Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Τεχνικές Ταυτοχρονισμού για την Υλοποίηση Αποδοτικών Δένδρων Αναζήτησης σε Πολυπύρηνα Συστήματα

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Δημήτριος Δ. Σιακαβάρας

Αθήνα, Δεκέμβριος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Τεχνικές Ταυτοχρονισμού για την Υλοποίηση Αποδοτικών Δένδρων Αναζήτησης σε Πολυπύρηνα Συστήματα

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Δημήτριος Δ. Σιακαβάρας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 23η Δεκεμβρίου 2021.

Αθήνα, Δεκέμβριος 2021

. . . .

Δημήτριος Δ. Σιακαβάρας
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο (2021)

Η παρούσα διατριβή αφιερώνεται στους γονείς μου
Δημήτρη και Μελπομένη στους οποίους χρωστάω τα πάντα!

Περίληψη

Τα δένδρα αναζήτησης αποτελούν μία από τις πιο κλασσικές και ευρέως διαδεδομένες δομές δεδομένων. Χρησιμοποιούνται σε εφαρμογές όπου απαιτείται η διατήρηση μεγάλου ταξινομημένου όγκου δεδομένων με δυνατότητα γρήγορης αναζήτησης, εισαγωγής, διαγραφής και επιπλέον λειτουργιών, όπως είναι η αναζήτηση εύρους τιμών. Λόγω της σημασίας τους, ένα μεγάλο πλήθος ερευνητικών εργασιών έχει προτείνει πολλούς διαφορετικούς τύπους δένδρων με διαφορετικά χαρακτηριστικά όπως είναι, για παράδειγμα, το μέγιστο μήκος που μπορεί να έχει ένα μονοπάτι μέσα στο δένδρο. Κάθε τύπος δένδρου προσφέρει και διαφορετικές εγγυήσεις επίδοσης για την κάθε λειτουργία και κάθε δένδρο επιλέγεται με βάση τις ανάγκες της εκάστοτε εφαρμογής στην οποία θα ενσωματωθεί.

Με την επικράτηση των πολυπύρηνων επεξεργαστών, όπου πολλαπλά νήματα εκτελούνται ταυτόχρονα και πιθανώς προσπελαύνουν κοινά δεδομένα, οι ταυτόχρονες δομές δεδομένων έχουν γίνει σημαντικό μέρος των εφαρμογών αυτών. Στις ταυτόχρονες δομές δεδομένων είναι αναγκαίος ο συντονισμός των ταυτόχρονων προσπελάσεων από διαφορετικά νήματα με τρόπο που να διατηρείται η ακεραιότητα της δομής και να εξασφαλίζεται η ορθή εκτέλεση όλων των επιμέρους λειτουργιών. Ο συντονισμός αυτός επιτυγχάνεται με τη χρήση κάποιου μηχανισμού συγχρονισμού όπως για παράδειγμα τα κλειδώματα, οι εντολές ατομικής προσπέλασης μνήμης που παρέχονται απο τους σύγχρονους επεξεργαστές, η τεχνική Διάβασε-Αντίγραψε-Ανανέωσε (Read-Copy-Update) και η μνήμη δοσοληψιών (Transactional Memory).

Τα ταυτόχρονα δένδρα αναζήτησης είναι μία από τις πιο ευρέως χρησιμοποιούμενες δομές δεδομένων για την αποθήκευση και ανάκτηση δεδομένων σε σύγχρονες πολυνηματικές εφαρμογές. Παρά τον πολύ μεγάλο όγκο σχετικής δουλειάς, παραμένει ακόμα σημαντική πρόκληση η υλοποίηση ταυτόχρονων δένδρων αναζήτησης υψηλών επιδόσεων. Αυτό οφείλεται κυρίως

στο γεγονός πως τόσο οι κλασσικές μέθοδοι συγχρονισμού (δηλαδή η χρήση κλειδωμάτων και η χρήση ατομικών λειτουργιών) όσο και οι πιο πρόσφατες (δηλαδή η τεχνική Read-Copy-Update και η Transactional Memory) δεν είναι αρκετές από μόνες τους ώστε να προσφέρουν λύσεις που θα είναι γενικές και εύκολα υλοποιήσιμες αλλά και την ίδια στιγμή θα προσφέρουν υψηλές επιδόσεις σε διαφορετικά σενάρια εκτέλεσης και επίπεδα συμφόρησης στη δομή.

Μέχρι πρόσφατα, η Transactional Memory χρησιμοποιούταν κυρίως μέσω κάποιας βιβλιοθήκης που την υλοποιούσε σε επίπεδο λογισμικού. Ωστόσο, τα τελευταία χρόνια δύο απο τις μεγαλύτερες εταιρείες παραγωγής επεξεργαστών, η Intel και η IBM, έχουν προσθέσει υποστήριξη για Transactional Memory σε επίπεδο υλικού, αφαιρώντας με αυτό τον τρόπο τις μεγάλες καθυστερήσεις που εισάγονταν από τις υλοποιήσεις σε επίπεδου λογισμικού. Σε αυτή την εργασία εξετάζουμε τους τρόπους με τους οποίους μπορεί να χρησιμοποιηθεί η Transactional Memory για την υλοποίηση ταυτόχρονων δένδρων αναζήτησης υψηλής επίδοσης. Πιο συγκεκριμένα, παρουσιάζουμε την *RCU-HTM*, μία τεχνική συγχρονισμού που συνδυάζει τις τεχνικές Read-Copy-Update (RCU) και Hardware Transactional Memory (HTM) και: α) υποστηρίζει την υλοποίηση ταυτόχρονης έκδοσης οποιουδήποτε τύπου δένδρου αναζήτησης, και β) επιτυγχάνει πολύ υψηλές επιδόσεις για ένα μεγάλο εύρος σεναρίων εκτέλεσης.

Στην *RCU-HTM* τα νήματα που τροποποιούν τη δομή του δένδρου με οποιοδήποτε τρόπο δουλεύουν σε αντίγραφα του τμήματος του δένδρου που επηρεάζουν. Μόλις το τοπικό τους αντίγραφο είναι έτοιμο, χρησιμοποιούν την HTM ώστε να επιβεβαιώσουν πως το μέρος του δένδρου που θα αντικατασταθεί δεν έχει στο μεταξύ τροποποιηθεί από κάποιο άλλο νήμα εκτέλεσης και, αν αυτό ισχύει, να αντικαταστήσουν το παλιό αντίγραφο με το τοπικό τους, το οποίο περιλαμβάνει τις κατάλληλες τροποποιήσεις.

Για να δείξουμε τις δυνατότητες της τεχνικής μας, υλοποιούμε και αξιολογούμε ένα σημαντικό αριθμό δένδρων αναζήτησης με χρήση του *RCU-HTM* και συγκρίνουμε την επίδοσή τους με ένα πλήθος ανταγωνιστικών ταυτόχρονων δένδρων. Πιο συγκεκριμένα, εφαρμόζουμε την τεχνική *RCU-HTM* σε 12 διαφορετικούς τύπους δυαδικών δένδρων, B+ δένδρων και (a-b)-δένδρων και συγκρίνουμε με πλήθος άλλων υλοποιήσεων που χρησιμοποιούν 4 διαφορετικούς μηχανισμούς συγχρονισμού, τα κλειδώματα, τις ατομικές λειτουργίες, το RCU και το HTM. Αξιολογούμε τα δένδρα αναζήτησης κάτω από πολλά διαφορετικά σενάρια εκτέλεσης μεταβάλλοντας το μέγεθος του κλειδιού που αποθηκεύεται στο δένδρο, τον αριθμό των κλειδιών που αποθηκεύονται στο δένδρο, το μείγμα από λειτουργίες που εκτελούνται καθώς και τον αριθμό των νημάτων που εκτελούν ταυτόχρονα λειτουργίες. Όλοι οι διαφορετικοί συνδυασμοί αυτών των παραμέτρων μας δίνουν 630 διαφορετικά σενάρια εκτέλεσης για κάθε δένδρο αναζήτησης. Επίσης, αξιολογούμε τα δένδρα χρησιμοποιώντας δύο μετροπρογράμματα που χρησιμοποιούνται κατα κόρον για την αξιολόγησης συστημάτων βάσεων δεδομένων, τα TPC-C και YCSB. Η αξιολόγηση μας δείχνει πως στην πλειονότητα των πειραμάτων τα δένδρα που χρησιμοποιούν το *RCU-HTM* έχουν υψηλότερες επιδόσεις από τους ανταγωνιστές τους, και ακόμα και στις ελάχιστες περιπτώσεις που δεν είναι τα καλύτερα, η επίδοσή τους είναι πολύ κοντά στην καλύτερη υλοποίηση. Αυτό, σε συνδυασμό με την ευκολία προγραμματισμού που προσφέρει η τεχνική *RCU-HTM* την καθιστούν την πρώτη τεχνική συγχρονισμού που μπορεί σχετικά εύκολα να εφαρμοστεί σε κάθε τύπου δένδρου αναζήτησης χωρίς να επηρεάζεται σε μεγάλο βαθμό η επίδοση τους.

