των δυο HTM υλοποιήσεων για 14 threads, 100% ρυθμό ενημερώσεων και μεγέθη skip lists 1 εκατ., 100 χιλ., 10 χιλ. και 1000.*



Σχήμα 5.7 *Αριθμός capacity aborts των δυο HTM υλοποιήσεων για 2 threads, 100% ρυθμό ενημερώσεων, μέγεθος skip list 1 εκατ., μεγέθη κλειδιού 4 bytes και 68 bytes (4 bytes + 1 cache line, όπου 1 cache line = 64 bytes) με το hyper-threading να είναι ενεργό ή ανενεργό.*

Παρατηρώντας τις γραφικές του σχήματος 5.7 διαπιστώνουμε και πειραματικά την επίδραση που έχουν τα footprints των υλοποιήσεων στη δημιουργία capacity aborts. Αρχικά, διαπιστώνουμε ότι "υπό φυσιολογικές" συνθήκες, δηλαδή όταν το μέγεθος κλειδιού είναι 4 bytes και το hyper-threading είναι απενεργοποιημένο, ο αριθμός των capacity aborts στην coarse-grained HTM εκδοχή είναι ήδη αρκετά μεγαλύτερος από τον αντίστοιχο αριθμό των aborts της fine-grained HTM εκδοχής (20000 έναντι 185). Στη συνέχεια βλέπουμε ότι η αύξηση του μεγέθους του κλειδιού κατά 1 μόλις cache line (64 bytes) οδηγεί σε σημαντική αύξηση των capacity aborts (έως και δέκα φορές) στην coarse-grained εκδοχή ενώ προκαλεί ανεπαίσθητη αύξηση στην fine-grained εκδοχή. Τέλος, παρατηρούμε ότι η επίδραση του hyper-threading, το οποίο μειώνει τους transactional

buffers ανά thread στο μισό, είναι κάτι παραπάνω από αισθητή στην coarse-grained εκδοχή, εκτοξεύοντας τα capacity aborts έως και 50 φορές πιο πάνω ενώ είναι ανεπαίσθητη στην fine-grained εκδοχή. Τα αποτελέσματα αυτά αναδεικνύουν εκτός από τη προβληματική συμπεριφορά του coarse-grained HTM, την θετική συμπεριφορά του fine-grained HTM. Το γεγονός ότι η πλειοψηφία των εντολών που αφορούν κλειδιά συναντώνται κατά τη φάση προσπέλασης και άρα εκτός transaction σε συνδυασμό με το μικρό footprint το καθιστά ανεξάρτητο του μεγέθους κλειδιού και της εφαρμογής hyper-threading.

Σύμφωνα με τα πειραματικά αποτελέσματα που έχουμε δει μέχρι στιγμής στο κεφάλαιο αυτό συμπεραίνουμε ότι η coarse-grained HTM αποτελεί καλή επιλογή για απλά data sets με μικρή συμφόρηση και μικρά μεγέθη κλειδιών. Οι περιορισμοί και οι παθολογίες του HTM, όπως ο υψηλός αριθμός abort λόγω capacity ορίων, την καθιστούν όμως αποθαρρυντική ως γενική λύση σε εμπορικά συστήματα τα οποία έχουν μεγάλη ποικιλία δεδομένων και workloads. Αυτός είναι και ο βασικός λόγος που στραφήκαμε σε πιο εκλεπτυσμένες τεχνικές HTM προγραμματισμού που αντιμετωπίζουν τέτοιου είδους προβλήματα.

5.3 HTM και NUMA

5.3.1 Πως επηρεάζει ένα σύστημα NUMA το HTM ?

Όπως αναφέραμε στην αρχή του κεφαλαίου, όταν ένα νήμα που ανήκει σε ένα NUMA κόμβο Α χρειάζεται μια πληροφορία που βρίσκεται στην L3 cache ενός άλλου NUMA κόμβου, θα δαπανήσει πολύ περισσότερο χρόνο για να την αποκτήσει από 'τι αν βρισκότανε στην L3 cache του ίδιου NUMA κόμβου. Για παράδειγμα στο αρχείο /sys/devices/system/node/node0/distance του συστήματος που τρέχουμε τα πειράματά μας παρατηρούμε τις τιμές 10 και 21 που είναι ενδεικτικές για τον χρόνο που χρειάζεται ένα νήμα για την πρόσβαση σε τοπική μνήμη και σε μνήμη άλλου κόμβου αντίστοιχα. Η επίδραση της ιδιότητας αυτής στην επίδοση παράλληλων προγραμμάτων μεγεθύνεται εάν τα παράλληλα προγράμματα είναι υλοποιημένα με HTM. Ο λόγος για τον οποίο συμβαίνει αυτό είναι γιατί οι καταργήσεις αντιγράφων σε caches γειτονικών sockets (cross-socket cache invalidations) οδηγούν σε μεγαλύτερο χρόνο ανάκτησης πληροφοριών που αντίστοιχα οδηγούν σε αύξηση του χρόνου ολοκλήρωσης των transactions και άρα μεγαλύτερο χρονικό περιθώριο για την παρουσία conflicts. Για να το καταλάβουμε καλύτερα ας δούμε το παρακάτω παράδειγμα:

- Έστω ότι έχουμε τρία νήματα Α,Β,Γ τα οποία ανήκουν στον NUMA κόμβο 1.
- Αρχικά το δεδομένο Χ βρίσκεται στις L1 και L3 caches του Α και η L3 μοιράζεται με τα υπόλοιπα νήματα του NUMA 1.
- Έστω επίσης ότι το διάβασμα από την τοπική L3 απαιτεί 10 μονάδες χρόνου ενώ από απομακρυσμένη 20. Το γράψιμο θεωρούμε ότι απαιτεί 15 μονάδες χρόνου και οι λειτουργίες transaction (start, commit, abort) από 5 μονάδες η κάθε μια.
- Τέλος, τα νήματα Α και Β βρίσκονται εντός transaction ενώ το Γ εκτός.

Χρόνος	Α	Β	Γ	Σχόλια
0-10		read X		ίδια L3 με Α → 10 χρ. μονάδες
10-25		write X		ο Β γράφει στο Χ, Χ στο read set του Α
25-30	abort			→ data conflict → ο Α δέχεται abort
30-35	start	commit		
70-80	read X			ίδια L3 με Β → 10 χρ. μονάδες
80-95	write X			
100-105	commit			
120-130			read X	

Ας δούμε τώρα ακριβώς το ίδιο παράδειγμα όπου τα νήματα Α και Γ ανήκουν στον NUMA κόμβο 1 και το Β στον κόμβο 2.

Χρόνος	Α	Β	Γ	Σχόλια
0-20		read X		διάβασμα από L3 του Α → 20 χρ. μονάδες
20-35		write X		
35-40	abort			
40-45	start	commit		πλέον το δεδομένο Χ βρίσκεται στην L3 του NUMA 2
80-100	read X			διάβασμα από L3 του Β → 20 χρ. μονάδες
100-115	write X			
120-140			read X	διάβασμα από L3 του Β → 20 χρ. μονάδες Αφού ο Γ διαβάσει το Χ θα έχουμε έγκυρα αντίγραφα και στην L3 του Β και στην L3 του Γ ο Γ διαβάζει το Χ, το Χ βρίσκεται στο write set του Α
120-125	abort			→ data conflict → ο Α δέχεται abort
125-130	start			
165-175	read X			διάβασμα από την μοιραζόμενη L3 των Α και Γ → 10 χρ. μονάδες
175-190	write X			
195-200	commit			

Όπως παρατηρούμε στο τελευταίο παράδειγμα η εντολή write X του Β οδήγησε σε cross-socket cache invalidation του Χ για τον Α με αποτέλεσμα την επόμενη φορά που ο Α θα διαβάζει το Χ να πρέπει να πάρει την πληροφορία από την L3 του Β (αφού ο Β έχει κάνει commit) που βρίσκεται σε άλλο NUMA κόμβο. Αυτός ο επιπλέον χρόνος που χρειάζεται ο Α για να ανακτήσει το Χ είναι αρκετός ώστε να προλάβει ο Γ, διαβάζοντας το Χ, να δημιουργήσει data conflict και άρα να οδηγήσει σε abort τον Α. Συνεπώς παρατηρούμε ότι στη περίπτωση του HTM η αύξηση του χρονικού παραθύρου των transactions οδηγεί στην παρουσία περεταίρω conflict aborts.

Πόσο όμως επηρεάζει ένα σύστημα NUMA το HTM στην πράξη? Για να το διαπιστώσουμε εκτελέσαμε το παρακάτω πείραμα:

- Τρέχουμε το πείραμα 1 που είδαμε στην ενότητα 5.1.3 εκτελώντας την fine-grained HTM υλοποίηση για ρυθμό ενημερώσεων 50% και αρχικό μέγεθος skip list 10 χιλ.
- Συγκρίνουμε τα αποτελέσματά μας για δύο περιπτώσεις που τρέχουμε με 14 threads, όπου στη μια περίπτωση έχουμε και τα 14 νήματα να ανήκουν στο ίδιο NUMA και στη δεύτερη περίπτωση έχουμε 7 νήματα να ανήκουν στο NUMA 0 και 7 στο NUMA 1.
- Για να δούμε κατά πόσο ο ισχυρισμός μας ότι η αύξηση του χρόνου ολοκλήρωσης των transactions οδηγεί εν τέλη στην αύξηση των conflict aborts είναι σωστός, προσθέσαμε στις συγκρίσεις μας δυο περιπτώσεις όπου τρέχουμε με 14 threads στο ίδιο NUMA στις οποίες έχουμε προσθέσει τεχνητή καθυστέρηση 50 και 300 επαναλήψεων πριν το commit των transactions ούτος ώστε να μιμηθούμε τη συμπεριφορά του NUMA συστήματος. Τα αποτελέσματα είναι τα εξής:

threads (NUMA 0 - NUMA 1)	14-0	7-7	14-0 τεχνητή καθυστέρηση 50 επαναλήψεων	14-0 τεχνητή καθυστέρηση 300 επαναλήψεων
throughput	49.2M	36.5M	41.9M	18.5M
#commits	24.6M	18.2M	20.9M	9.1M
#aborts	0.6M	8.5M	2.8M	5.4M
aborts(%) $(\frac{\#aborts}{\#aborts+\#commits} \times 100\%)$	2.6 %	31.8 %	11.6 %	37.2 %
#conflict aborts	0.3M	8.3M	2.5M	5.3M
conflict aborts(%) $(\frac{\#conflict\ aborts}{\#aborts} \times 100\%)$	57 %	98 %	91.5 %	97 %

Πίνακας 5.3 *Αποτελέσματα πειράματος όπου στη δεύτερη στήλη το NUMA είναι απενεργοποιημένο και στην τρίτη είναι ενεργοποιημένο. Στη τέταρτη και στην πέμπτη στήλη το NUMA είναι απενεργοποιημένο αλλά έχουμε προσθέσει τεχνητή καθυστέρηση πριν κάνουν commit τα transactions.*

Παρατηρώντας τον ανωτέρω πίνακα με τα αποτελέσματα επιβεβαιώνουμε ότι το NUMA οδηγεί σε μεγάλη αύξηση των conflict aborts. Αντίστοιχα οι δύο τελευταίες στήλες μας δείχνουν ότι με τη προσθήκη μικρής καθυστέρησης τα ποσοστά των conflict aborts και κατ' επέκταση των aborts συνολικά αυξάνονται σημαντικά, μιμούμενα τα αποτελέσματα που λαμβάνουμε εκτελώντας σε δυο sockets.

5.3.2 Αντιμετωπίζοντας το NUMA

Έχουν προταθεί διάφορες τεχνικές για την αντιμετώπιση των παρενεργειών του NUMA που προαναφέραμε. Για παράδειγμα, αρκετές δημοσιεύσεις [14], [15], [16] παρατηρούν ότι η χρησιμοποίηση λιγότερων νημάτων από το διαθέσιμο αριθμό πυρήνων μπορεί να οδηγήσει σε καλύτερες επιδόσεις.