Abstract

Concurrent search trees are one of the most popular and widely used family of data structures. They are used in applications where it is necessary to store a large volume of sorted data with the ability to efficiently search, insert, remove, as well as more advanced operations, such as range queries. Due to their importance, a large amount of research has led to many different types of search trees with different characteristics such as, for example, the max allowed length of a path of the tree. Each search tree provides different performance guarantees for each tree operation and each tree is chosen based on the needs of the specific application.

With the proliferation of multicores, where multiple threads execute concurrently and access shared data, concurrent data structures have become a critical component of parallel applications. In concurrent data structures it is necessary to coordinate the concurrent accesses by multiple threads in a way that guarantees the integrity of the data structure and the correctness of the operations. This coordination is achieved using some kind of synchronization mechanism such as, locks, hardware-provided atomic operations, Read-Copy-Update (RCU) and Transactional Memory (TM).

Despite the high amount of prior work, it still remains challenging to implement highly efficient concurrent search trees. This is mainly due to the fact that both traditional synchronization methods (i.e., locks and atomic operations) and more novel ones (i.e., Read-Copy-Update and Transactional Memory) fail to provide solutions that are generic and at the same time able to attain high performance under diverse execution scenarios.

Until recently, TM was mainly implemented in software and used through a library. However, recently two of the biggest processor manufacturers, Intel and IBM, have added support for Transactional Memory in the hardware level, allowing TM to be used without the large overheads imposed by the software implementations. In this work, we explore how HTM can be used to implement highly efficient concurrent search trees. More specifically, we present *RCU-HTM*, a synchronizatiom mechanism that combines RCU and HTM, and: a) supports the implementation of a concurrent version of any type of search tree, and b) achieves high performance across all execution scenarios.

In *RCU-HTM* threads that modify the tree structure in any way work in copies of the affected part of the tree. Once their local copy is ready, they use HTM to validate that the part of the tree that will be replaced has not been modified in the meanwhile and, if this is true, to replace the old part of the tree with their new modified version.

To showcase the capabilities of our technique, we implement and evaluate multiple *RCU-HTM* trees and compare their performance with several state-of-the-art competitors. More specifically, we apply *RCU-HTM* to 12 different types of binary, B+-trees and (a-b)-trees and compare against several state-of-the-art implementations that use 4 different synchronization mechanisms, namely locks, atomic operations, RCU, and HTM. We evaluate the trees under multiple different execution scenarios by varying the size of the keys stored in the tree, the size of the trees, the operations mix, and the number of threads, for a total of 630 execution scenarios for each implementation. We also evaluate the search trees using two well-known real-life benchmarks, namely TPCC and YCSB, which are widely used for the evaluation of database management systems. Our evaluation shows that in the majority of executions, *RCU-HTM* trees outperform their state-of-the-art alternatives, and even in the few cases where they do not, their performance is very close to that of the best performing implementation.

Ευχαριστίες

Με την ολοκλήρωση της συγγραφής της παρούσας διατριβής, ένα πολύ όμορφο ταξίδι εννέα ετών φτάνει στο τέλος του. Μέσα σε αυτά τα χρόνια έχω γνωρίσει, συναναστραφεί και συνεργαστεί με πάρα πολλά αξιόλογα άτομα καθένα εκ των οποίων με βοήθησε με τον τρόπο του να ανταπεξέλθω στις υψηλές απαιτήσεις της διεξαγωγής ενός διδακτορικού αλλά και να εξελιχθώ σαν ερευνητής αλλά κυρίως σαν άνθρωπος. Η συνύπαρξη και η καθημερινή συναναστροφή με αυτούς τους ανθρώπους ήταν που έκανε όλη αυτή την πορεία τόσο απολαυστική που δε θα την άλλαζα με τίποτα!

Αρχικά, θα ήθελα να ευχαριστήσω απο τα βάθη της καρδιάς μου τον επιβλέποντα καθηγητή μου, Γιώργο Γκούμα, για τη συνεχή καθοδήγηση του όλα αυτά τα χρόνια. Η ικανότητα του να μου δίνει συνεχώς κίνητρο για να συνεχίζω την πορεία μου προς την ολοκλήρωση του διδακτορικού ήταν καθοριστική. Ήταν πάντα εκεί για να προσφέρει λύσεις στα αδιέξοδα που προέκυπταν και να μου δίνει ώθηση να συνεχίζω, σε περιόδους που η έρευνά μου φαινόταν να βρίσκεται σε τέλμα. Ένα τεράστιο ευχαριστώ οφείλω στον μεταδιδακτορικό ερευνητή Κωστή Νίκα. Η όρεξη και η διάθεσή του να βοηθήσει, είτε μέσα απο πολύωρες τεχνικές και μη συζητήσεις, είτε γράφοντας ο ίδιος κείμενο, είναι που τον κάνουν ξεχωριστό. Δε θα ξεχάσω ποτέ τα ξενύχτια που έριξε ώστε να προλάβουμε να υποβάλλουμε εντός της διορίας εργασίες σε συνέδρια.

Θερμές ευχαριστίες οφείλω επίσης στον κύριο Νεκτάριο Κοζύρη ο οποίος από τα προπτυχιακά μου χρόνια μου μετέδιδε την απίστευτη ενέργεια και αγάπη του για το αντικείμενο της επιστήμης των υπολογιστών. Ακόμα, τον ευχαριστώ επειδή μου έδωσε την ευκαιρία να γίνω μέλος του εργαστηρίου υπολογιστικών συστημάτων. Ευχαριστώ θερμά τα υπόλοιπα μέλη της επταμελούς μου επιτροπής, τον κύριο Παναγιώτη Τσανάκα, τον κύριο Διονύση Πνευματικάτο, την κυρία Παναγιώτα Φατούρου, τον κύριο Κωνσταντίνο Σαγώνα καθώς και τον κύριο Χρήστο Κοτσελίδη για τον χρόνο τους και για τα πολύτιμα σχόλια τους σχετικά με την εργασία μου.

Στο εργαστήριο υπολογιστικών συστημάτων, όλα αυτά τα χρόνια, γνώρισα πάρα πολλά αξιόλογα άτομα και τους ευχαριστώ όλους έναν προς έναν για όλες τις στιγμές που ζήσαμε. Όλοι συνεισφέρανε ώστε να μετατρέψουν τα χρόνια του διδακτορικού μου απο μία επίπονη διαδικασία σε ένα απολαυστικό ταξίδι που θα μου μείνει για πάντα αξέχαστο. Ευχαριστώ ιδιαιτέρως τα παιδιά με τα οποία ήρθα πιο κοντά, την παρέα των ηλεκτρολόγων του Ταρτούφου, τον Αντρέα, τον Δημήτρη, την Ιωάννα, τον Κωστή, την Νικέλα, τον Ορέστη, τον Παναγιώτη, τον Στράτο και την Χλόη. Δεν θα μπορούσα να φανταστώ καλύτερη παρέα για να περάσω αυτά τα χρόνια. Οι ατελείωτες συζητήσεις επι παντός επιστητού και τα αμέτρητα γέλια μας ήταν η καλύτερη αποφόρτιση όταν τα πράγματα δυσκόλευαν. Επίσης, ευχαριστώ ιδιαίτερα τον Τάσο Νάνο ο οποίος ήταν πάντα εκεί, όποτε τον χρειαζόμουν, για να βοηθήσει είτε σε τεχνικά ζητήματα είτε σε προσωπικό επίπεδο.