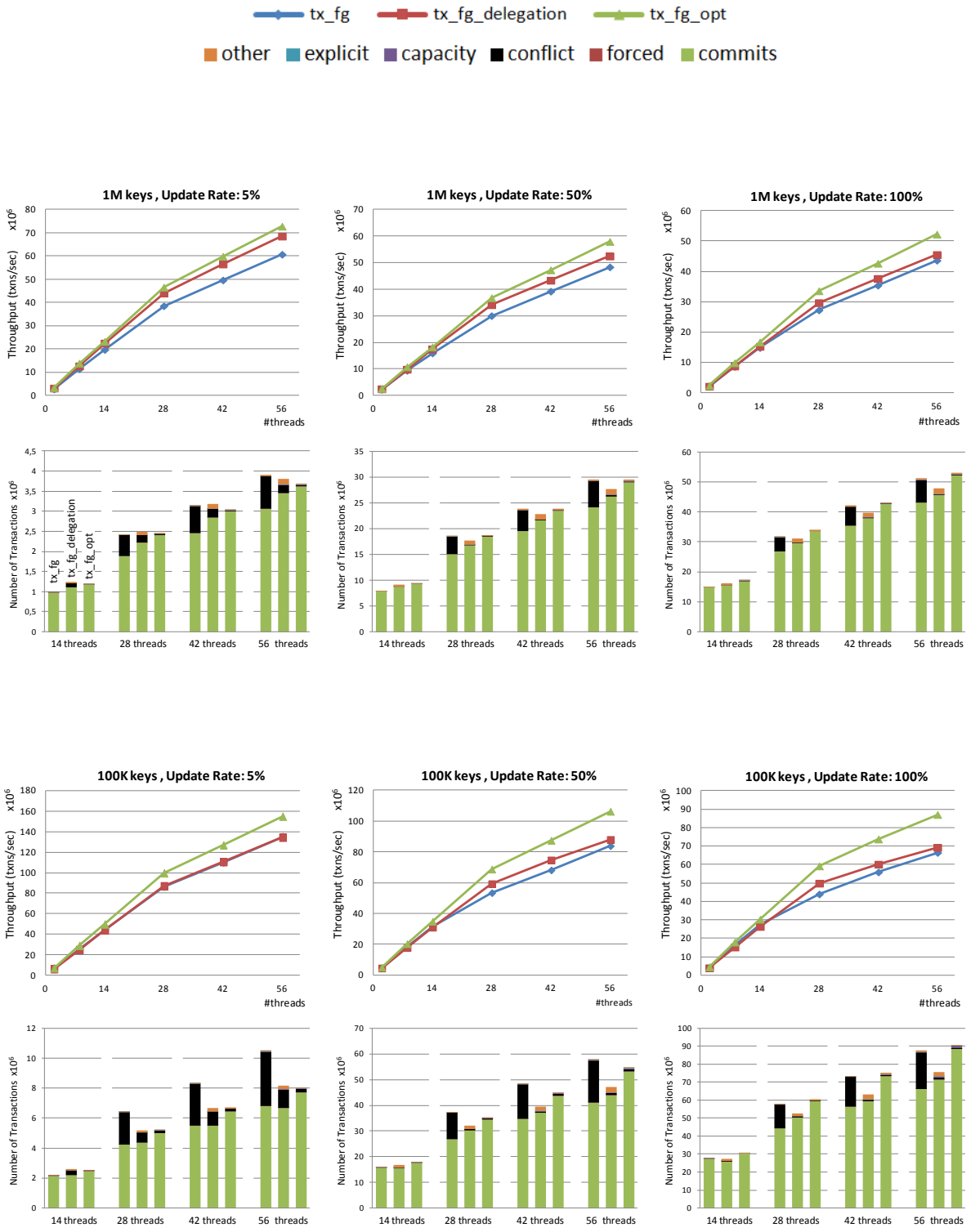
Μια άλλη προσέγγιση για συστήματα με δυο sockets είναι να υποχρεώνουμε τα νήματα στο δεύτερο socket να αναστέλλουν την λειτουργία τους για ένα χρονικό διάστημα πριν ξανά εκκινήσουν την εκτέλεση ενός transaction που είχε δεχθεί abort. Στη δημοσίευση των Brown κ.ά. [17] προτείνεται η τεχνική *NATLE* (*for NUMA-aware Transactional Lock Elision*) σύμφωνα με την οποία για κάθε critical section θεσπίζουμε μια *αναγνωριστική περίοδο* (*profiling phase*) και μια *μετά-αναγνωριστική περίοδο* (*post-profiling phase*). Στην αναγνωριστική περίοδο έχουμε αρχικά ένα διάστημα στο οποίο εκτελούνται νήματα μόνο από το πρώτο socket, ακολουθούμενο από ένα διάστημα στο οποίο εκτελούνται νήματα μόνο από το δεύτερο socket, ακολουθούμενο από ένα διάστημα που εκτελούνται νήματα και από τα δύο socket. Στην μετά-αναγνωριστική περίοδο αξιολογούμε τις επιδόσεις που πετύχαμε στα διάφορα διαστήματα της αναγνωριστικής περιόδου και κατανέμουμε τον χρόνο στα διάφορα socket όσο το δυνατών βέλτιστα σύμφωνα με τα αποτελέσματα της αξιολόγησης. Κατά τη διάρκεια εκτέλεσης ενός προγράμματος οι δυο περίοδοι εναλλάσσονται συνεχώς.

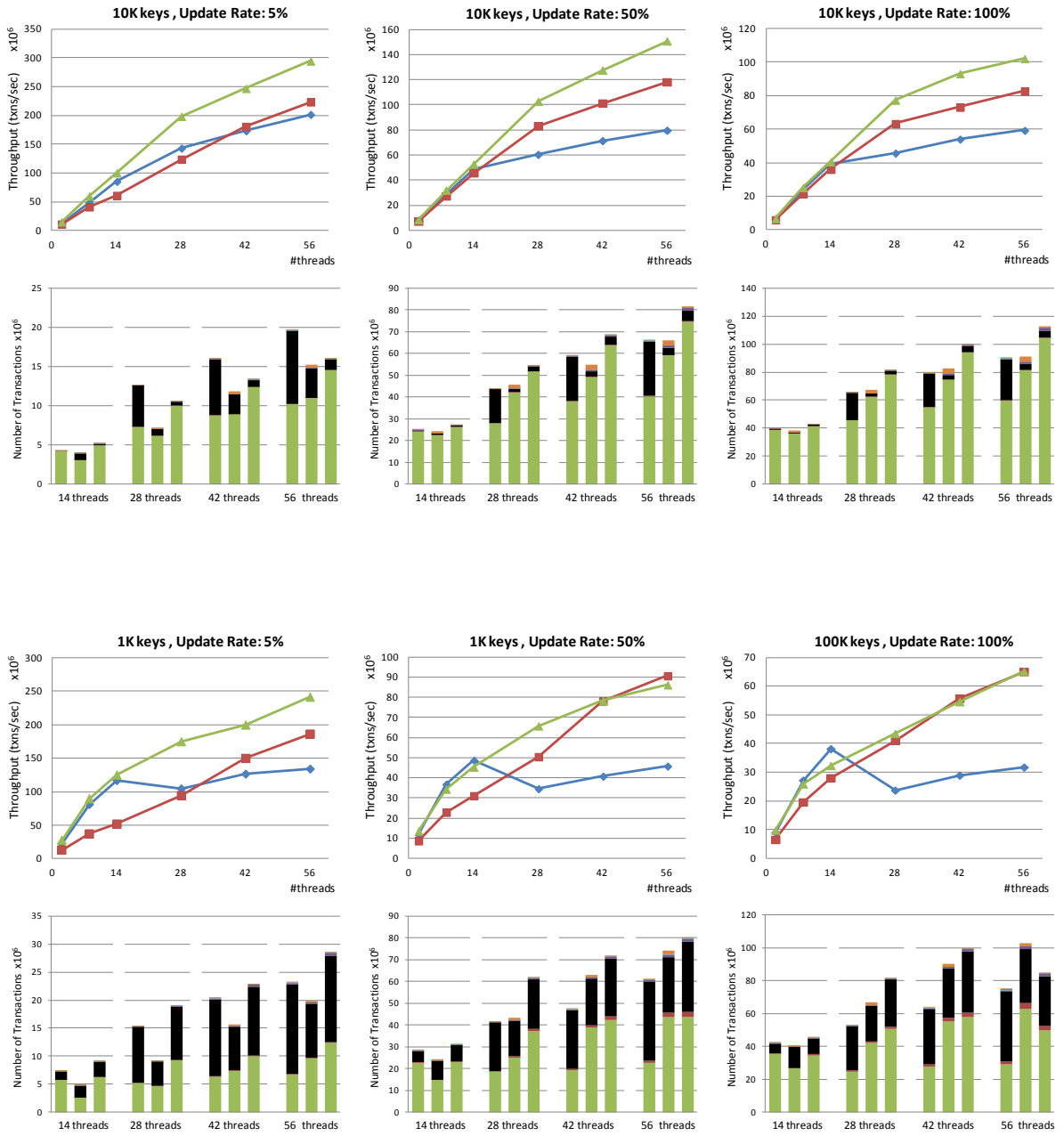
Μια άλλη τεχνική που στοχεύει στη μείωση των απομακρυσμένων cache misses είναι η λεγόμενη *Τεχνική Εξουσιοδότησης* (*Delegation*). Η βασική ιδέα της τεχνικής αυτής είναι η επικοινωνία μέσω κοινής μνήμης μεταξύ client και server threads, όπου οι clients αναθέτουν στους servers την πρόσβαση και εκτέλεση λειτουργιών σε συγκεκριμένες δομές δεδομένων ή κρίσιμα τμήματα. Στη βιβλιογραφία [18], [19], [20] συναντάμε διάφορους τρόπους υλοποίησης της τεχνικής αυτής.

Στο πλαίσιο της διπλωματικής αναπτύξαμε την εξής παραλλαγή της τεχνικής εξουσιοδότησης: Αντί για μια ενιαία δομή δεδομένων δημιουργούμε δύο όπου η πρώτη θα περιέχει στοιχεία στο πρώτο μισό του εύρους κλειδιών ενώ η δεύτερη θα περιέχει στοιχεία στο δεύτερο μισό του εύρους κλειδιών. Τα threads του πρώτου socket αναλαμβάνουν να εκτελέσουν λειτουργίες επί της πρώτης δομής ενώ τα threads του δεύτερου socket αναλαμβάνουν να εκτελέσουν λειτουργίες επί της δεύτερης δομής. Αν ένα thread κληθεί να εκτελέσει κάποια λειτουργία για στοιχείο με κλειδί που δεν ανήκει στο εύρος κλειδιών της δομής που αναλαμβάνει τότε στέλνει μήνυμα σε ένα thread του άλλου socket και το εξουσιοδοτεί να εκτελέσει αυτό τη λειτουργία. Ο μηχανισμός μηνυμάτων που χρησιμοποιούμε είναι ο εξής: Κάθε νήμα του πρώτου socket επικοινωνεί με ακριβώς ένα νήμα του δεύτερου socket και ο καθορισμός των ζευγών γίνεται στην αρχή της εκτέλεσης του προγράμματος. Η επικοινωνία γίνεται μέσω ουρών. Κάθε thread έχει τρεις ουρές, μια με στοιχεία που προορίζονται για αναζητήσεις, μια με στοιχεία για εισαγωγές και μια με στοιχεία για διαγραφές. Όταν ένα νήμα θέλει να εξουσιοδοτήσει ένα άλλο νήμα για μια λειτουργία τότε εισάγει στην αντίστοιχη ουρά του άλλου νήματος το εν λόγω στοιχείο. Κάθε νήμα πριν ξεκινήσει την εκτέλεση μιας λειτουργίας κοιτάει στην αντίστοιχη ουρά αν υπάρχει κάποιο στοιχείο και ανάλογα ξεκινάει μια λειτουργία με καινούριο κλειδί ή με το κλειδί του στοιχείου της ουράς. Επειδή σε κάθε ουρά έχουμε μόνο έναν που εισάγει και μόνο έναν που διαγράφει δεν απαιτείται κάποια μέθοδος συγχρονισμού.

Εκτελέσαμε πειραματικά την παραπάνω υλοποίηση. Για να εξετάσουμε το κόστος του μηχανισμού μηνυμάτων αλλά και για να έχουμε μια εικόνα της βέλτιστης επίδοσης της HTM skip list σε περιβάλλον NUMA υλοποιήσαμε και μια εκδοχή της παραπάνω υλοποίησης χωρίς εξουσιοδότηση. Με άλλα λόγια τα νήματα του πρώτου socket θα δραστηριοποιούνται μόνο στη πρώτη δομή και τα νήματα του δεύτερου socket μόνο στη δεύτερη δομή χωρίς να υπάρχει επικοινωνία μεταξύ τους.

Τα αποτελέσματα των μετρήσεων φαίνονται στο σχήμα 5.8 και ακολουθούν την λογική του πειράματος 1 που είδαμε στην ενότητα 5.1.3. Εκτελέσαμε και μετρήσεις με βάση το πείραμα 2 αλλά τα αποτελέσματα ήταν παρόμοια οπότε δεν τα συμπεριλάβαμε.


tx_fg
tx_fg_delegation
tx_fg_opt
other
explicit
capacity
conflict
forced
commits
1M keys , Update Rate: 5%
1M keys , Update Rate: 50%
1M keys , Update Rate: 100%
100K keys , Update Rate: 5%
100K keys , Update Rate: 50%
100K keys , Update Rate: 100%
Throughput (txns/sec) x10^6
#threads
Number of Transactions x10^6
14 threads
28 threads
42 threads
56 threads



Σχήμα 5.8 *Εκτέλεση του πειράματος 1 και παρουσίαση αποτελεσμάτων για απόδοση (throughput) και μεγεθών commit και abort των transactions.Τα aborts διαχωρίζονται σε conflict,forced,capacity,explicit και other. Τα forced aborts η αλλιώς invalidation aborts μετρούνται διαφορετικά από τα υπόλοιπα explicit aborts και προκύπτουν όταν αναγκαζόμαστε να απορρίψουμε ένα transaction επειδή βρήκαμε πρόβλημα κατά τη διάρκεια του έλεγχου εγκυρότητας όπως για παράδειγμα όταν preds[h].next[h] != succs[h]. Οι υλοποιήσεις που εξετάζουμε είναι: 1) tx_fg: η γνωστή υλοποίηση της ενότητας 4.3 (μπλέ χρώμα/πρώτη στήλη), 2) tx_fg_delegation: η παραλλαγή της tx_fg υλοποίησης που χρησιμοποιεί delegation (κόκκινο χρώμα/δεύτερη στήλη), 3) tx_fg_opt: η θεωρητικά βέλτιστη tx_fg υλοποίηση όπου κάθε socket δραστηριοποιείται σε άλλη δομή και δεν υπάρχει επικοινωνία μεταξύ τους (πράσινο χρώμα/τρίτη στήλη).*

Παρατηρώντας τις γραφικές του σχήματος 5.8 συμπεραίνουμε ότι μια τεχνική εξουσιοδότησης μπορεί να βελτιώσει την απόδοση HTM Skip List υλοποιήσεων σε συστήματα με NUMA κυρίως για μεσαίο και μεγάλο αριθμό νημάτων (>28 threads). Για μικρότερο αριθμό νημάτων πολλές φορές παρατηρούμε ότι το κόστος του συστήματος επικοινωνίας είναι μεγαλύτερο σε σχέση με τα οφέλη που προκύπτουν από τον περιορισμό των NUMA επιπτώσεων. Σε σχέση με τα στατιστικά των transactions των διάφορων υλοποιήσεων, βλέπουμε ότι η τεχνική εξουσιοδότησης περιορίζει αρκετά τον αριθμό των conflict aborts και κατ' επέκταση των aborts συνολικά. Τέλος στην πλειοψηφία των μετρήσεων η tx_fg_opt υλοποίηση είναι σταθερά καλύτερη σε απόδοση συγκριτικά με τις υπόλοιπες, γεγονός που μας παροτρύνει να βρούμε αποδοτικότερες μεθόδους αντιμετώπισης των NUMA επιπτώσεων σε μελλοντικές εργασίες.

5.4 Τι συμβαίνει όταν εκτελούνται παραπάνω νήματα από τον συνολικό αριθμό των πυρήνων του συστήματος ?

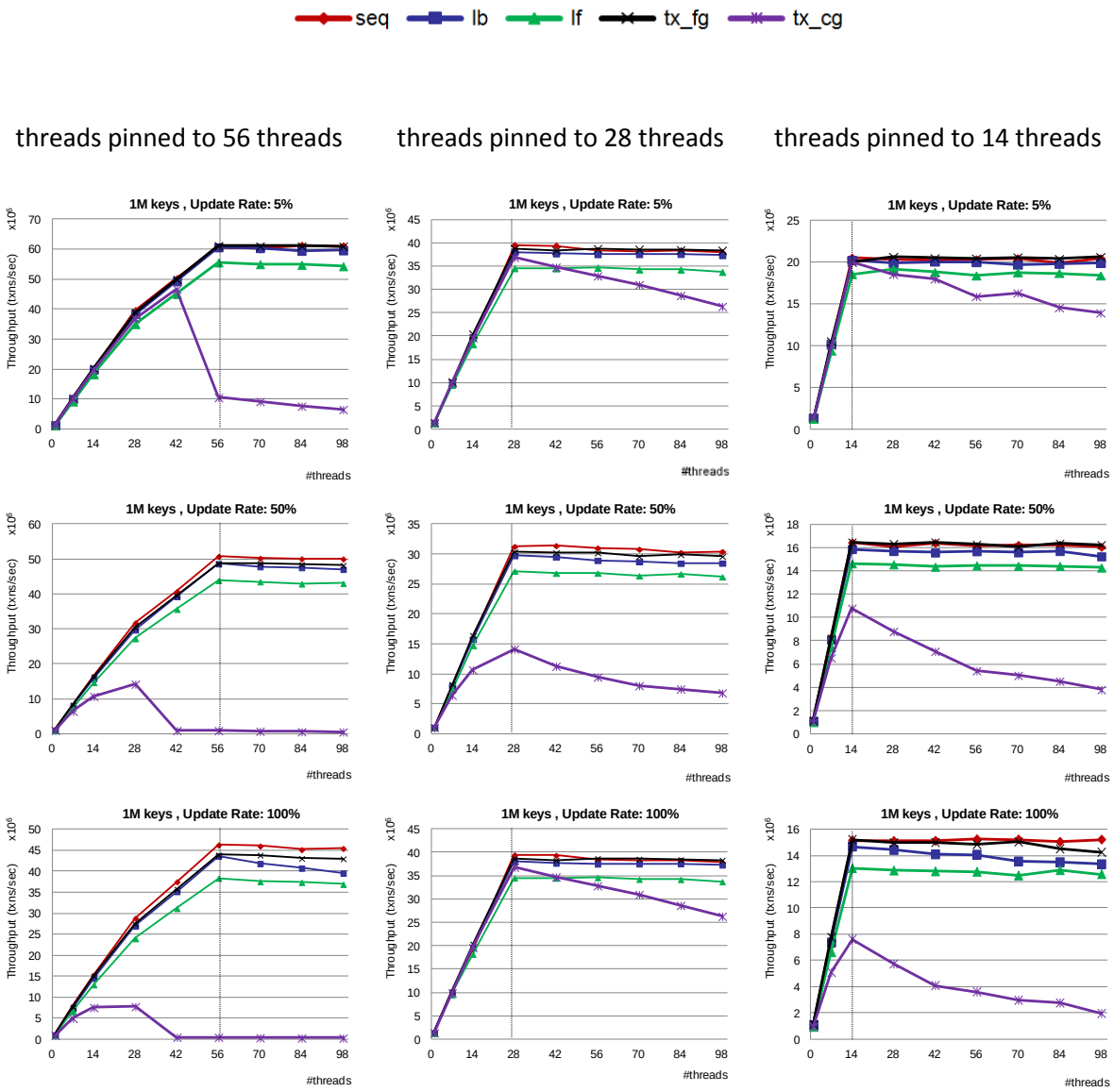
Στην εισαγωγή είδαμε ότι ένα από τα προβλήματα των lock-based υλοποιήσεων είναι η δημιουργία κονβόι λόγω κλειδώματος (lock convoying). Πιο αναλυτικά, όταν ένα νήμα Α κρατάει ένα κλείδωμα και ανασταλεί από το λειτουργικό σύστημα λόγω ενός context switch, τότε όλα τα νήματα που περιμένουν από το Α να ελευθερώσει το κλείδωμα θα παραμείνουν στάσιμα μέχρις ότου το λειτουργικό σύστημα επαναφέρει το Α και εν τέλει ελευθερωθεί το κλείδωμα. Το φαινόμενο γίνεται ιδιαίτερα έντονο όταν ο αριθμός των νημάτων που τρέχουνε είναι μεγαλύτερος από τον αριθμό των πυρήνων του συστήματος. Σε αυτή την περίπτωση τα νήματα που χρησιμοποιούν τον ίδιο πυρήνα μοιράζονται τον χρόνο λειτουργίας του πυρήνα και ως εκ τούτου παραμένουν για μεγάλο χρονικό διάστημα ανενεργά.

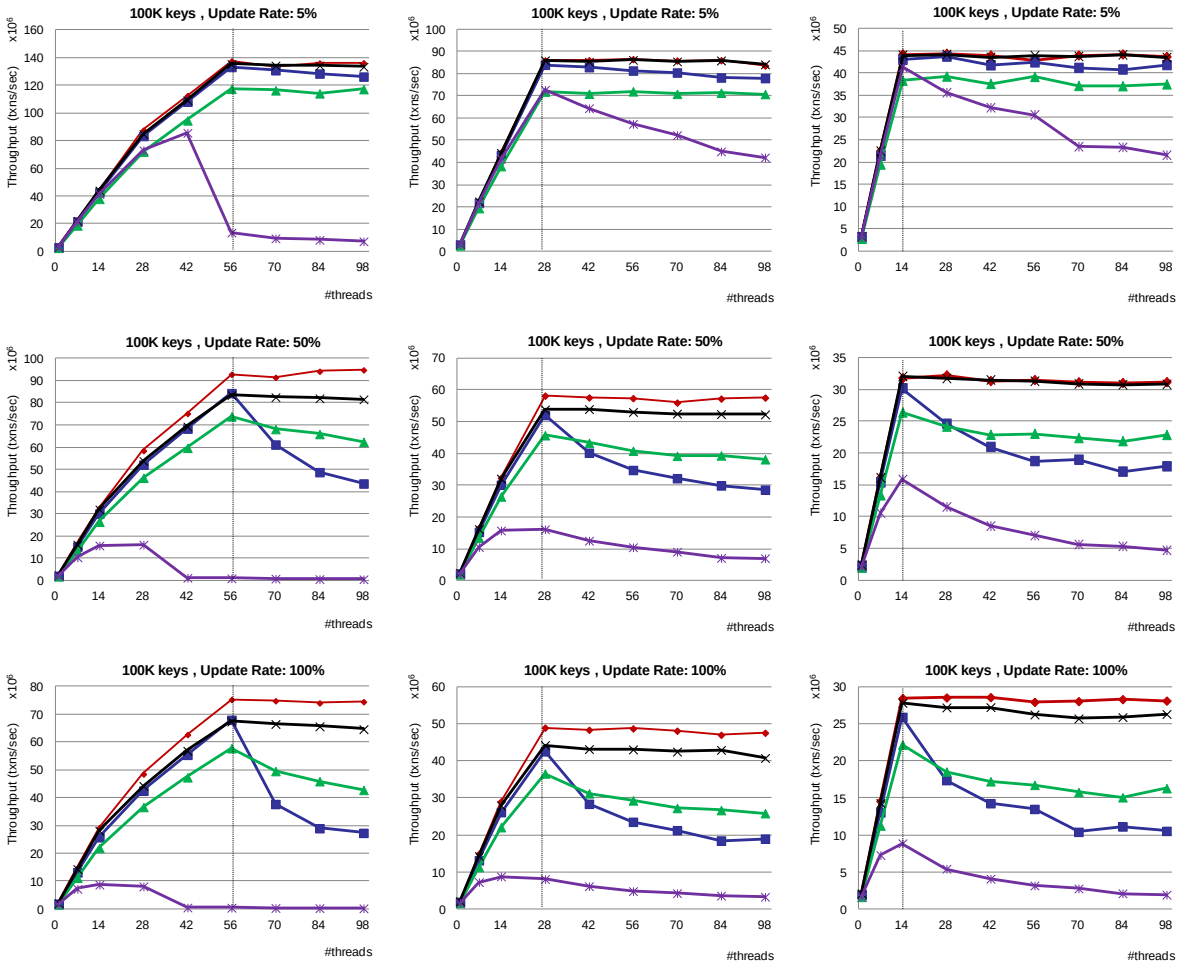
Το πρόβλημα αυτό δεν εμφανίζεται στις lock-free υλοποιήσεις καθώς όταν ένα νήμα ανασταλεί δεν κρατάει κάποιο κλείδωμα ούτε έχει δημιουργήσει κάποια εξάρτηση με κάποιο άλλο νήμα που θα το υποχρεώσει να περιμένει.

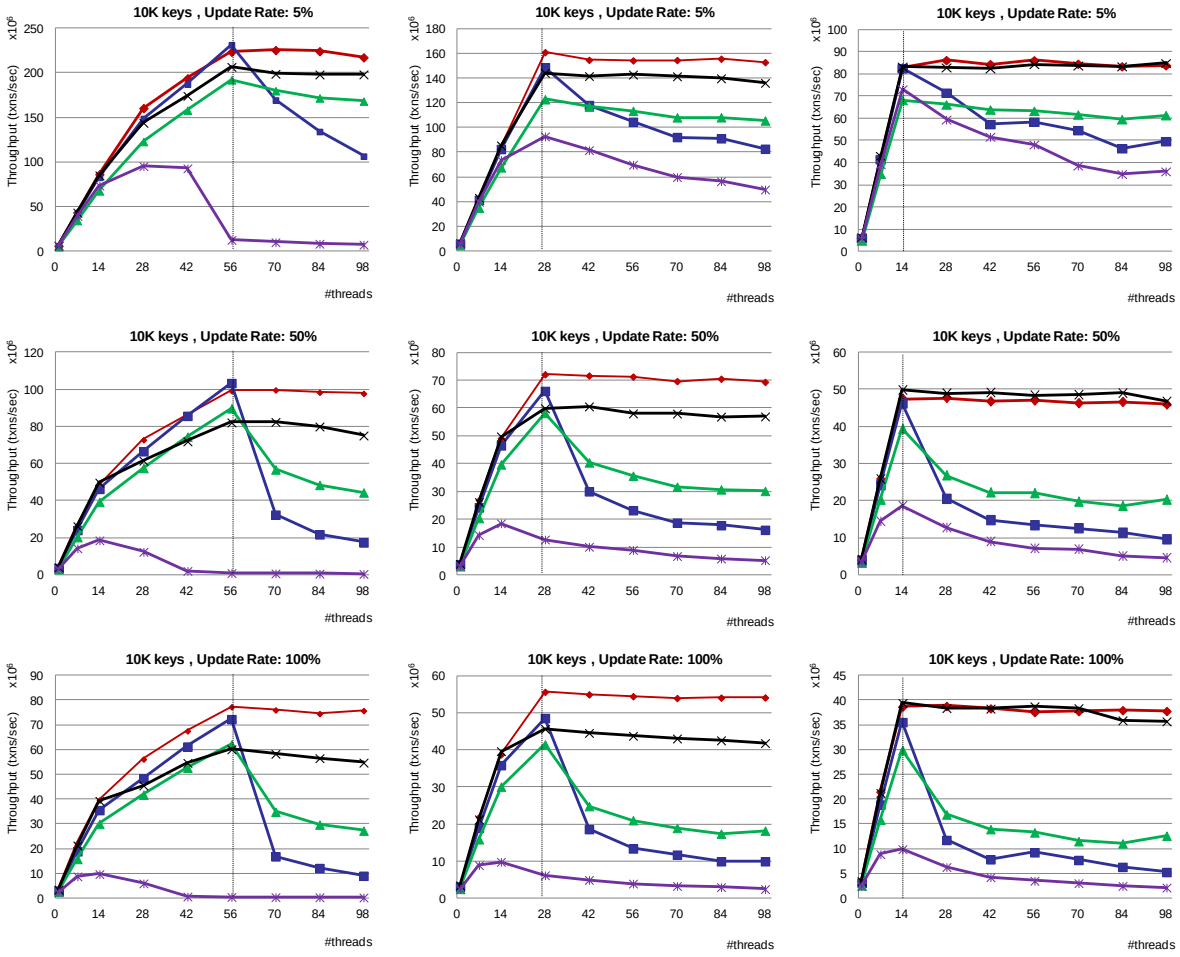
Παρόμοια, το πρόβλημα του lock convoying δεν εμφανίζεται ούτε στις HTM υλοποιήσεις καθώς ούτε και εδώ υπάρχουν εξαρτήσεις ανάμεσα στα threads. Το πρόβλημα που δημιουργείται από την άλλη όμως είναι ότι τα context switches οδηγούν σε aborts τα transactions που διακόπτονται. Το πρόβλημα αυτό, όπως θα δούμε στη συνέχεια, είναι αρκετά μικρότερο σε σχέση με το lock convoying, ειδικά όταν τα transactions είναι μικρά.