Ακόμα, θα ήθελα να ευχαριστήσω τα παιδιά με τα οποία συνυπάρξαμε στην διαχειριστική ομάδα του εργαστηρίου. Συγκεκριμένα, τους Βαγγέλη Αγγέλου, Στέφανο Γεράγγελο, Γιάννη Γιαννακόπουλο, Κωστή Παπαζαφειρόπουλο, Γιάννη Παπαδάκη και Χρήστο Κατσακιώρη. Μας συνέδεσε η αγάπη μας για τους υπολογιστές και η επιθυμία μας να δουλεύουν τα μηχανήματα του εργαστηρίου όπως πρέπει. Δε θα ξεχάσω ποτέ τις ατελείωτες τεχνικές συζητήσεις και διαφωνίες μας, τις δύσκολες νύχτες που έπεφταν τα ρεύματα και έπρεπε να ανέβουμε στο εργαστήριο για να τα επαναφέρουμε και όλα όσα ζήσαμε σαν ομάδα αυτά τα χρόνια.

Ευχαριστώ επίσης τα παιδιά τα οποία έκαναν τη διπλωματική τους μαζί μου, τον Χαράλαμπο Στυλιανόπουλο, την Χριστίνα Γιαννούλα, τον Κωνσταντίνο Καζατζή, τον Παναγιώτη Μπίλλη, τον Μάριο Καρδάρα, τον Σωτήρη Δραγώνα και τον Γιώργο Δημητρακόπουλο. Η συνεργασία με τον κάθε έναν ήταν εξαιρετική και η δουλειά που έκαναν με βοήθησε πάρα πολύ για την ολοκλήρωση του διδακτορικού μου.

Θα ήθελα να ευχαριστήσω και κάποιους πολύ σημαντικούς ανθρώπους στη ζωή μου. Ευχαριστώ ιδιαίτερα τον Βασίλη Μπούγλα, ο οποίος ήταν ο άνθρωπος που μου μετέδωσε την αγάπη για τους υπολογιστές, όταν ακόμα ήμουν μικρός και αυτός εργαζόταν στο γραφείο του πατέρα μου, και με έκανε να θέλω να ασχοληθώ με αυτό το αντικείμενο. Τον ευχαριστώ πάρα πολύ για όλο τον χρόνο που αφιέρωσε τότε για να μου δείξει πως δουλεύει ένας υπολογιστής, να παίζει μαζί μου παιχνίδια στον υπολογιστή του γραφείου και να επισκευάζει τους υπολογιστές που είχα στο σπίτι. Επίσης, ευχαριστώ πάρα πολύ τους φίλους μου, και την κοπέλα μου, οι οποίοι αποτελούν ένα σημαντικό κομμάτι της ζωής μου και την κάνουν πολύ πιο όμορφη.

Τέλος, δε θα μπορούσα να μην ευχαριστήσω την οικογένειά μου. Τις δύο μου αδερφές, Κωνσταντίνα και Κυριακή, οι οποίες είναι πάντα δίπλα μου και με στηρίζουν σε όλες μου τις δυσκολίες. Απεριόριστη ευγνωμοσύνη οφείλω στους δύο μου γονείς, Δημήτρη και Μελπομένη, στους οποίους χρωστάω τα πάντα. Τους ευχαριστώ για τη στήριξή τους, την απεριόριστη αγάπη που μου έχουν δώσει και την υπομονή τους.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Motivation

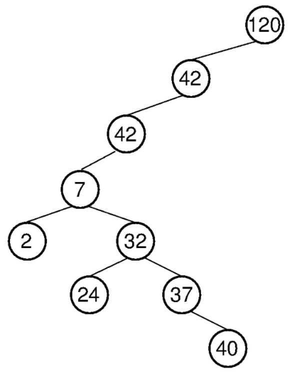
As Steven Skiena states in his book[1], the dictionary abstract data type (ADT) is one of the most important data structures in computer science and search trees are the most common data structures used for its implementation. A dictionary, also known as *map* or *associative array*[2], stores key-value pairs and supports four operations, namely, *lookup(key)*, *insert(key, value)*, *delete(key)* and *range_query(key1, key2)*.

With the proliferation of multi-core systems the need for concurrent data structures has become even more intense. Related research around simple data structures such as linked lists, hash tables, etc, has resulted in several efficient implementations that scale well for high numbers of threads. In such simple data structures, whose operations typically involve a very limited number of modifications, it is relatively easy to apply fine-grained synchronization schemes, using either locks or hardware-provided atomic operations, and allow multiple threads to efficiently access the data structure concurrently.

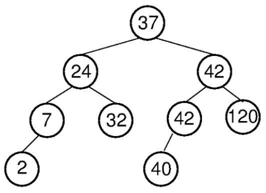
On the contrary, search trees may need to support complex operations such as rebalancing the tree and replacing a node in the high levels of the tree with one that lays in the last level of the tree. These operations may affect a large number of tree nodes and make the application of fine-grained synchronization challenging. For this, proposed fine-grained approaches do not

[1] *The abstract data type "dictionary" is one of the most important structures in computer science. Dozens of data structures have been proposed for implementing dictionaries, including hash tables, skip lists, and balanced/unbalanced binary search trees. This means that choosing the best one can be tricky.*, The Algorithm Design Manual, by Steven S. Skiena

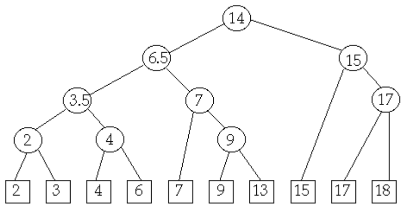
[2] `https://en.wikipedia.org/wiki/Associative_array`



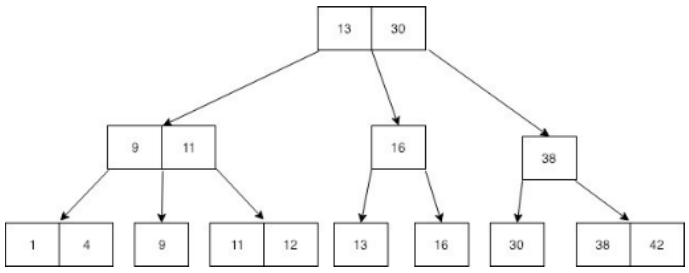
(a) Unbalanced internal binary search tree



(b) Balanced internal binary search tree



(c) Balanced external binary search tree



(d) Balanced B+-tree

Figure 1.1: Different types of search trees.

support these operations and resort to trees that are unbalanced or relaxed-balanced and external (i.e., store all the data in the last levels of the tree) or partially-external (i.e., mark nodes instead of physically removing them). On the other hand, coarse-grained synchronization mechanisms, such as locking the whole tree or using transactional memory, which can easily support balanced and internal trees, lead to excessive serialization of tree operations even when these operations modify different parts of the tree. In this work we propose a synchronization mechanism, called *RCU-HTM*, which closes the gap between coarse-grained and fine-grained synchronization mechanisms and provides applicablity (i.e., can be applied to any search tree) along with high performance across a wide variety of execution scenarios.

1.1 Serial Search Trees: no one-size-fits-all

Search trees come in many different flavors depending on the number of keys stored in each tree node (i.e., binary trees and (a-b)-trees), the balancing guarantees (i.e., unbalanced, relaxed-balanced and balanced) and the way data is stored in the nodes of the tree (i.e., internal, partially-external and external). Four examples of different search trees are shown in Figure 1.1. Apart from the above three basic parameters that categorize a search tree, there can be other characteristics that also differentiate one from another. Some examples include: the splay tree which keeps the most commonly accessed elements on the top levels of the tree for faster acquisition [ST85]; the treap, a combination of a tree and heap where each node has a weight and the nodes with the largest weights are placed at the top levels of the tree [AS89].

All these different variations of search trees, along with many more, make it hard for the designers of applications to choose the appropriate search tree for their use case. Each search tree is appropriate for some workloads but may not be a good choice for another. We validate this with a simple set of experiments with Figure 1.2 presenting the results. We evaluate 9 different search trees under 90 execution scenarios with different key sizes, number of keys in the tree and operation mixes. For every scenario we normalize the throughput of all trees to the throughput of the best performing tree. As the figure shows, all trees have high variance and can be the best in some cases while in others are up to 15% close to the best execution.