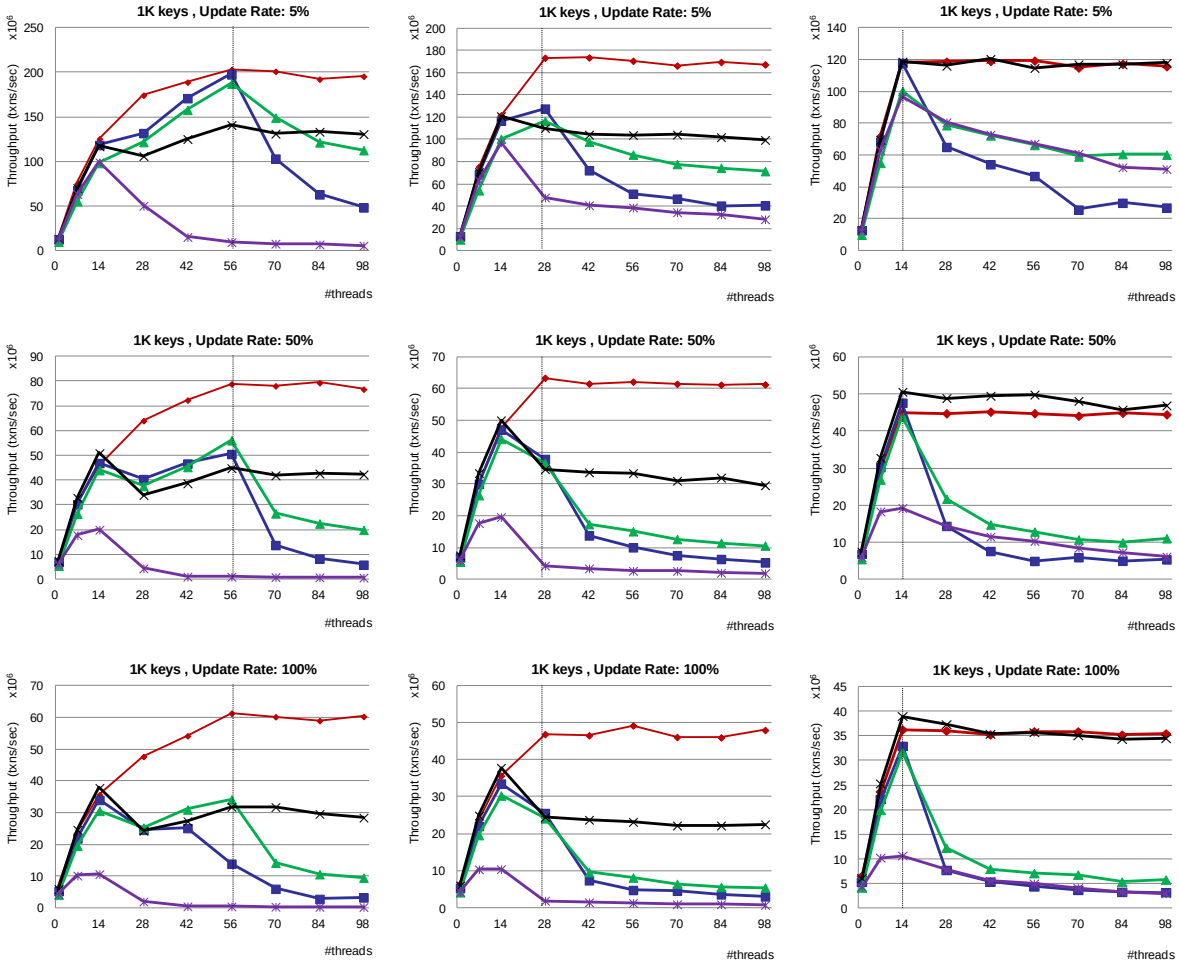
Στο σχήμα 5.9 φαίνονται τα αποτελέσματα του πειράματος που εκτελέσαμε με σκοπό να δούμε τις συμπεριφορές των υλοποιήσεων όταν ο συνολικός αριθμός των νημάτων που τρέχουνε είναι μεγαλύτερος από τον αριθμό των πυρήνων του συστήματος.
Οι υλοποιήσεις που εξετάζουμε είναι οι ίδιες που είδαμε στα πειράματα 1 και 2 της ενότητας 5.1. Εκτελέσαμε μετρήσεις όπου τρέχουν 1,14,28,42,56,70,84 ,98 νήματα τα οποία προσδένονται (γίνονται pinned) σε 56 (πρώτη στήλη), 28 (δεύτερη στήλη) ή 14 πυρήνες (τρίτη στήλη).

Πείραμα 1 : Alternate

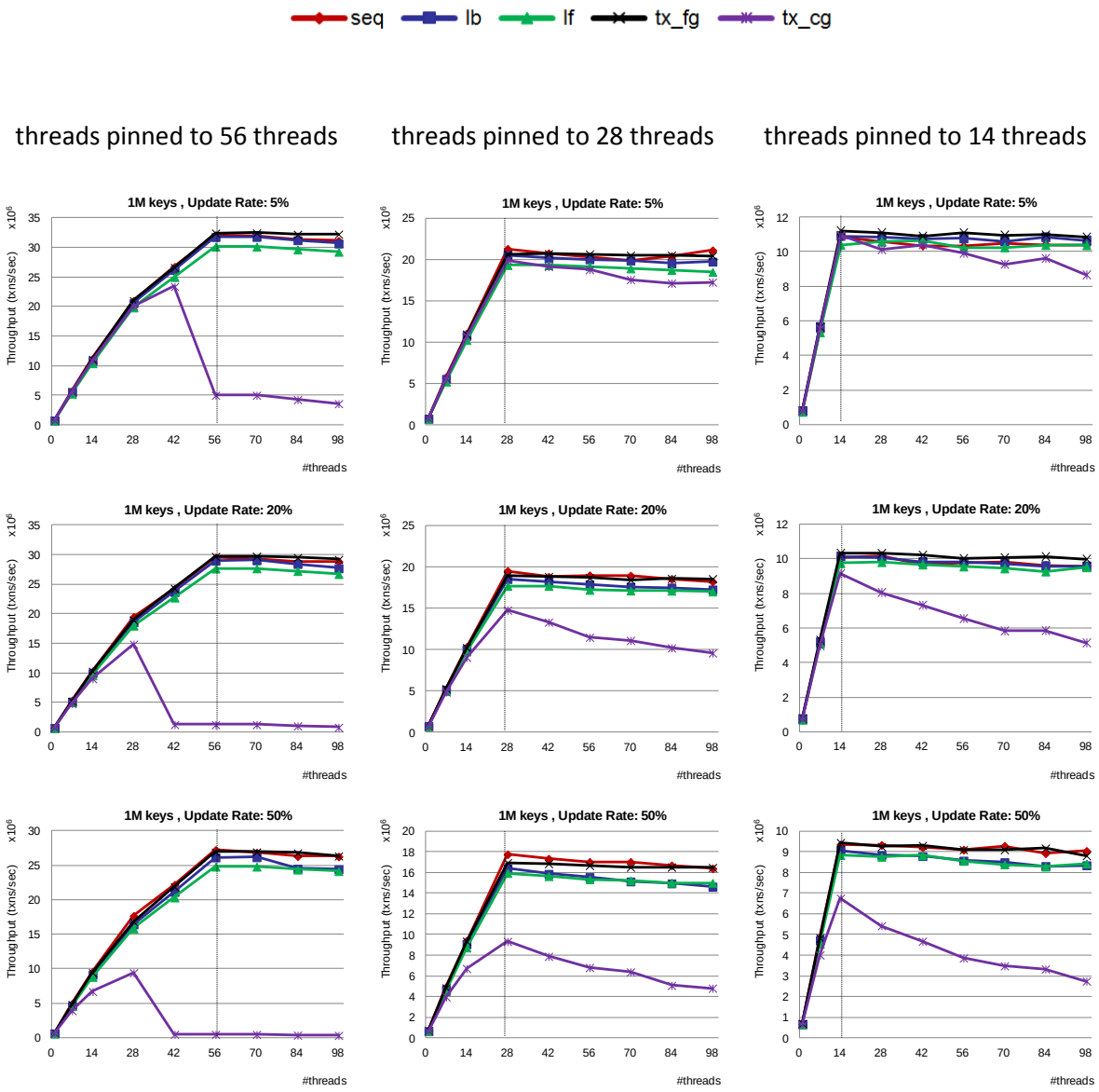


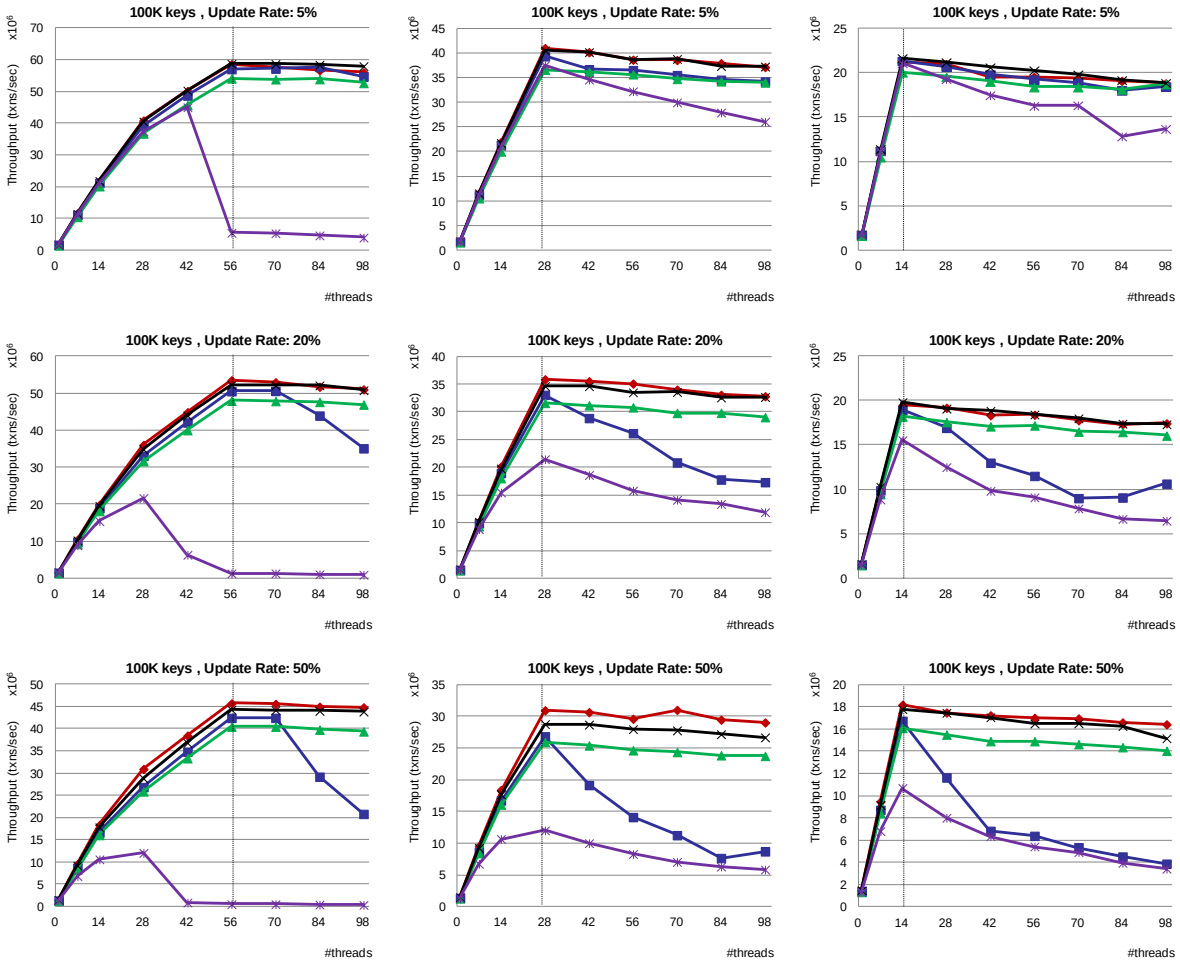

100K keys , Update Rate: 5%
100K keys , Update Rate: 5%
100K keys , Update Rate: 5%
100K keys , Update Rate: 50%
100K keys , Update Rate: 50%
100K keys , Update Rate: 50%
100K keys , Update Rate: 100%
100K keys , Update Rate: 100%
100K keys , Update Rate: 100%
Throughput (txns/sec)
x10^6
#threads

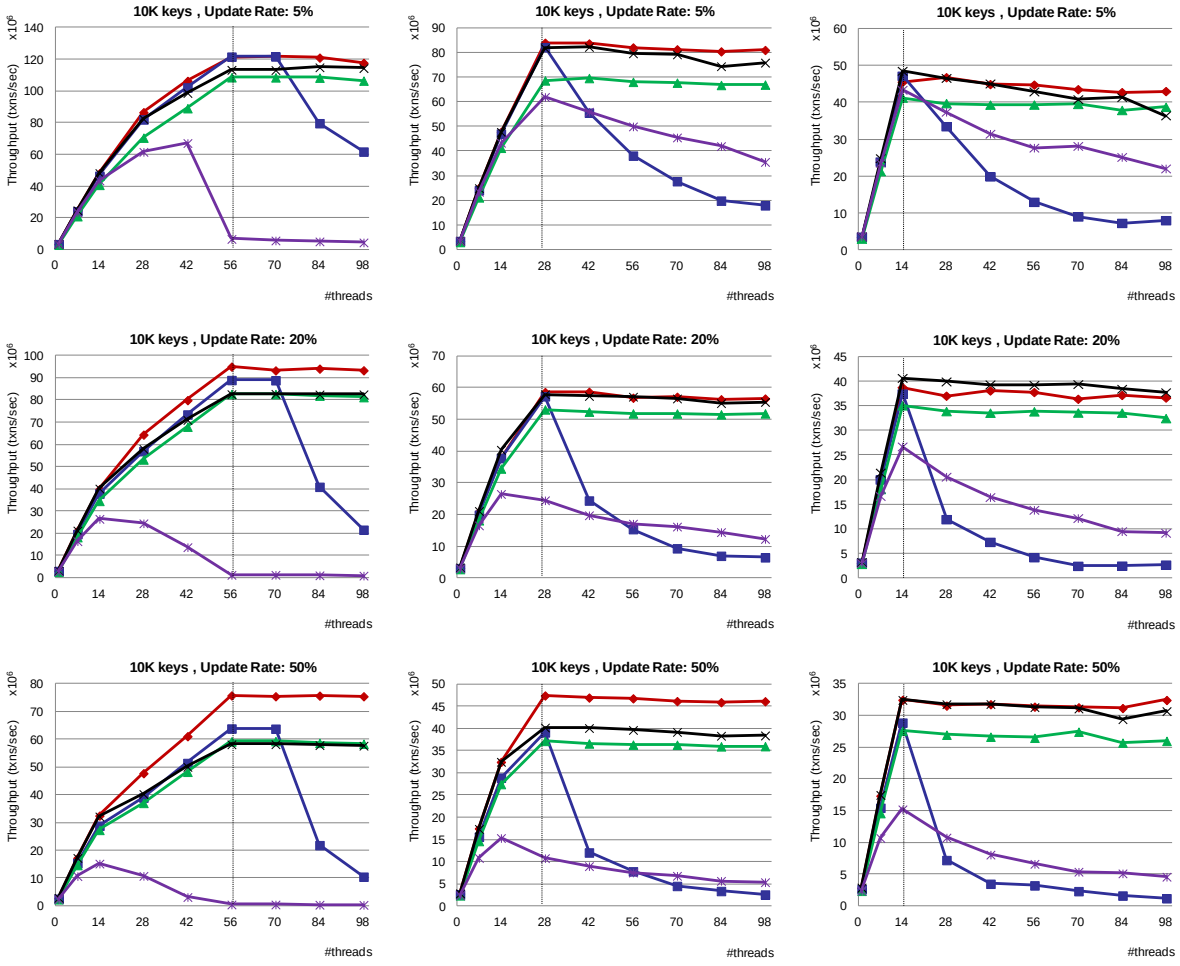

10K keys , Update Rate: 5%
10K keys , Update Rate: 5%
10K keys , Update Rate: 5%
10K keys , Update Rate: 50%
10K keys , Update Rate: 50%
10K keys , Update Rate: 50%
10K keys , Update Rate: 100%
10K keys , Update Rate: 100%
10K keys , Update Rate: 100%
Throughput (txns/sec)
x10^6
#threads

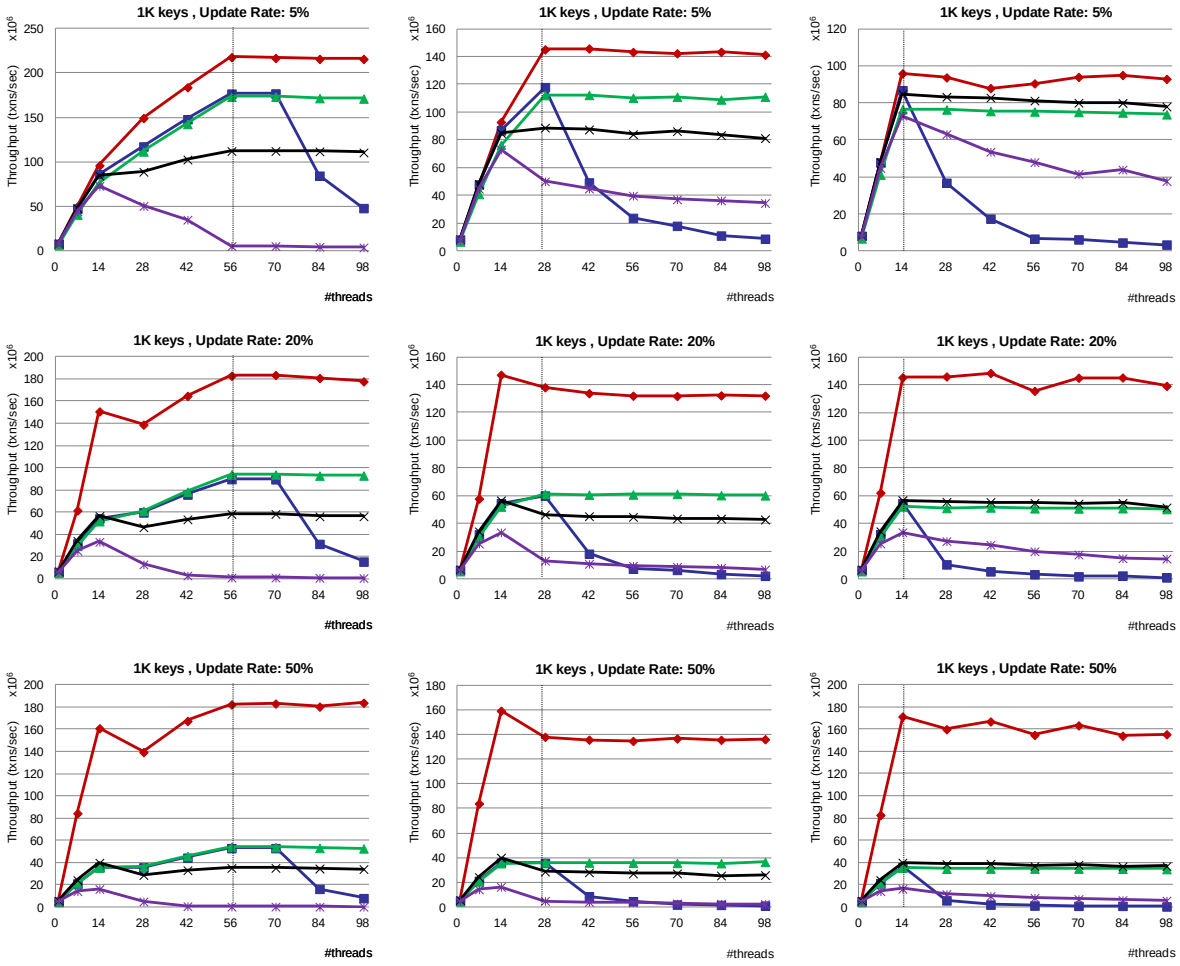

1K keys , Update Rate: 5%
1K keys , Update Rate: 5%
1K keys , Update Rate: 5%
1K keys , Update Rate: 50%
1K keys , Update Rate: 50%
1K keys , Update Rate: 50%
1K keys , Update Rate: 100%
1K keys , Update Rate: 100%
1K keys , Update Rate: 100%
Throughput (txns/sec) x10^6
#threads

Πείραμα 2 : Random




100K keys , Update Rate: 5%
100K keys , Update Rate: 5%
100K keys , Update Rate: 5%
100K keys , Update Rate: 20%
100K keys , Update Rate: 20%
100K keys , Update Rate: 20%
100K keys , Update Rate: 50%
100K keys , Update Rate: 50%
100K keys , Update Rate: 50%
Throughput (txns/sec) x10^6
#threads


10K keys , Update Rate: 5%
10K keys , Update Rate: 5%
10K keys , Update Rate: 5%
10K keys , Update Rate: 20%
10K keys , Update Rate: 20%
10K keys , Update Rate: 20%
10K keys , Update Rate: 50%
10K keys , Update Rate: 50%
10K keys , Update Rate: 50%
Throughput (txns/sec) x10^6
#threads



Σχήμα 5.9 *Επίδοση παράλληλων Skip List υλοποιήσεων όταν ο συνολικός αριθμός των νημάτων που τρέχουνε είναι μεγαλύτερος από τον αριθμό των πυρήνων του συστήματος. Τα νήματα προσδένονται σε 56 (πρώτη στήλη),28 (δεύτερη στήλη) ή 14 πυρήνες (τρίτη στήλη) αντίστοιχα.*

Παρατηρώντας τις παρακάτω γραφικές βλέπουμε την επίδραση του lock convoying στην lock-based υλοποίηση. Είναι εμφανές ότι από το σημείο που ξεκινάει το pinning και μετά η επίδοση της lock-based skip list πέφτει διαρκώς. Το γεγονός ότι οι γραφικές της HTM υλοποίησης παραμένουν σταθερές και ανεξάρτητες από τον αριθμό των νημάτων που γίνονται pinned αποτελεί το μεγάλο της πλεονέκτημα και την καθιστά αξιόλογη εναλλακτική έναντι της lock-based Skip List. Ενώ στη γενική περίπτωση όπου σε κάθε πυρήνα είναι προσδεμένο ένα μόνο νήμα η lock-based υλοποίηση υπερέχει ελαφρώς έναντι της HTM, παρατηρούμε ότι όταν εκτελούνται παραπάνω νήματα από τον συνολικό αριθμό των πυρήνων η απόδοση της HTM υλοποίησης μπορεί να είναι έως και 250% καλύτερη σε σχέση με την απόδοση της lock-based Skip List.

5.5 Γενικά συμπεράσματα

Στο κεφάλαιο αυτό μπορέσαμε μέσα από πειράματα να κατανοήσουμε καλύτερα τις διάφορες υλοποιήσεις Skip List καθώς και να εντοπίσουμε τα προτερήματα και μειονεκτήματα κάθε μίας. Συνοπτικά τα συμπεράσματά μας είναι τα εξής:

- HTM Based:
  - Coarse-Grained: + πολύ απλή υλοποίηση
    + ικανοποιητική απόδοση σε απλά set με μικρή συμφόρηση και μικρά payloads
    - πολύ μεγάλο footprint που οδηγεί σε ανεπιθύμητα aborts και capacity overflows
  - Fine-Grained: + σχετικά απλή υλοποίηση, ανάλογης δυσκολίας με εκείνης της lock-based υλοποίησης
    + μικρό footprint που πλησιάζει το βέλτιστο για τη διασφάλιση εγκυρότητας → εξασφαλίζει αξιόλογες επιδόσεις
    + σταθερή απόδοση όταν εκτελούνται παραπάνω νήματα από τον συνολικό αριθμό των πυρήνων του συστήματος
    - οι επιπτώσεις του NUMA μεγεθύνονται στο HTM
    - read conflicts προκαλούν αρνητικά (ανεπιθύμητα) aborts
    - Στη περίπτωση που ένα νήμα T1 περιμένει από ένα νήμα T2 να ελευθερώσει ένα κλείδωμα στην lock-based εκδοχή, οι T1 και T2 θα οδηγούσαν το ένα ή και τα δύο νήματα σε abort σε αντίστοιχη περίπτωση στην HTM εκδοχή.

- Lock-Based: + σχετικά απλή υλοποίηση
  + γρήγορες non-blocking και wait-free προσπελάσεις της δομής
  + πειραματικά στις περισσότερες περιπτώσεις, όταν ο αριθμός των νημάτων που εκτελούνται δεν υπερβαίνει τον συνολικό αριθμό των πυρήνων του συστήματος, υπερέχει σε απόδοση έναντι των υπόλοιπων υλοποιήσεων
  - lock convoying. Αντιμετωπίζει μεγάλη πτώση σε απόδοση όταν ο αριθμός των νημάτων που εκτελούνται υπερβαίνει τον συνολικό αριθμό των πυρήνων του συστήματος
  - locks → έλλειψη robustness

- Lock-Free: + robustness
  + non-blocking λειτουργίες, συνεπώς τρέχει χωρίς παύσεις
  + καλή επίδοση ιδιαίτερα σε περιπτώσεις με υψηλή συμφόρηση
  - πολύπλοκη υλοποίηση
  - χρονικό κόστος ατομικών λειτουργιών
  - στις ενημερώσεις ξοδεύει χρόνο για cleaning σκοπούς

Κεφάλαιο 6

Skip List - Ουρές Προτεραιότητας

Πολλές εφαρμογές όπως αλγόριθμοι γράφων (π.χ Dijkstra, Prim), αλγόριθμοι συμπίεσης (π.χ. κώδικες Huffman), εργασίες σε λειτουργικά συστήματα (π.χ. load balancing, interrupt handling) κ.α., χρειάζονται να αντλήσουν δεδομένα σύμφωνα με κάποια προτεραιότητα. Αυτό το επιτυγχάνουν χρησιμοποιώντας *ουρές προτεραιότητας (priority queues).*

Η ουρά προτεραιότητας είναι ένας αφηρημένος τύπος δεδομένων που παρέχει τις εξής πράξεις:

- *insert(element, key)*, για την εισαγωγή ενός στοιχείου με ένα κλειδί και
- *extract_highest_priorty()*, για την επιστροφή και διαγραφή του πρώτου σε προτεραιότητα στοιχείου. Ανάλογα με το πως καθορίζουμε τη σειρά προτεραιότητας το στοιχείο που αντλούμε μπορεί να είναι το στοιχείο με το μεγαλύτερο ή με το μικρότερο κλειδί.

Στο εξής για λόγους ευκολίας θα θεωρούμε ότι το στοιχείο με τη μέγιστη προτεραιότητα είναι αυτό που έχει το μικρότερο κλειδί.

Οι περισσότερες σειριακές ουρές προτεραιότητας βασίζονται στη δομή δεδομένων *Heap*. Από την άλλη όμως, οι παράλληλες heap-based ουρές προτεραιότητας αντιμετωπίζουν προβλήματα όπως αυξημένο συνωστισμό κατά τη πρόσβαση σε κοινή μνήμη (memory contention) και στενωπά απόδοσης λόγω αναγκαστικής σειριοποίησης (sequential bottlenecks), όχι μόνο όταν εκτελούνται πράξεις διαγραφής ελαχίστου αλλά και όταν γίνεται μετακίνηση στοιχείων σε θέσεις με υψηλότερη προτεραιότητα.

Για την αντιμετώπιση των προβλημάτων αυτών προτάθηκε η χρησιμοποίηση ουρών προτεραιότητας βασισμένων σε *Skip Lists*. Στη βιβλιογραφία συναντάμε αρκετές παραλλαγές υλοποίησης της ουράς προτεραιότητας Skip List [21], [22], [7]. Στο κεφάλαιο αυτό θα αναπτύξουμε συνοπτικά δυο από τις πιο διαδεδομένες υλοποιήσεις, *την ουρά προτεραιότητας των Lotan και Shavit* [23] και την αντίστοιχη *των Linden και Jonsson* [24].

Στη συνέχεια θα ασχοληθούμε με μια προσεγγιστική ουρά προτεραιότητας που το τελευταίο διάστημα έχει κερδίσει το ενδιαφέρον της ερευνητικής κοινότητας, τη λεγόμενη *SprayList* των *Alistarh κ.α.* [25]. Επιπλέον, υλοποιήσαμε την ίδια ουρά προτεραιότητας χρησιμοποιώντας HTM και συγκρίναμε τις δυο υλοποιήσεις πειραματικά. Στο τέλος του κεφαλαίου παρουσιάζουμε τα αποτελέσματα των μετρήσεων.