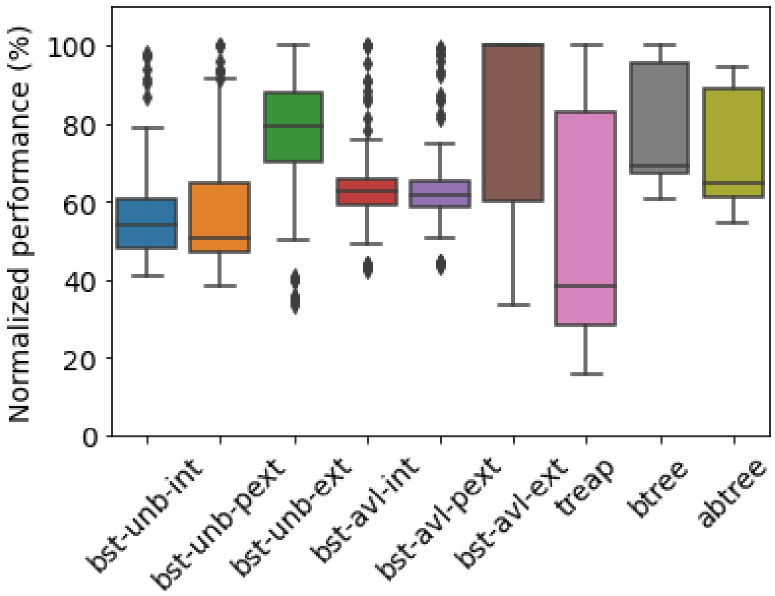


Figure 1.2: Normalized performance (compared to the best in each case) of 9 serial search trees in a single-threaded execution. For each search tree we have 90 different execution scenarios.

1.2 Concurrent Search Trees: making things even worse

In multi-threaded environments we need to implement concurrent search trees, i.e., trees that allow multiple concurrent threads to execute operations on them without compromising the integrity of the data structure. This adds an extra parameter on the different possible execution scenarios, i.e., the number of threads, which significantly increases the difficulty of choosing the appropriate search tree for a specific application. Even worse, the design and implementation spectrum of concurrent search trees is much larger than the serial one since we now also have to choose between several different synchronization mechanisms. When a programmer needs to find the best concurrent search tree candidate for an application he/she has the following choices regarding the synchronization mechanism to use:

- **Locks (lock-based trees).** Locks can be applied on search trees in coarse-grained, medium-grained or fine-grained way depending on the size of the parts of the tree a single lock may protect. In coarse-grained implementations we only have a single global lock to protect the whole tree. Coarse-grained locking is trivial to implement for any type of search tree, however, the excessive serialization of operations does not allow concurrent threads to exploit the available parallelism that trees provide through their multiple disjoint tree paths. Contention-adaptive locking [SW15] is a medium-grained application of locks, in which a baseline serial data structure is used and is split accordingly when high levels of contention are observed. Contention-adaptive trees allow multiple threads to work concurrently on disjoint paths of the tree but they have two disadvantages. First, they require from the underlying serial data structures to support join and split operations, which for some types of search trees is difficult to implement efficiently. For example, in balanced trees, such as AVL, it is very challenging to join and split trees while at the same time maintaining their balancing guarantees. Second, contention-adaptive trees require an additional tree structure on top of the baseline search trees, which in high contention cases induces high traversal overheads. State-of-the-art lock-based search trees [BCCO10, DVY14, CGR13] use *fine-grained* locking schemes where each tree node is protected by a different lock. While these approaches allow for maximum concurrency between threads, which typically translates to higher performance, they are hard to design and implement because the programmer has to manually add code to lock/unlock a large number of locks per operation. This complexity is the main reason why these trees do not support complex tree operations such as rebalancing.

- **Hardware-provided atomic operations (lock-free trees).** Lock-free search trees exploit the hardware-provided atomic instructions, such as Compare-and-Swap (CAS) and Load-Linked-Store-Conditional (LL/SC), either directly [EFRvB10, HJ12, NM14, CDT14, RM15] or indirectly with higher level primitives, such as Double-Compare-Single-Swap (DCSS) and Load-Linked-Store-Conditional-Extended (LLX/SCX), both implemented on top of CAS [Bro17, BPA20]. The first category provides very high performance but the direct use of low-level atomic primitives makes it hard to support complex operations such as rebalancing. When using higher level primitives more complex operations can be supported, such as local rebalancing steps in relaxed-balanced trees, but each operation is translated in multiple CAS instructions, thus inducing severe overheads.

- **Read-Copy-Update (RCU-based trees).** In RCU-based search trees threads that need to modify the tree first create a copy of the affected subtree and then replace its old version with their new modified one. This replacement is performed in a single atomic step and this allows reader threads to traverse the tree without using any kind of synchronization. However, updaters still need to be synchronized in some way, and this is done using either coarse-grained locking or fine-grained locking. In the fine-grained locking approach [HW14] updaters use a single lock and only one of them can access the data structure at any time. This design is simple to implement but serializes all operations that modify the tree. A fine-grained locking scheme can be used along with RCU [AA14] to allow for concurrent updates, but this complicates the implementation of complex tree operations such as rebalancing.
- **Transactional Memory (TM-based trees).** Transactional Memory is either implemented in software as a library (STM) or provided by the hardware (HTM). In this work we only use HTM since STM implementations typically induce very high runtime overheads [CBM+08]. HTM can be applied on search trees in either a coarse-grained or a fine-grained way. In the first case, tree operations are simply enclosed in an HTM transaction in a way similar to the coarse-grained locking approach. In contrast to locking, HTM optimistically executes concurrent operations, and serializes them only when they conflict with each other. This serialization is achieved using a non-transactional fallback path, that is executed when a transaction has repeatedly aborted for a number of times, in which a global lock is acquired causing all concurrent operations to halt. This coarse-grained HTM approach is easy to implement and provides high performance under certain circumstances. However, such coarse-grained transactions are large, both in terms of memory size and time duration, and are thus prone to transactional aborts. Brown et. al [Bro17] have proposed the 3-path HTM approach, which, instead of resorting to a global lock upon repetitive transactional aborts, uses a middle transactional path and a lock-free non-transactional fallback path. The middle and fallback paths can execute concurrently, thus avoiding the serialization of operations that occurs in the classic 2-path coarse-grained approach. Consistency-Oblivious programming (COP) [AAS11, AK14] uses smaller transactions by splitting the

tree operations in two parts: a read-only prefix and an updating suffix. Only the updating suffix is enclosed inside a transaction. Although COP indeed reduces the transaction size relatively to the coarse-grained HTM synchronization, it still has two drawbacks; first, lookup operations need to use transactions and hence pay their overhead, and second, the whole update operation is enclosed in a single transaction, leading to a large transactional write set which results in a high probability of conflicts.

All these different synchronization mechanisms cause the design and implementation space of search trees to expand significantly. However, things get more complicated from the fact that not all synchronization mechanisms are actually applicable to all serial search trees. For example, fine-grained mechanisms such as fine-grained locking and CAS-based lock-free can not be applied to strictly balanced search trees, because the rebalancing operations require multiple nodes to be modified in a single atomic step. This limitation can lead to contradicting situations, where a serial search tree works well for a serial application but when parallelizing this application the appropriate synchronization mechanism may not be applicable to this search tree.

Figure 1.3 presents how close each concurrent search tree performs to the best performing tree for the different execution scenarios. Each boxplot includes 630 different execution scenarios for each search tree. The results validate that no search tree provides sustainably high performance across all the different execution scenarios as there is large variation for every single one. It is thus not possible to choose a single best implementation.

1.3 The programmability-performance tradeoff

As explained, choosing the right synchronization mechanism for a given workload is not an easy task. Another property to consider is the programmability effort required for each synchronization mechanism.