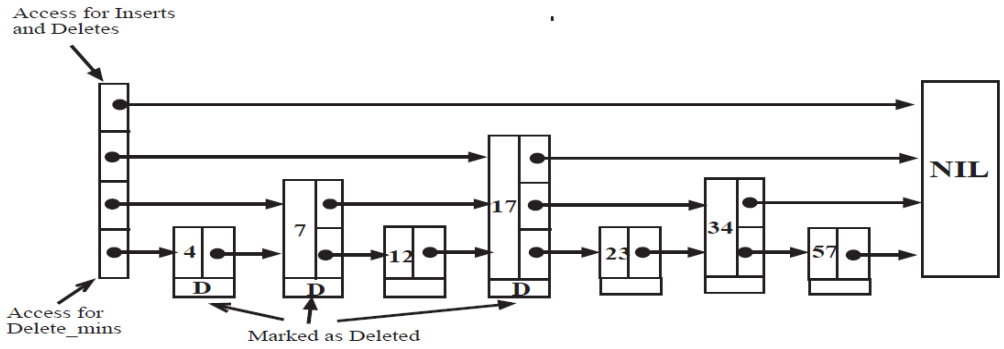
6.1 Βιβλιογραφικές αναφορές

6.1.1 Ουρά προτεραιότητας των Lotan και Shavit

Η ουρά προτεραιότητας των Lotan και Shavit [23], ή αλλιώς *SkipQueue* όπως την ονομάζουνε είναι η πρώτη παράλληλη Skip List ουρά προτεραιότητας που υλοποιήθηκε και προτάθηκε το 2000 με σκοπό να αντιμετωπίσει τα προβλήματα των Heap-based ουρών προτεραιότητας που ήταν δημοφιλή εκείνη την εποχή. Ας δούμε τώρα πως λειτουργεί η SkipQueue.

Όπως παρατηρούμε στο σχήμα 6.1 κάθε κόμβος της δομής έχει και από ένα deleted flag, το οποίο τίθεται στη τιμή false όταν εισέρχεται στην δομή. Όταν ένα νήμα εκτελεί την εντολή DeleteMin και θέλει να εντοπίσει τον κόμβο με το μικρότερο κλειδί αρχικά ξεκινά προσπελάζοντας τη λίστα του κατώτερου επιπέδου μέχρι να βρει έναν κόμβο του οποίου το delete flag δεν έχει γίνει set. Μόλις το βρει, του αλλάζει το delete flag σε true πραγματοποιώντας έτσι τη λογική διαγραφή του. Αυτό που μένει τώρα είναι να αφαιρέσει τον κόμβο από τη δομή. Επειδή το νήμα γνωρίζει πιο κόμβο θέλει να αφαιρέσει, χρησιμοποιεί την κλασική λειτουργία διαγραφής που είδαμε στο κεφάλαιο 3. Στην υλοποίησή τους οι Lotan και Shavit χρησιμοποιούν Lock-based τεχνικές, αλλά στην βιβλιογραφία υπάρχουν και αντίστοιχες Lock-free υλοποιήσεις [7], [11]. Είναι σίγουρο ότι δύο νήματα δεν θα διαγράψουν ποτέ τον ίδιο κόμβο καθώς μόνο ένα από τα δυο μπορεί να θέσει το delete flag. Αυτό συμβαίνει γιατί χρησιμοποιείται η register-to-memory εντολή *SWAP* για να τεθεί το delete flag. Αυτό επιτρέπει σε οποιονδήποτε αριθμό νημάτων να ψάξουν για το ελάχιστο στοιχείο ταυτόχρονα. Ο πρώτος που θα εκτελέσει με επιτυχία την εντολή SWAP η οποία θα θέσει το delete flag από false σε true θα προχωρήσει με την αφαίρεση του κόμβου ενώ οι υπόλοιποι θα συνεχίσουν προχωρώντας στην λίστα προσπαθώντας να βρουν τον επόμενο διαθέσιμο κόμβο.

Παρόλο που ο η παραπάνω υλοποίηση θα λειτουργούσε στην πράξη, δεν είναι *linearizable*: ένα νήμα μπορεί να εισάγει πρώτα ένα στοιχείο υψηλότερης προτεραιότητας (χαμηλότερης τιμής) και στη συνέχεια ένα στοιχείο χαμηλότερης προτεραιότητας ενώ ένα άλλο νήμα που προσπελαύνει την λίστα να βρει και να επιστρέψει το στοιχείο χαμηλότερης προτεραιότητας. Είναι όμως *quiescently consistent*: αν ένα στοιχείο x ήταν παρόν στη δομή πριν ξεκινήσει η κλήση DeleteMin, τότε το στοιχείο που επιστρέφει η κλήση θα έχει τιμή κλειδιού μικρότερη ή ίση από τη τιμή κλειδιού του x.



Σχήμα 6.1 *Η ουρά προτεραιότητας SkipQueue.*

Προκειμένου να γίνει ο αλγόριθμος linearizable προστέθηκε ένας μηχανισμός χρονοσφραγίδων (*timestamping mechanism*): κάθε νήμα που εκτελεί DeleteMin επιστρέφει το ελάχιστο στοιχείο ανάμεσα σε αυτά που εισήχθησαν πλήρως πριν ξεκινήσει να εκτελείται η εντολή. Αυτό επιτυγχάνεται ως εξής: Κατά τη λειτουργία εισαγωγής μόλις ο νέος κόμβος συνδεθεί πλήρως αποκτά μια χρονοσφραγίδα. Αντίστοιχα μια λειτουργία διαγραφής ελαχίστου σημειώνει τη χρονική στιγμή έναρξης αναζήτησης ελαχίστου και εξετάζει μόνο τους κόμβους που έχουν τιμή χρονοσφραγίδας μικρότερη της χρονικής στιγμής που σημείωσε, αγνοώντας με αυτόν τον τρόπο κόμβους που εισήχθησαν κατά τη διάρκεια προσπέλασης. Για την απόδειξη του linearizability παραπέμπουμε τον αναγνώστη στο πρωτότυπο κείμενο [23].

6.1.2 Ουρά προτεραιότητας των Linden και Jonsson

Περνάμε τώρα στην ουρά προτεραιότητας των Linden και Jonsson [24] η οποία αποτελεί την πιο αντιπροσωπευτική linearizable Skip List ουρά προτεραιότητας που συναντάμε στην βιβλιογραφία.

Όπως αναφέραμε και στην ενότητα 5.3, όταν ένας πυρήνας ενημερώνει κάποιο κοινό δεδομένο, αυτό που συμβαίνει αρχικά είναι η ακύρωση των όποιων αντιγράφων υπάρχουν στις άλλες caches και στη συνέχεια η cache-to-cache μεταφορά της cache line που περιέχει το δεδομένο στους πυρήνες που θα το ζητήσουν. Συνεπώς η ενημέρωση συχνά προσπελάσιμων θέσεων μνήμης οδηγεί σε σημαντικές καθυστερήσεις.

Το πρόβλημα αυτό είναι ιδιαίτερα αισθητό σε Skip List ουρές προτεραιότητας καθώς λόγω των DeleteMin λειτουργιών οι περισσότερες αλλαγές γίνονται στους πρώτους κόμβους της δομής (σε όλα τα επίπεδα), δηλαδή σε δεδομένα που είναι συχνά προσπελάσιμα.

Για την αντιμετώπιση του προβλήματος αυτού οι Linden και Jonsson δημιούργησαν έναν αλγόριθμο διαγραφής ελαχίστου σύμφωνα με τον οποίο οι κόμβοι διαγράφονται λογικά και όχι φυσικά και μόνο όταν ο συνολικός αριθμός των λογικά διαγραμμένων στοιχείων ξεπεράσει έναν συγκεκριμένο αριθμό προχωράμε σε μαζική αφαίρεση των μαρκαρισμένων κόμβων από τη δομή. Ο αλγόριθμος αυτός βοηθάει μεν στην αντιμετώπιση του προβλήματος του κόστους των cache-to-cache μεταφορών, δημιουργεί δε το πρόβλημα ότι οι παράλληλες DeleteMin λειτουργίες αναγκάζονται να διαβάσουν πιο πολλά δεδομένα προκειμένου να βρουν το στοιχείο με το ελάχιστο κλειδί. Με βάση έρευνες σχετικές με το κόστος λειτουργιών στην μνήμη [26] οι Linden και Jonsson συμπεραίνουν ότι το κόστος των επιπλέον αναγνώσεων είναι σχετικά μικρό συγκριτικά με το κόστος που δημιουργείται από τον μεγάλο αριθμό των καθολικών εγγραφών (global writes). Το συμπέρασμα αυτό επιβεβαιώνεται και πειραματικά.

Όλες οι λειτουργίες είναι lock-free. Κάθε κόμβος της δομής κρατάει και από μια τιμή μαρκαρίσματος η οποία αποθηκεύεται μαζί με τον δείκτη προς το επόμενο στοιχείο του κατώτερου επιπέδου και εάν είναι set υποδηλώνει ότι ο επόμενος κόμβος στο κατώτερο επίπεδο είναι λογικά διαγραμμένος. Όπως θα δούμε και στην περιγραφή της λειτουργίας εισαγωγής το bit μαρκαρίσματος μας βοηθάει να αποτρέψουμε την εισαγωγή στοιχείων ανάμεσα σε κόμβους που είναι διαγραμμένοι λογικά. Έτσι οι λογικά διαγραμμένοι κόμβοι συνιστούν ένα πρόθεμα της λίστας κατώτερου επιπέδου.

Λειτουργία Διαγραφής Ελαχίστου

Για την διαγραφή ελαχίστου αρχικά προσπελαύνουμε την λίστα κατώτερου επιπέδου μέχρι να βρούμε έναν κόμβο του οποίου το delete flag δεν έχει γίνει set. Μόλις βρεθεί, τότε αλλάζουμε με ατομική εντολή το delete flag σε true πραγματοποιώντας έτσι τη λογική διαγραφή του. Σε περίπτωση που ο συνολικός αριθμός των στοιχείων που προσπελάσαμε είναι μεγαλύτερος από ένα συγκεκριμένο κατώφλι τότε προχωράμε στη λειτουργία αναδόμησης. Σε αντίθετη περίπτωση απλά επιστρέφουμε.

Η *λειτουργία αναδόμησης* (*restructure operation*) αποσκοπεί στην μαζική αφαίρεση (φυσική διαγραφή) λογικά διαγραμμένων κόμβων από όλα τα επίπεδα. Αρχικά στο κατώτερο επίπεδο ενημερώνουμε τον δείκτη της κεφαλής να δείχνει στον κόμβο που μόλις διαγράψαμε λογικά ή αν κατά τη διάρκεια προσπέλασης συναντήσαμε κόμβο που δεν είχε εισαχθεί πλήρως αλλά στο μεταξύ είχε διαγραφεί από άλλο νήμα τότε ενημερώνουμε τον δείκτη της κεφαλής να δείχνει σε αυτόν τον κόμβο. Στη συνέχεια ξεκινώντας από το ανώτερο επίπεδο και πηγαίνοντας προς τα κάτω συνδέουμε κάθε φορά την κεφαλή με τον τελευταίο κόμβο του επιπέδου που έχει διαγραφεί λογικά.

Λειτουργία Εισαγωγής

Η λειτουργία εισαγωγής είναι παρόμοια της κλασικής lock-free λειτουργίας εισαγωγής που είδαμε στο τρίτο κεφάλαιο με τη διαφορά ότι ο νέος κόμβος συνδέεται στο κατώτερο επίπεδο μόνο εάν ο διάδοχος κόμβος στο επίπεδο αυτό δεν είναι λογικά διαγραμμένος. Εάν είναι τότε επαναλαμβάνει την διαδικασία εύρεσης προκατόχων και διαδόχων μέχρις ότου μπορέσει να εισαχθεί. Στο σημείο αυτό αξίζει να τονίσουμε για πιο λόγο το deleted flag ενός κόμβου δεν βρίσκεται σε κάποιο πεδίο του ιδίου κόμβου αλλά στο πεδίο του προκατόχου μαζί με το δείκτη προς το επόμενο στοιχείο. Όταν συνδέουμε τον προκάτοχο με τον νέο κόμβο εκτελούμε μια ατομική εντολή σαν και αυτή :

pred.next[0].compareAndSet(succ, newNode, false, false);

Στη πράξη ο δείκτης προς τον διάδοχο (πρώτο όρισμα στην παρένθεση) και το deleted flag του διαδόχου (τρίτο όρισμα) είναι μια λέξη και αλλάζει ατομικά στη λέξη που συντίθεται από το δεύτερο και τέταρτο όρισμα που είναι η διεύθυνση του νέου κόμβου και το bit μαρκαρίσματός του. Συνεπώς με μια ατομική εντολή ελέγχουμε αν ο διάδοχος κόμβος έχει διαγραφεί λογικά και εάν όχι προχωράμε στη σύνδεση του νέου κόμβου στο κατώτερο επίπεδο. Όπως βλέπουμε κάτι τέτοιο είναι εφικτό μόνο επειδή το deleted flag βρίσκεται σε πεδίο του προκατόχου και όχι σε πεδίο του ιδίου κόμβου.

Αφού επομένως εισαχθεί με επιτυχία ο νέος κόμβος επιχειρούμε να τον συνδέσουμε και στα υπόλοιπα επίπεδα. Πριν γίνει οποιαδήποτε σύνδεση ελέγχουμε εάν ο νέος κόμβος έχει διαγραφεί από κάποιο παράλληλο νήμα που εκτελεί την λειτουργία διαγραφής ελαχίστου. Εάν ελεγχθεί ότι ο νέος κόμβος έχει διαγραφεί τότε σταματάει τη λειτουργία εισαγωγής και επιστρέφει. Σε αντίθετη περίπτωση συνεχίζουμε τη σύνδεση ακριβώς όπως και στον lock-free αλγόριθμο που είδαμε στο τρίτο κεφάλαιο.

Για περισσότερες λεπτομέρειες σχετικά με τον αλγόριθμο καθώς και για απόδειξη linearizability παραπέμπουμε τον αναγνώστη στο πρωτότυπο κείμενο [24].

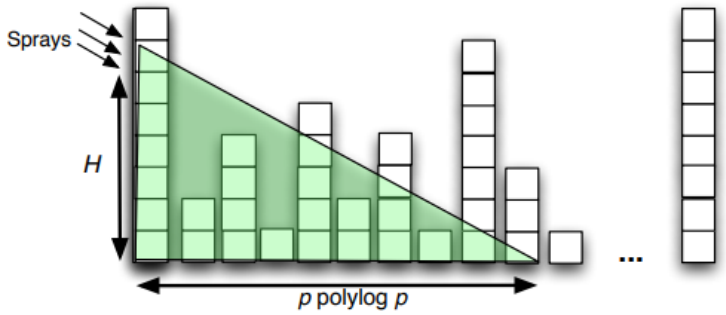
6.2 Προσεγγιστική ουρά προτεραιότητας Spray List

Στην ενότητα αυτή ασχολούμαστε με την *προσεγγιστική* (*relaxed*) ουρά προτεραιότητας *SprayList* των *Alistarh κ.α.* [25]. Όταν λέμε προσεγγιστική ουρά προτεραιότητας εννοούμε μια ουρά προτεραιότητας που δεν επιστρέφει κατ' ανάγκη το στοιχείο με το ελάχιστο κλειδί αλλά μπορεί να επιστρέψει και ένα στοιχείο με κλειδί αρκετά κοντά στο ελάχιστο. Εδώ εγείρεται το ερώτημα κατά πόσο μια τέτοια ουρά προτεραιότητας έχει πρακτική εφαρμογή. Αυτό εξαρτάται από το κατά πόσο είμαστε διατεθειμένοι να θυσιάσουμε ακρίβεια στα αποτελέσματά μας για επίτευξη υψηλότερης κλιμακωσιμότητας. Αυτό που παρατηρούμε είναι ότι ολοένα και περισσότερες εφαρμογές χρησιμοποιούν προσεγγιστικές ουρές προτεραιότητας. Για παράδειγμα η SprayList έχει χρησιμοποιηθεί με επιτυχία σε προσομοιώσεις διακριτών γεγονότων (discrete-event simulation) και σε προβλήματα ελαχίστων μονοπατιών από κοινή αφετηρία (single source shortest paths problems) όπως σε κοινωνικούς γράφους, οδικά δίκτυα και πλέγματα.

Ο βασικός παράγοντας που περιορίζει την απόδοση των Skip List-based ουρών προτεραιότητας που είδαμε μέχρι στιγμής είναι το γεγονός ότι όλα τα νήματα που εκτελούν την εντολή DeleteMin συνωστίζονται στο πρώτο στοιχείο - πρακτικά στα p πρώτα στοιχεία αν έχουμε p νήματα να καλούν την DeleteMin ταυτόχρονα - γεγονός που οφείλεται στους περιορισμούς εγκυρότητας που πρέπει να πληρούνται (linearizability, quiescent consistency). Αντίθετα αν χαλαρώσουμε τους περιορισμούς και επιτρέψουμε στα νήματα να επιστέψουν στοιχεία με τιμή κλειδιού κοντά στο ελάχιστο τότε μπορούμε να βελτιώσουμε την κλιμακωσιμότητα.

6.2.1 Περιγραφή αλγορίθμου

Η βασική ιδέα της ουράς προτεραιότητας SprayList είναι η εκτέλεση μιας relaxed εντολής DeleteMin σύμφωνα με την οποία ξεκινάμε από ένα συγκεκριμένο ύψος του πρώτου στοιχείου και διασχίζουμε τη δομή σύμφωνα με ένα τυχαίο περίπατο καταλήγοντας σε ένα από τα πρώτα $O(p\ log^3\ p)$ στοιχεία της δομής, όπου p είναι ο αριθμός των νημάτων που τρέχουνε (σχήμα 6.2).



Σχήμα 6.2 *Η ιδέα πίσω από τη SprayList. Τα νήματα ξεκινούν από το ύψος H και εκτελούν ένα τυχαίο περίπατο προσπελαύνοντας κόμβους στην αρχή της δομής και στη συνέχεια προσπαθούν να αποκτήσουν τον κόμβο στον οποίον καταλήγει ο περίπατος.*

Ας δούμε τώρα τις λειτουργίες της SprayList πιο αναλυτικά:

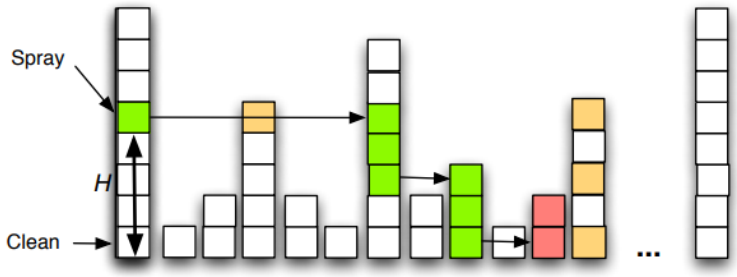
Λειτουργίες Αναζήτησης και Εισαγωγής

Οι λειτουργίες αναζήτησης και εισαγωγής είναι πανομοιότυπες με αυτές της κλασικής lock-free Skip List που είδαμε στο κεφάλαιο 3.

Λειτουργία Διαγραφής Ελαχίστου

Η λειτουργία διαγραφής ελαχίστου χωρίζεται σε δύο επιμέρους λειτουργίες. Πρώτα εκτελείται η λειτουργία **Spray** η οποία στοχεύει να πραγματοποιήσει μια ομοιόμορφη επιλογή ανάμεσα στα πρώτα $O(p\ log^3\ p)$ στοιχεία της δομής και ύστερα εκτελείται η λειτουργία **DeleteKey** που στοχεύει να αφαιρέσει από τη δομή το στοιχείο που επιλέχθηκε από την Spray και είναι πανομοιότυπη με την κλασική lock-free DeleteKey που είδαμε στο κεφάλαιο 3.

Η Spray λειτουργεί ως εξής: Αρχικά ξεκινάμε από το επίπεδο $H = log\ p + K$ όπου p είναι ο αριθμός των νημάτων που εκτελούνται και K είναι μια σταθερά. Σε κάθε επίπεδο l της δομής αρχικά εκτελούμε έναν τυχαίο αριθμό οριζόντιων βημάτων e, με το e να επιλέγεται ομοιόμορφα στο διάστημα [0,L], όπου $L = M\ log^3\ p$ και $M \geq 1$ σταθερά. Στη συνέχεια κατεβαίνουμε D επίπεδα, όπου $D = \max(1, \lfloor \log \log p \rfloor)$. Στη πράξη, συνήθως είναι $D = 1$. Έπειτα συνεχίζουμε με την οριζόντια προσπέλαση. Επαναλαμβάνουμε την διαδικασία αυτή μέχρις ότου φτάσουμε στο κατώτερο επίπεδο. Εφόσον βρισκόμαστε στο κατώτερο επίπεδο και έχουμε ολοκληρώσει την οριζόντια διαδικασία προσπέλασης επιχειρούμε να αποκτήσουμε (να μαρκάρουμε κατά κάποιον τρόπο) το τρέχον στοιχείο. Αν επιτύχουμε, συνεχίζουμε καλώντας την κλασική λειτουργία DeleteKey που έχουμε δει. Αν αποτύχουμε, είτε επαναλαμβάνουμε τη λειτουργία Spray από την αρχή, είτε με μικρή πιθανότητα το νήμα γίνεται cleaner thread και προσπελαύνει γραμμικά τη κατώτερη λίστα μέχρις ότου βρει διαθέσιμο κόμβο. Για περισσότερες λεπτομέρειες καθώς και για ορισμένες βελτιστοποιήσεις που προτείνονται παραπέμπουμε τον αναγνώστη στη σχετική δημοσίευση [25].



Σχήμα 6.3 *Ένα απλό παράδειγμα της λειτουργίας Spray. Η Spray κατά τη διαδικασία προσπέλασης περνάει από τους πράσινους κόμβους και σταματάει στον κόκκινο κόμβο. Οι πορτοκαλί κόμβοι μπορούσαν να είχαν επιλεγεί για άλματα αλλά δεν επιλέχθηκαν.*

6.2.2 Υλοποίηση SprayList με Hardware Transactional Memory και πειραματική αξιολόγηση

Το γεγονός ότι στις κλασικές Skip List ουρές προτεραιότητας τα νήματα δρουν σε γειτονικούς κόμβους για την εξαγωγή ελαχίστων στοιχείων καθιστά ανούσια την υλοποίηση αντίστοιχων HTM εκδοχών, καθώς λόγω των αναπόφευκτων conflicts που δημιουργούνται το ποσοστό των aborts θα είναι πολύ μεγάλο.

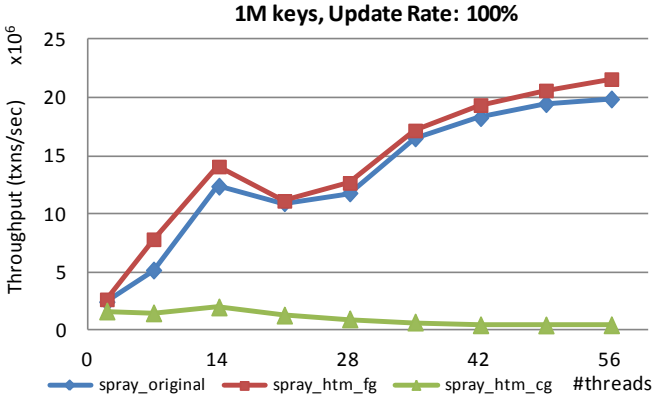
Αντίθετα η εκτέλεση DeleteMin της SprayList μοιάζει με την εκτέλεση της DeleteKey κανονικών Skip Lists σε ένα σετ $O(p\ log^3\ p)$ δεδομένων. Συνεπώς η καλή εικόνα των HTM υλοποιήσεων σε μικρά data sets που είδαμε στο προηγούμενο κεφάλαιο μας προϊδεάζει ότι μια HTM υλοποίηση της SprayList μπορεί να έχει ανταγωνιστική επίδοση.

Για να το διαπιστώσουμε, υλοποιήσαμε μια HTM εκδοχή της SprayList όπου οι λειτουργίες αναζήτησης, εισαγωγής και διαγραφής στοιχείου είναι πανομοιότυπες με εκείνες της κλασικής Skip List και η λειτουργία Spray είναι ίδια με εκείνης της lock-free εκδοχής που μόλις είδαμε χωρίς όμως το cleaning. Στην fine-grained HTM SprayList εκδοχή η λειτουργία Spray εκτελείται εκτός transaction.

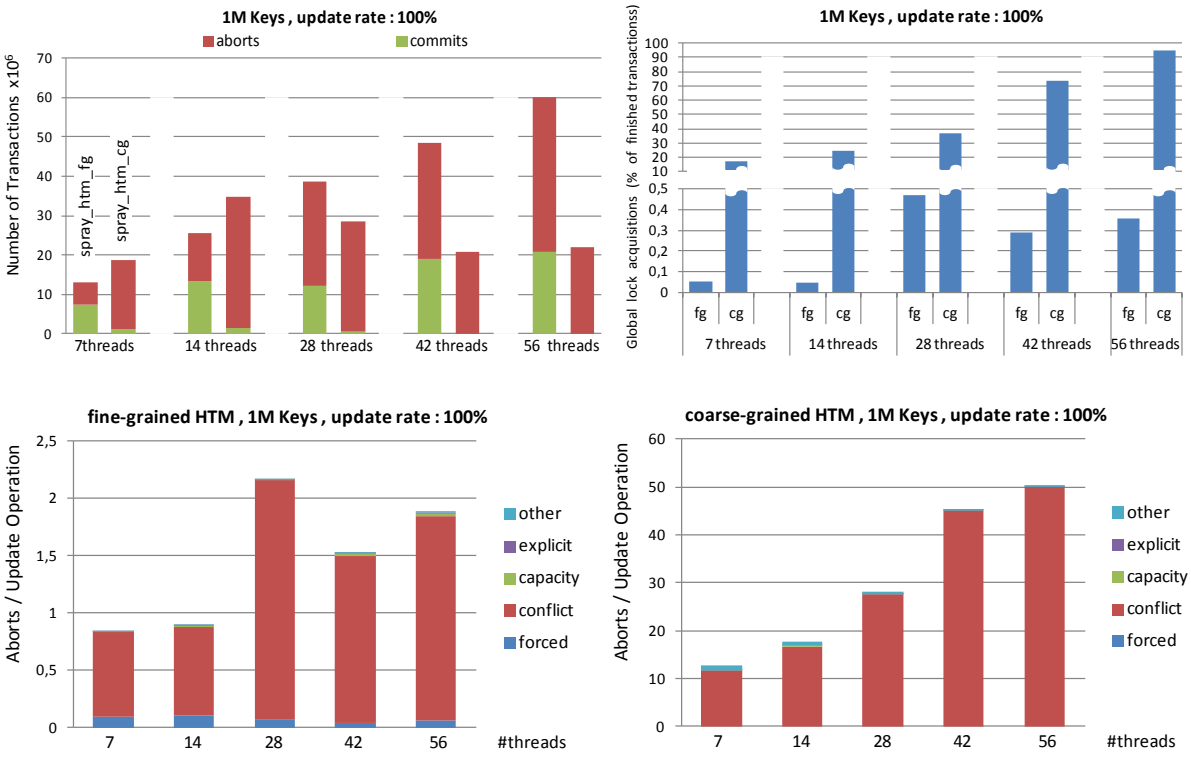
Μετρήσαμε το throughput των SprayList υλοποιήσεων που συζητήσαμε για σετ δεδομένων 1 εκατ. στοιχείων και ρυθμό ενημερώσεων 100% με εναλλαγές εισαγωγών/διαγραφών (λειτουργία alternate). Επιπλέον μετρήσαμε τα στατιστικά των transactions (commits/aborts) των fine-grained και coarse-grained HTM υλοποιήσεων. Παρουσιάζουμε τα αποτελέσματά μας στα σχήματα 6.4 και 6.5 που ακολουθούν.

Παρατηρώντας τις γραφικές του σχήματος 6.5 αρχικά διαπιστώνουμε εύκολα ότι η coarse-grained HTM σχεδίαση είναι ακατάλληλη για υλοποίηση SprayList. Τα ποσοστά των aborts ξεπερνούν το 90% και ένας μεγάλος αριθμός νημάτων δεν ολοκληρώνει την λειτουργία του εντός transaction αλλά έχοντας καταλάβει το fallback-lock, ύστερα από 50 επαναδοκιμές (τόσες έχουν προκαθοριστεί). Αυτό οφείλεται στο μεγάλο πλήθος επικαλυπτόμενων footprints των transactions στην αρχή της δομής.