Coarse-grained synchronization is trivial to implement; each operation is simply enclosed inside a critical section and the programmer does not need to know the implementation details of the data structure. Coarse-grained synchronization can thus be implemented on top of any serial data structure. However its downside is the excessive serialization of operations as only one thread can be in the critical section at any time. Even if reader-writer locks are used, the serialization of the writers induces very high overheads. According to Amdhal's law[3], this serialization puts an upper limit on the performance of the data structure; when a large part of the application time is spent on accesses on the data structure, coarse-grained implementations become the bottleneck for the whole application. In general, coarse-grained strategies are easily

[3]`https://en.wikipedia.org/wiki/Amdahl's_law`

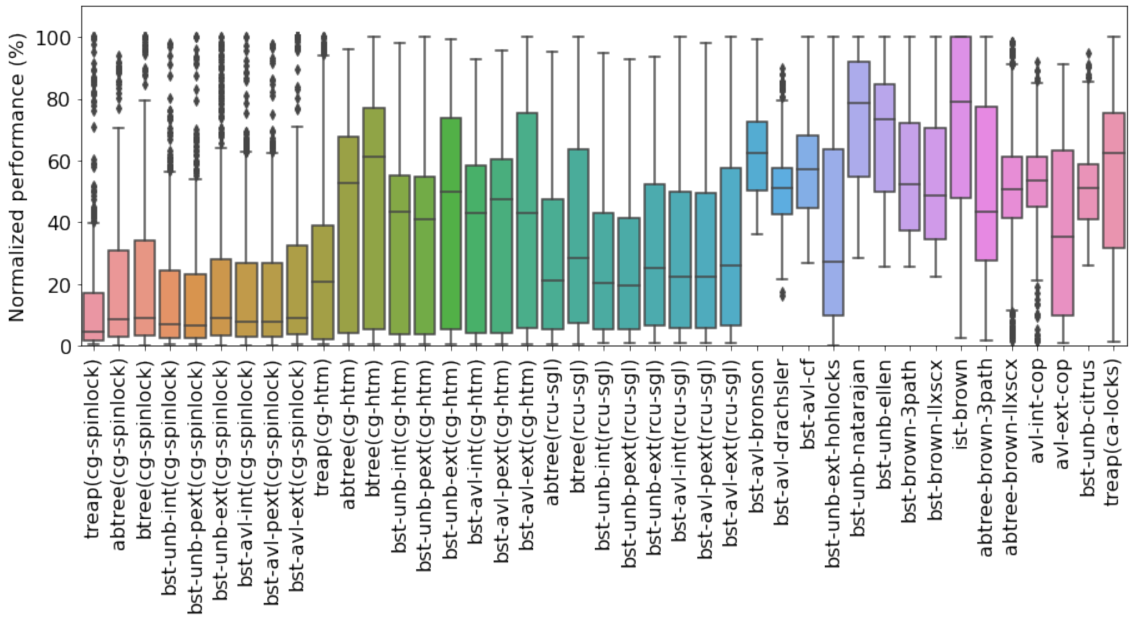


Figure 1.3: Performance of concurrent search trees in a multi-threaded execution. For each search tree we have 90 different execution scenarios.

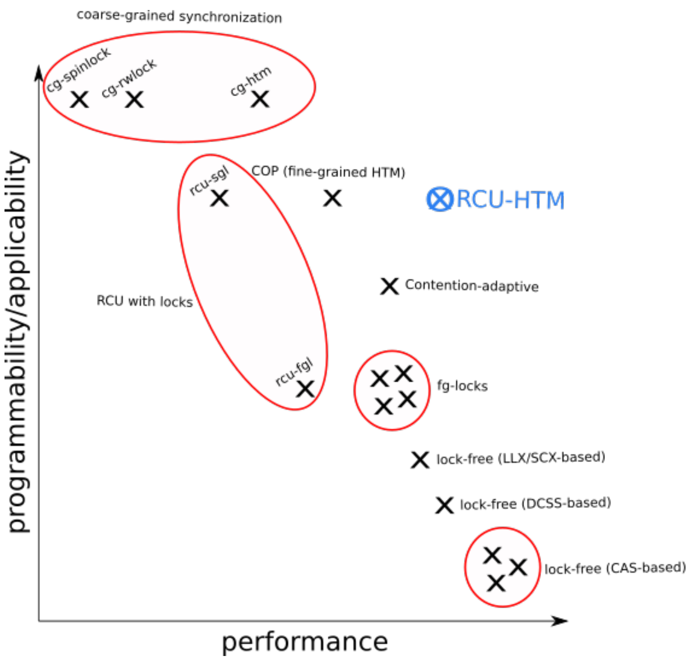


Figure 1.4: Qualitative assesment of the programmability-performance tradeoff for concurrent search trees.

applicable but typically fail to exploit concurrency. Only locks and HTM can be applied in such a coarse-grained way.

On the other hand, fine-grained synchronization enables higher parallelism, thus achieving higher performance, but is commonly tailored to a specific version of a data structure. For less complex data structures than search trees, such as linked lists and hash tables, fine-grained strategies are able to provide high performance with relatively simple and generic implementations. Unfortunately, this is not the case for complex data structures such as search trees, mainly for two reasons. First, for some search trees, an operation may need to modify multiple memory locations, a fact that greatly complicates fine-grained approaches. Second, the large variety of different search tree implementations make it challenging to come up with an approach generic enough to be applicable on all these versions.

This creates a programmability-performance tradeoff. Figure 1.4 presents the different synchronization mechanisms that can be currently leveraged for concurrent search trees and where they stand in the programmability-performance spectrum. We should note here that programmability is too hard to quantify, so we qualitatively assess it based on our experience with understanding and implementing these methods. It is not our intention to strictly rate each method, but rather show the greater picture and the pattern that emerges.

Coarse-grained synchronization lies at the top left corner of the spectrum, being the most straightforward to apply on any serial search tree with absolutely no knowledge of its internal details. Coarse-grained locking, however, is the worst performance-wise even when reader-writer locks are used. HTM allows some degree of parallelism by allowing multiple transactions to execute concurrently and only serializes them in case of conflicts. The problem with coarse-grained HTM is that tree operations typically read and modify several memory locations, making the occurring transactions large both in terms of memory space and time duration; this leads to large number of transactional aborts and low performance.

Moving to the right of the spectrum, the synchronization mechanisms become more fine-grained, requiring more programming effort but providing higher performance. We first encounter techniques that combine RCU with locks, namely *rcu-sgl* and *rcu-fgl*. In *rcu-sgl* updaters are serialized using a single lock, which, similarly to coarse-grained locking, leads to poor performance. In *rcu-fgl*, updaters are synchronized using fine-grained per-node locks. Consistency-oblivious programming (COP) uses HTM in a fine-grained way. Rather than enclosing the whole tree operation in a transaction, it splits the operation in three steps; traversal, validation of the traversal and modifications phase. Only the last two phases need to be enclosed in an HTM transaction and while this gives some performance benefits, it still requires large transactions.

Fine-grained locking and lock-free implementation lie at the bottom right end of the spectrum. Fine-grained locking techniques use one lock per tree node. This greatly complicates the design and implementation of these trees, because the programmer is required to get into the details of the tree operations and add the logic for acquiring and releasing all these locks. Regarding lock-free trees, there are two categories based on the atomic primitives that are used; those that directly exploit the low level atomic instructions, such as CAS, and those that use higher level primitives, such as Double-compare-single-swap (DCSS) [HFP03] and load-linked-store-conditional-extended (LLX/SCX) [BER13]. The first category is the most difficult to design and implement. This is due to the fact that hardware-provided atomic instructions can only work on one, or at most two consecutive, memory locations. It is thus very difficult to implement advanced operations of trees, as for example, the rebalancing of a tree. Double-compare-single-swap (DCSS) [HFP03] and load-linked-store-conditional-extended (LLX/SCX) [BER13] are implemented on top of CAS and provide a more flexible way to the programmers to perform atomic operations that include more than one memory locations. These two techniques facilitate the design and implementation of a lock-free concurrent search tree but they incur very high overheads.

1.4 Our approach: *RCU-HTM*

This thesis focuses on concurrent search trees and aims to improve the programmability-performance tradeoff. More specifically, we propose, implement and evaluate a novel synchronization technique that targets the top right corner of the programmability-performance spectrum. We call this technique *RCU-HTM*, since it is a combination of RCU and HTM. *RCU-HTM* can be applied to any kind of search tree and provides high and robust performance. It leverages both RCU and HTM to implement concurrent search trees that exhibit the benefits of both schemes, namely *asynchronized read-only operations* and *optimistic concurrent execution of updaters*. To demonstrate the gains of combining these two synchronization mechanisms, we compare in Figure 1.5 the performance of an *RCU-HTM* internal AVL tree (avl-int-rcu-htm) with the two coarse-grained implementations that use RCU with coarse-grained locking (avl-int-rcu-sgl) and coarse-grained HTM (avl-int-cg-htm). In the left, read-only case the three implementations achieve similar performance. The advantages of *RCU-HTM* become evident when update operations are performed. Even when only 20% of operations are updates, the performance of both RCU and HTM drops for different reasons; RCU serializes updaters with a global lock and HTM encounters a high number of conflicts. On the other hand, *RCU-HTM* maintains its high performance on all three contention levels.