Αντίθετα η fine-grained HTM υλοποίηση εκμεταλλευόμενη τη λειτουργία Spray καταφέρνει να περιορίσει τα conflicts γεγονός που φαίνεται τόσο στα στατιστικά των transactions (≤ 2 aborts ανά λειτουργία ενημέρωσης) όσο και στα ποσοστά κλειδωμάτων (< 0,4%). Κατά συνέπεια όπως παρατηρούμε στο σχήμα 6.4 καταφέρνει να έχει ανταγωνιστική επίδοση, μάλιστα λίγο καλύτερη σε σχέση με την κλασική SprayList. Η "κοιλιά" στην απόδοση για 14 ως 28 threads οφείλεται στις παρενέργειες του NUMA.



Σχήμα 6.4 *Επίδοση παράλληλων SprayList υλοποιήσεων.*



Σχήμα 6.5 *Στατιστικά transactions. Πάνω αριστερά απεικονίζεται ο αριθμός των commits και aborts ενώ πάνω δεξιά φαίνεται το ποσοστό των κλειδωμάτων. Κάτω φαίνονται πόσα aborts χρειάστηκαν κατά μέσο όρο για την ολοκλήρωση μιας λειτουργίας update καθώς και ανάλυση των aborts στις επιμέρους συνιστώσες.*

Κεφάλαιο 7

Επίλογος

Με το κεφάλαιο αυτό ολοκληρώνεται η παρούσα διπλωματική εργασία και μαζί με αυτήν ολοκληρώνεται μια προσπάθεια ενάμιση περίπου χρόνου. Στο διάστημα αυτό είχα την ευκαιρία να εμβαθύνω στη περιοχή του παράλληλου προγραμματισμού, να πειραματιστώ με HTM γράφοντας αρκετό κώδικα, να μάθω τεχνικές συγχρονισμού ακόμα και σε δομές δεδομένων διαφορετικές από Skip List, όπως B+ δέντρα, δυαδικά δέντρα, ουρές κ.α. και εν τέλη να συμβάλω στη δημιουργία νέας γνώσης μελετώντας και αξιολογώντας το HTM για την παραλληλοποίηση Skip Lists.

7.1 Συμπεράσματα

Στο πρώτο κεφάλαιο είχαμε αναφερθεί στους στόχους της διπλωματικής. Ας τους ξαναθυμηθούμε και ας εξετάσουμε κατά πόσο μπορέσαμε να τους πετύχουμε.

Οι στόχοι που είχαμε θέσει ήταν οι εξής:

1) *Να υλοποιήσουμε και να αξιολογήσουμε διάφορες τεχνικές HTM σε Skip Lists και ουρές προτεραιότητας.*

Στο τέταρτο κεφάλαιο είδαμε πως να υλοποιήσουμε παράλληλες Skip Lists χρησιμοποιώντας τεχνικές coarse-grained και fine-grained HTM. Στο πέμπτο κεφάλαιο αναλύσαμε τα πλεονεκτήματα και μειονεκτήματα κάθε τεχνικής τόσο θεωρητικά όσο και πρακτικά και στη συνέχεια στο έκτο τις εξετάσαμε πάνω στην προσεγγιστική ουρά προτεραιότητας Spray List.

Συμπεραίνουμε ότι η επιλογή του coarse-grained HTM είναι μια αρκετά καλή επιλογή εάν επιθυμούμε προγραμματιστική ευκολία και έγκυρες λειτουργίες αλλά εάν επιθυμούμε υψηλές επιδόσεις η επιλογή του fine-grained HTM είναι μονόδρομος.

2) *Να μελετήσουμε την επίδραση που έχουν ορισμένες παράμετροι όπως το hyper-threading ή ένα σύστημα NUMA στην λειτουργία HTM υλοποιήσεων.*

Στο πέμπτο κεφάλαιο είδαμε πως το hyper-threading επηρεάζει τις HTM υλοποιήσεις και πιο συγκεκριμένα πόσο αυξάνονται τα capacity aborts στις coarse-grained και fine-grained HTM υλοποιήσεις. Επίσης είδαμε πως οι παρενέργειες του NUMA μεγεθύνονται σε HTM υλοποιήσεις, αυξάνοντας κατά πολύ τα conflict aborts, καθώς και εξετάσαμε ορισμένες λύσεις του προβλήματος.

Συμπεραίνουμε ότι ναι μεν οι παράμετροι αυτοί αποτελούν περιοριστικούς παράγοντες στην κλιμακωσιμότητα παράλληλων HTM υλοποιήσεων, υπάρχουν όμως

αποτελεσματικοί τρόποι να αντιμετωπιστούν όπως 1) ο περιορισμός του footprint των transactions για τη μείωση των capacity aborts και 2) η αποδοτικότερη κατανομή των εργασιών στα διάφορα sockets του συστήματος με ταυτόχρονη ελαχιστοποίηση της επικοινωνίας μεταξύ τους για την μείωση των conflict aborts.

3) *Να συγκρίνουμε τις HTM υλοποιήσεις μας με τις state-of-the-art που βασίζονται σε Lock-Based και Lock-Free αλγορίθμους και να εξάγουμε συμπεράσματα.*

Κάθε τεχνική συγχρονισμού έχει τα πλεονεκτήματα και τα μειονεκτήματά της. Στο τρίτο κεφάλαιο είδαμε τις state-of-the-art υλοποιήσεις Skip List βασισμένες σε Lock-Based και Lock-Free τεχνικές. Ήταν πολύ σημαντικό να μπορέσουμε να στοιχειοθετήσουμε επιχειρήματα που θα υποδείκνυαν πότε κάθε τεχνική είναι κατάλληλη και πότε όχι. Για το σκοπό αυτό στο πέμπτο κεφάλαιο παρουσιάσαμε και αξιολογήσαμε τα αποτελέσματα ενός μεγάλου αριθμού πειραμάτων που πραγματοποιήσαμε.

Συμπεραίνουμε ότι η coarse-grained HTM υλοποίηση έχει ικανοποιητική επίδοση σε περιπτώσεις με μικρή συμφόρηση και όταν η πλειοψηφία των λειτουργιών είναι αναγνώσεις. Στις υπόλοιπες περιπτώσεις υστερεί σημαντικά σε αντίθεση με την fine-grained υλοποίηση η οποία έχει ανταγωνιστική επίδοση σε όλες τις περιπτώσεις. Μάλιστα όταν ο συνολικός αριθμός των νημάτων που εκτελούνται είναι μεγαλύτερος από τον αριθμό των πυρήνων του συστήματος, τότε η fine-grained HTM υλοποίηση αποτελεί την καλύτερη επιλογή καθώς υπερτερεί σημαντικά σε απόδοση σε σχέση με τις lock-based και lock-free υλοποιήσεις με τη διαφορά να φτάνει σε ορισμένες περιπτώσεις έως και 250%.

7.2 Μελλοντικές επεκτάσεις

Μια αρκετά σημαντική περιοχή έρευνας που αφορά τις παράλληλες δομές δεδομένων αποτελεί η ασφαλής ανάκτηση μνήμης (*safe memory reclamation*). Όταν αφαιρούμε έναν κόμβο από τη δομή δεν μπορούμε απλά να ελευθερώσουμε τη μνήμη που καταλαμβάνει καθώς άλλα νήματα μπορεί ακόμα να επενεργούν στον κόμβο. Συνεπώς χρειαζόμαστε πιο πολύπλοκες τεχνικές ανάκτησης μνήμης. Στο πλαίσιο της διπλωματικής δεν ασχοληθήκαμε με memory reclamation καθώς θέλαμε να εστιάσουμε στις τεχνικές συγχρονισμού αυτές καθαυτές. Επομένως, μια ενδιαφέρουσα επέκταση των υλοποιήσεών μας θα ήταν η προσθήκη κάποιου μηχανισμού ανάκτησης μνήμης.

Στη βιβλιογραφία συναντάμε διάφορες τεχνικές ανάκτησης μνήμης τις οποίες μπορούμε να εντάξουμε σε τρεις κατηγορίες. Πρώτον, τις *quiescence-based* τεχνικές [27], [28] σύμφωνα με τις οποίες τα νήματα προχωρούν σε ανάκτηση μνήμης όταν εισέλθουν σε μία κατάσταση ηρεμίας στην οποία κανένα νήμα δεν διατηρεί αναφορά σε κοινό κόμβο. Δεύτερον τεχνικές καταμετρήσεων αναφορών (*reference counting*) [29], [30], [31] και τέλος *pointer-based* τεχνικές όπως *hazard pointers* [32], *pass-the-buck* [33] ή *drop-the-anchor* [34] σύμφωνα με τις οποίες μαρκάρουμε κόμβους που είναι προσπελάσιμοι από άλλα νήματα έτσι ώστε να αποτρέψουμε πιθανή ανάκτηση μνήμης. Τα τελευταία χρόνια έχουν αναπτυχθεί και τεχνικές που χρησιμοποιούν *HTM* [35], [36].

Στο τέταρτο κεφάλαιο όταν αναπτύξαμε τον fine-grained HTM αλγόριθμο αναφερθήκαμε σε ορισμένες βελτιστοποιήσεις που ενώ θεωρητικά θα βελτίωναν την

επίδοση των υλοποιήσεών μας, πρακτικά είτε δεν την βελτίωσαν είτε δεν ήταν εφικτό να υλοποιηθούν. Μια πιθανή μελλοντική επέκταση θα έβρισκε τρόπο να αξιοποιήσει τις βελτιστοποιήσεις αυτές.

Τέλος, στην ενότητα που συζητήσαμε για τους τρόπους αντιμετώπισης των παρενεργειών του NUMA διαπιστώσαμε ότι υπάρχουν περιθώρια βελτίωσης και συνεπώς μελλοντικές επεκτάσεις μπορούν να ερευνήσουν αποδοτικότερες μεθόδους αντιμετώπισης των NUMA επιπτώσεων.

Βιβλιογραφία

[1] *M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93, (New York, NY, USA), pp. 289–300, ACM, 1993.*

[2] Wikipedia: The Free Encyclopedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Transactional_memory#Available_implementations.

[3] *Intel 64 and ia-32 architectures optimization reference manual https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.*

[4] *W. Pugh. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM, 33(6):668–676, June 1990.*

[5] *D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "Combining HTM and RCU to ImplementHighly Efficient Balanced Binary Search Trees," TRANSACT, 2017.*

[6] *H. Avni and B. C. Kuszmaul, "Improving htm scaling with consistency- oblivious programming," TRANSACT , vol. 6, p. 5, 2014.*

[7] *M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.*

[8] *Pugh W. A Skip List Cookbook. Tech. Rep. UMIACS–TR–89–72.1, 1989.*

[9] *H. Schweizer, M. Besta, T. Hoefler: Evaluating the Cost of Atomic Operations on Modern Architectures presented in San Francisco, CA, USA, ACM, Oct. 2015.*

[10] *T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.*

[11] *URL: http://lpd.epfl.ch/site/ascylib.*

[12] *M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. SIROCCO '07.*

[13] *M. Herlihy, Y. Lev, andN. Shavit. Concurrent lock-free skiplist with wait-free contains operator, May 3 2011. US Patent 7,937,378.*

[14] *G. Chadha, S. Mahlke, and S. Narayanasamy. When less is more (LIMO): Controlled parallelism for improved efficiency. In Proc. International Conference on Compilers,*

Architectures and Synthesis for Embedded Systems (CASES), pages 141-150, 2012.

[15] *K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In Proc. IEEE International Symposium on Workload Characterization (IISWC), pages 116-125, 2011.*

[16] *A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. In Proc. ACM PLDI, pages 26-37, 2011.*

[17] *Brown T., Kogan A., Lev Y., Luchangco V., "Investigating the Performance of Hardware Transactions on a Multi-Socket Machine", in 11th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2016.*

[18] *J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In Proc. USENIX ATC, 2012.*

[19] *I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In Proc. Int. Conference on Principles of Distributed Systems OPODIS, pp.83-97,2013.*

[20] *A. Hassan, R. Palmieri, and B. Ravindran. Remote transaction commit: Centralizing software transactional memory commits. IEEE Transactions on Computers, pages 26-33, 2015.*

[21] *C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. Introduction to algorithms. The MIT press, 2001.*

[22] *Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multithread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (May 2005).*

[23] *I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, pages 263–268. IEEE, 2000.*

[24] *J. Linden and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Principles of Distributed Systems, pages 206–220. Springer, 2013.*

[25] *Dan Alistarh, Justin Kopinsky, Jerry Li, Nir Shavit: The SprayList: a scalable relaxed priority queue. PPOPP 2015: 11-20.*

[26] *Molka, D., Hackenberg, D., Schone, R., Muller, M.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: PACT. pp. 261-270. ACM (2009).*

[27] *T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In Proceedings of*

the International Conference on Distributed Computing (DISC), pages 300–314, 2001.

[28] *T. E. Hart, P. E. McKenney, A. D. Brown, and J.Walpole. Performance of memory reclamation for lockless synchronization. J. Parallel Distrib. Comput., 67(12):1270–1285, 2007.*

[29] *D. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr. Lock-free reference counting. Distributed Computing, 15(4):255–271, 2002.*

[30] *A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. IEEE Trans. Parallel Distrib. Syst., 20(8): 1173–1187, 2009.*

[31] *J. D. Valois. Lock-free linked lists using compare-and-swap. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 214–222, 1995.*

[32] *M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst., 15 (6):491–504, 2004.*

[33] *M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lockfree data structures. In Proceedings of the 16th International Conference on Distributed Computing (DISC), pages 339– 353, 2002.*

[34] *A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In Proc. of the 25th ACM symposium on Parallelism in algorithms and architectures, SPAA '13, pages 33– 42, New York, USA, 2013. ACM.*

[35] *A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 99– 108, 2011.*

[36] *P. Pirkelbauer, A. Wilson, H. Ahmed, R. Milewicz, "Memory Management for Concurrent Data Structures on Hardware Transactional Memory", TRANSACT, 2017.*

[37] *A. Kleen, "Tsx anti patterns in lock elision code.".*