In a nutshell, *RCU-HTM* exploits RCU and HTM in the following way:

- **RCU:** Readers enjoy the properties of RCU-based implementations, that is, they are completely asynchronized and unaffected by concurrent updates. This is achieved by having updaters perform their modifications on copies of the affected nodes rather than directly on the shared

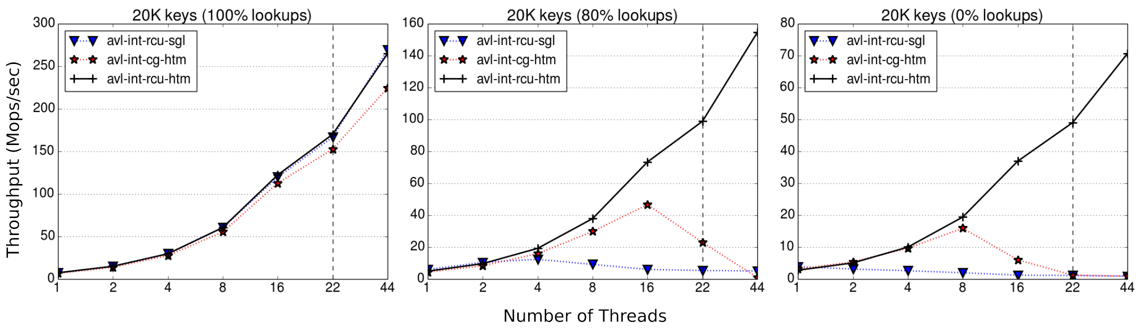


Figure 1.5: Comparison of HTM-based, RCU-based and *RCU-HTM* based internal AVL trees.

tree. The copies are then atomically installed in the tree by modifying a single node's child pointer.

- **HTM:** Updaters enjoy the properties of HTM-based implementations, that is, they are optimistically executed concurrently and serialize only when conflicts are present. To exploit HTM, we modify the RCU-based coarse-grained implementation in the following ways: first, we augment the traversal phase to store the state of the traversed nodes as we need it to validate later that no other updater has modified them in the meanwhile; second, when the copy is ready to be installed in the shared tree, we execute an HTM transaction where we atomically perform two steps: 1. we validate that the nodes to be replaced have not been modified since they were read, and, 2. we install the copy in the tree.

The contributions of this thesis are the following:

- We create a library of concurrent search trees which contains several C and C++ implementations and can be used through a well-defined API. This is, to the best of our knowledge, the most complete library of concurrent search trees since it contains a very large number of implementations both from related research papers and the ones that are presented in this thesis. The library is publicly available at the author's github page[4].

- We propose, implement and evaluate the *RCU-HTM* synchronization technique describing all the necessary details to implement an *RCU-HTM* search tree. We use an internal AVL tree and a B+-tree as examples, but we provide all the necessary guidelines for the same procedure to be used for any type of search tree.

- We implement *RCU-HTM* versions of 12 types of search trees, showcasing the applicability of the *RCU-HTM* technique. We implement 3 unbalanced binary search trees (internal, partially-external and external), 6 balanced binary search trees (internal, partially-external

[4]`https://www.github.com/jimsiak/concurrent-maps-cpp`

and external versions of both AVL and Red-Black trees), 1 relaxed-balanced partially-external AVL tree, a B+-tree (i.e., the external version of a B-tree) and an (a-b)-tree (i.e., a relaxed-balanced variant of a B+-tree).

- We evaluate *RCU-HTM* search trees and compare their performance with several state-of-the-art implementations that use 4 different synchronization methods, namely, locks, atomic operations, RCU and HTM. Our evaluation includes three benchmarks, one artificial microbenchmark and two real-life benchmarks, namely, TPC-C and YCSB. Our experimental results show that *RCU-HTM* is able to provide sustainable performance under various execution scenarios.

- We incorporate DEBRA, a state-of-the-art epoch-based memory reclamation scheme [Bro15], in the *RCU-HTM* trees to make them practical to use in large-scale, long-running applications. We then evaluate the overheads induced by this memory reclamation scheme, and show that *RCU-HTM* based trees are able to maintain their high performance even when reclamation is performed.

- We implement range query operations for the *RCU-HTM* based trees. Range queries are very popular operations for the map ADT especially when they are used as indices in database management systems.

1.5 Structure of the thesis

The rest of this thesis is structured in the following way; Chapter 2 provides all the necessary background information and Chapter 3 performs an overview of the state-of-the-art approaches that currently exist for concurrent search trees. In Chapter 4, we analyze the proposed synchronization mechanism, namely *RCU-HTM* and in Chapter 5 we present the experimental evaluation results. In Chapter 6, we explain how we can add support for efficient range query operations in *RCU-HTM*. Finally, Chapter 7 summarizes the findings of the thesis along with some future extensions of the work.

CHAPTER 2

Background

In this chapter we provide the necessary background information. We first present the interface provided by our search tree implementations and then explain some basic concepts around search trees. Finally, we explain how we use the hardware transactional memory (HTM) instructions that are provided on the Intel processors we used in our experiments.

2.1 The *Map* Interface

As mentioned in Chapter 1, we use search trees to implement the *map* ADT which stores key-value pairs and supports insertions, deletions, lookups and range queries on these pairs. In our library a *map* is represented by the *Map* class that is shown in Listing 2.1. All search trees inherit this base class and implement their own specific versions of the four methods. In this thesis we mainly focus on the *lookup()*, *insert()* and *remove()* operations and dedicate Chapter 6 to the efficient implementation of range queries with *RCU-HTM*. *Lookup()* searches for the given key and returns either the value associated with it, in case the key is found, or a special value object called *NO_VALUE*. *Insert()* adds the given key-value pair in the *map* if the key is not already present; otherwise, *NO_VALUE* is returned to indicate the the new key-value pair was inserted. *Remove()* removes the given key and its associated value from the *map* and returns its value; if the key was not found in the *map*, *NO_VALUE* is returned.

```cpp
template <typename K, typename V>
class Map {
public:
  V lookup(K& key);
  V insert(K& key, V& val);
  V remove(K& key);
  int rangeQuery(K& lo, K& hi, vector<pair<K,V>> kv_pairs);
};
```

Listing 2.1: The *Map* interface.

2.2 Search Trees

The most popular data structures to implement maps are search trees, such as binary search trees, B-trees, (a-b)-trees, radix trees, splay trees, and others. Each tree has different characteristics and thus is a better candidate for different execution scenarios. Nevertheless, they all share some basic characteristics presented in this section. These common characteristics allow us to devise a synchronization mechanism that can be applied to any type of search tree. More specifically, in all search trees each operation is split into two phases:

- **Traversal phase**: A *read-only* phase in which a set of nodes is being traversed until the appropriate node is reached. This set of nodes is called the *access path* of the operation. In a range query operation the traversal phase may need to extend to more nodes.
- **Modification phase**: A *read-write* phase in which the tree is modified according to the given operation, i.e., a key-value pair is inserted or removed from the tree. This phase also includes the rebalancing of the tree in the case of a balanced search tree, such as AVL or a B-tree. Read-only operations, such as lookups and range queries, do not include this modification phase.

Algorithm 1 shows the generic structure of an update operation, i.e., an insertion or a removal of a key-value pair, in a serial search tree, and Figure 2.1 depicts the execution of *update_and_rebalance()* of an internal AVL tree. While some details of the implementation may vary for different search trees (e.g., AVL, B+-trees, balanced/unbalanced), the general procedure remains the same and follows the two phases discussed before. During the traversal we store the access path in the *accpath* array with *top* indexing the last node accessed to enable the reverse traversal of the tree if rebalance needs to be performed.

The most complex operations that can be encountered in a search tree are the following:

1. **Balancing the tree.** The perfomance of search tree operations depends highly on that of the traversal phase [DGT15], which in turn depends on the length of the access path. In *unbalanced* trees, access paths can become too long, leading to poor performance. *Balanced* trees, such as AVL, Red-Black and B+-trees, eliminate such long paths by rebalancing the

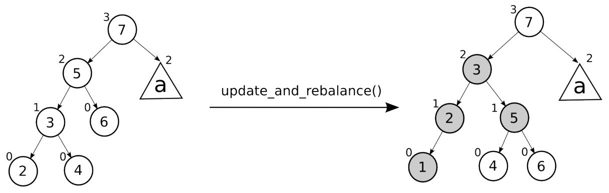


Figure 2.1: An example insertion of key 1 followed by the rebalance phase in a serial internal AVL tree.

ALGORITHM 1: Update operations in serial search trees.

```
int st_update_seq (st *st, K& key, V& value)
    // Traverse the access path.
    st_traverse(st, key, &accpath, &top);
    // Returns 0 if tree has not been modified, 1 otherwise.
    ret = update_and_rebalance(st, key, value, accpath, top);
    return ret;
```

tree, when necessary, after insertions and deletions. Rebalancing is a complex operation because it modifies several nodes and in a concurrent search tree exclusive access to all of them needs to be granted prior to modifying them. *Relaxed-balanced* trees stand in the middle between unbalanced and balanced. In this case, the rebalancing is split in multiple local steps which can be performed independently. This decreases the complexity of the rebalancing operation and the synchronization overheads, but long traversed paths can still be observed, that negatively affect the performance of the search tree.

2. **Removing an internal node.** To remove an internal node (e.g., a node with two children in the case of a binary search tree) from an *internal* search tree, we first need to replace its key with the key of its successor node, that is the node that contains the first key that is larger than the key of this node. After this replacement, we remove the successor node from the tree. While this kind of internal node removal is common for serial search trees, in a concurrent search tree extra care needs to be taken. Otherwise traversals that search for the successor's key may be led astray to the previous position of the successor instead of the new one. To avoid such incorrect executions, traversals need to synchronize with concurrent updaters. *External* trees overcome this complexity by storing the actual data on external nodes. This way deletions always take place at the lowest levels of the tree, thus requiring less synchronization. External trees, however, have two drawbacks; first, they require more nodes, i.e., twice the nodes of an internal tree, and second, the traversal paths are longer due to traversing the routing nodes. *Partially-external* trees [BCCO10] require less nodes than external ones but still more than internal ones. In these trees, internal

nodes are not physically removed from the tree but are only marked as deleted. Deleted nodes are still traversed, thus increasing the length of the access paths.

All the different combinations of the above search trees, combined with the various structures of tree nodes (e.g., B+-trees versus binary trees) result in a large design and implementation space for search trees. Different trees are appropriate for different execution scenarios. As discussed in Chapter 1, for concurrent search trees things get more complicated since fine-grained synchronization schemes, which exploit parallelism, can not be applied to all search trees while coarse-grained schemes are generic but provide poor performance. In many cases this makes it extremelly difficult to choose the appropriate type of search tree for a specific application and combine it with a high performing synchronization scheme. Our *RCU-HTM* technique moves the synchronization scheme out of the decision path, as it can be applied to any search tree while maintaining high performance under all the workloads of our evaluation methodology.

2.3 Transactional Memory (TM)

Transactional Memory (TM) [HM93] is a synchronization mechanism based on the idea of transactions in database management systems (DBMS). A transaction is a block of code that is guaranteed to execute atomically. In the context of a DBMS, a transaction accesses database records, while in the context of low-level synchronization a TM transaction accesses memory locations. The main goal of TM is to remove the burden of synchronization from the programmer and delegate it to the TM system, which can be either implemented as a software library (STM) or integrated in hardware (HTM).

Since 2013, most research has focused around STM and HTM implementations integrated in simulators. Although STM approaches [ST97, HLMS03, HLM06] have provided some encouraging results, they typically incur very high overheads compared to the corresponding serial data structures severely harming their popularity. The overheads mainly arise from the need to track each and every memory access and maintain the read- and write-sets of the transactions. Simulator-based HTM approaches on the other hand [TPK$^+$09, FSBA11] have shown very promising results, but the lack of support on real hardware discouraged further research efforts. This changed when Intel and IBM released processors with HTM support. Intel added their Transactional Synchronization eXtensions (TSX) in processors based on the Haswell architecture and all their successors. Similarly, IBM's Power8 provides support for assembly instructions which enable HTM.

The experiments of this thesis have been executed on an Intel Broadwell processor which supports TSX and provides the following assembly instructions to manage hardware transactions [1]:

- *XBEGIN*: Starts an HTM transaction. This instruction returns a status code which indicates whether a transaction has just started or a transaction has been aborted, as when a transaction aborts, the execution flow returns to the point where *XBEGIN* was called.
- *XEND*: Commits an HTM transaction.
- *XABORT*: Explicitly aborts a transaction. A representative code can be passed to the abort instruction to enable the distinction among different abort reasons.
- *XTEST*: Returns true or false depending on whether the code currently executes in transactional mode or not.

The basic TM characteristics of the TSX implementation are the following:

- *Lazy data versioning:* TSX uses lazy versioning. All memory writes performed inside a transaction become visible to other threads only after the transaction successfully commits.
- *Eager conflict detection:* Upon the detection of a conflict the transaction immediately aborts.
- *Cache line granularity:* Conflicting operations are detected at a cache line granularity. This can result in false conflicts when concurrent threads modify disjoint parts of a cache line.
- *Strong isolation:* TSX provides strong isolation, meaning that a conflict is detected even if the conflicting access occurs in non-transactional code.
- *Best-effort:* TSX is a best-effort HTM. This means that no forward progress is guaranteed using only transactional mode, and a transaction may always fail to commit. It is therefore necessary that the programmer implements a non-transactional fallback path.

In general, a transaction may fail to commit (abort) for various reasons including:

- *Data conflict:* When another thread, executing in transactional or non-transactional mode, writes to a memory location that belongs to the transaction's read or write set.
- *Capacity abort:* When the transaction's footprint has exceeded the size of the transactional buffers that are used to store the read- and write- set of an HTM transaction. Table 2.1 presents their size for the Broadwell processor used in our experiments.

[1]IBM's HTM implementation is very similar

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

Table 2.1: The size of the transactional buffers provided in our Broadwell server.

- *Explicit abort:* When the programmer explicitly aborts the transaction.
- *Other:* A transaction may abort due to several other reasons including interrupts, unsupported instructions, system calls etc.

CHAPTER 3

Concurrent Search Trees: State-of-the-art

In this chapter we provide an overview of the current state of concurrent search trees. The aim of this chapter is to summarize all the different synchronization mechanisms that can be applied on search trees and discuss the advantages, disadvantages, limitations and the implementation challenges of each mechanism. In the following sections we provide snippets of C++ code that facilitate the understanding of our main points. The reader can access all implementations in our library's github repository [1].

3.1 Lock-based Search Trees

3.1.1 Coarse-grained Locking

Coarse-grained locking is the easiest approach to allow concurrent access to search trees. The programmer needs only to enclose each one of the four operations inside a critical section protected by a single lock. There is no need for the programmer to have any knowledge of the internal implementation details and can use the serial search tree as a black box. Listing 3.1 shows the C++ class that wraps a serial search tree with coarse-grained locking. The same code is used for the coarse-grained HTM implementations that we discuss later. All operations are simply wrappers of the corresponding serial search tree operations, executed inside a critical section, which is either a lock-based critical section or an HTM transaction. Although trivial

[1]`https://github.com/jimsiak/concurrent-maps-cpp`

to implement, these coarse-grained locking approaches do not provide high performance since they serialize all operations, thus hindering concurrency.

```
template <class K, class V>
class cg_st : public Map<K, V> {
private:
  Map<K,V> *serial_st;
  cg_sync *sync_mechanism;
public:
  cg_st(Map<K,V> *prot, string& sync_type) {
    serial_st = prot;

    if (sync_type == "cg-htm")           sync_mechanism = new cg_sync_htm();
    else if (sync_type == "cg-rwlock")   sync_mechanism = new cg_sync_rwlock();
    else if (sync_type == "cg-spinlock") sync_mechanism = new cg_sync_spinlock();
  }

  V lookup(K& key) {
    sync_mechanism->cs_enter_ro();
    V ret = serial_st->lookup(key);
    sync_mechanism->cs_exit();
    return ret;
  }

  V insert(K& key, V& val) {
    sync_mechanism->cs_enter_rw();
    V ret = serial_st->insert(key, val);
    sync_mechanism->cs_exit();
    return ret;
  }

  V remove(K& key) {
    sync_mechanism->cs_enter_rw();
    V ret = serial_st->remove(key);
    sync_mechanism->cs_exit();
    return ret;
  }
};
```

Listing 3.1: A search tree protected by coarse-grained synchronization.

3.1.2 Contention-Adapting Locking

Contention-Adapting (CA) search trees were proposed by Sagonas et. al [SW15]. The structure of CA trees is shown in Figure 3.1. The actual data is stored in sequential data structures and on top of these data structures an additional tree based structure is maintained, which consists of a set of routing and base nodes. When searching for a specific key, they locate the appropriate sequential data structure, i.e., find the corresponding base node by following a path of routing nodes. Each sequential data structure is protected by the base node's lock.

Every base node maintains statistics about the contention level in its sequential data structure, i.e., the times that its lock was found to be already acquired. Under certain circumstances when the contention level is found to be very high or very low, the sequential data structure is either split to two base nodes or joined with a sibling base node sequential data structure respectively. Thus, for CA synchronization mechanism to be applicable to a sequential data structure, the data structure must support split and join operations. The interface of the methods that should be implemented by the sequential data structure is shown in Listing 3.2. If these two methods are available, the implementation of the *Map* methods is straightforward. Listing 3.3 presents the implementation of the *insert()* operation; *lookup()* and *remove()* are identical.

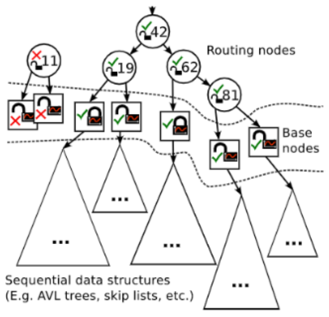

42
11
19
62
81
Routing nodes
Base nodes
Sequential data structures (E.g. AVL trees, skip lists, etc.)

Figure 3.1: The structure of a CA tree. The image was taken directly from the authors' original paper [SW15].

```
template <typename K, typename V>
class Map {
public:
  ...

  //> Methods necessary to support Contention-Adapting locking
  void *split(void **right_part);
  void *join(void *other_ds);
};
```

Listing 3.2: Additions to the *Map* interface to enable support for Contention-Adapting locking.

```
void adapt_if_needed(base_node_t *bnode, route_node_t *parent, route_node_t *gparent) {
  if (bnode->lock_statistics > STAT_LOCK_HIGH_CONTENTION_LIMIT) {
    split(bnode, parent);
    bnode->lock_statistics = 0;
  } else if (bnode->lock_statistics < STAT_LOCK_LOW_CONTENTION_LIMIT) {
    join(bnode, parent, gparent);
    bnode->lock_statistics = 0;
  }
}

V insert(K& key, V& value) {
  V ret;
  base_node_t *bnode;
  route_node_t *parent, *gparent;

  while (1) {
    bnode = find_base_node(&parent, &gparent, key);
    bnode->lock();
    if (!bnode->is_valid()) {
      bnode->unlock();
      continue;
    }
    ret = bnode->serial_st->insert(key, value);
    adapt_if_needed(bnode, parent, gparent);
    bnode->unlock();
    return ret;
  }
}
```

Listing 3.3: Insert operation for a contention-adapting search tree.

3.1.3 Fine-grained Locking

Several fine-grained locking search trees have been proposed [BCCO10, DVY14, CGR13] and all use one lock for each tree node. There are two major problems with fine-grained locking. First, the design and implementation procedures are complex and error-prone since the programmer needs to manually write all the code to acquire and release multiple locks. Second, complex tree operations, such as rebalancing, would require a very large number of nodes to be locked, stoping other concurrent operations from accessing large parts of the tree for a large time duration. Also, specifically for rebalancing, the reverse traversal of the tree performed during rebalance, increases the risk of deadlocks with conflicting operations that traverse the tree in the typical root-to-leaves direction.

In order to avoid locking large parts of the tree, all current fine-grained locking trees [BCCO10, DVY14, CGR13] are based on relaxed-balanced AVL trees, where the rebalancing procedure follows the algorithm presented by Bouge et. al [BGMS98]. This splits a rebalancing operation which may span a large subtree, in multiple independent rebalancing steps, which affect only a set of two or three nodes. These rebalancing steps can either be triggered immediately after an insertion or removal of a node [BCCO10, DVY14], or can be delegated to a thread that is responsible only for rebalancing the tree [CGR13].

Another problematic case with fine-grained locking when applied specifically to binary search trees, is when an internal node needs to be removed from the tree [2]. In this case, the node will be replaced by its successor, i.e., the leftmost node in its right subtree. The successor may be found several levels lower and all the nodes leading to the successor need to be locked, otherwise operations that search the successor's key may be led to the wrong direction. Current fine-grained locking implementations avoid this problematic case by using two different techniques. Bronson's [BCCO10] and Crain's [CGR13] search trees use a partially external tree structure, where each node has a marked field which indicates whether its key-value pair is currently present in the *map* or not. Marked nodes are still present in the structure of the tree but their key-value pair does not actively participate in the *map* data structure. To avoid these marked nodes, which lead to increased traversal times, Drachsler's tree [DVY14] uses two additional pointers on each node to point to the predecessor and the successor node respectively. These pointers are then used by traversals to locate the correct location of a node that has been displaced.

3.2 Lock-free Search Trees

Lock-free search trees exploit the hardware-provided atomic instructions, such as Compare-and-Swap (CAS) and Load-Linked-Store-Conditional (LL/SC). These two variants are used in a very

[2]This is not a problem in B+-trees since it is never the case that an internal node needs to be removed.

similar way, so from now on we will only refer to CAS. Several lock-free search trees have been proposed [EFRvB10, HJ12, NM14, CDT14, RM15, Bro17, BPA20, WPL$^+$18], which either use CAS directly (CAS-based) or use Double-Compare-Single-Swap (DCSS) and Load-Linked-Store-Conditional-Extended (LLX/SCX), two high level synchronization primitives that are implemented on top of CAS.

3.2.1 Compare-And-Swap (CAS)

In CAS-based lock-free trees [EFRvB10, HJ12, NM14, CDT14, RM15], it is challenging to support operations that typically modify multiple nodes, such as rebalancing and internal node deletion. The problem arises from the restriction of CAS instruction to a single memory location and it is the reason why these lock-free binary search trees are unbalanced. For the same reason, lock-free search trees are either external trees [EFRvB10, NM14] or partially-external [HJ12, CDT14, BER14, RM15], which may lead to long traversal paths, especially for large trees.

3.2.2 High-level lock-free primitives

High-level lock-free primitives, such as DCSS and LLX/SCX provide extended CAS operations which validate that multiple locations have not been modified (instead of only one when using the typical CAS operation), and modify a single one. These two techniques have been applied on search trees but they incur high overheads. DCSS augments each memory location with a tag field which is manipulated using bitwise operations. This is very expensive in the case of complex search tree operations, where multiple memory locations need to be read and validated. LLX-SCX adds an SCX record on each tree node, and requires multiple CAS operations to execute a complex search tree operation.

3.3 RCU-based Search Trees

Read-Copy-Update [MS98] is a synchronization mechanism ideal for mostly-read workloads. It is widely used in the Linux kernel and on several production libraries. Two RCU-based trees can be found in the literature, specifically for search trees, one by Howard et. al [HW14] and one by Arbel et. al [AA14]. In RCU, read-only operations do not use any synchronization and are never affected by concurrent operations. To enable this, update operations create copies of the appropriate parts of the search tree prior to modifying them and, when their private copy is ready, they install it in the shared data structure by changing only a single node's child pointer. This change is performed in a single atomic step, allowing concurrent readers to safely read either the old or the new version of the specific subtree.

While read-only operations can safely run concurrently with other read-only and/or update operations, update operations still need to be synchronized in some way. The two aforementioned search trees [HW14, AA14] use locks to synchronize updaters, the first one uses coarse-grained locking, i.e., a single updaters lock for the whole data structure, while the second one uses fine-grained locking, i.e., one lock per tree node.

3.3.1 RCU with coarse-grained locking

Relativistic programming [HW14] uses RCU with a single lock to synchronize updaters (we will refer to this implementation as RCU single-global-lock, RCU-SGL). Its implementation is simple and shown in Listing 3.5. The *rcu_sgl* class wraps a serial search tree. For a serial search tree to be used with RCU-SGL, three methods need to be implemented which are shown in Listing 3.4. The advantage of RCU-SGL is that these three methods can be implemented relatively easily. As an example, Listing 3.6 presents the implementation of *insert_with_copy()* for an external AVL tree. *Traverse_with_stack()* and *install_copy()* are omitted since their implementation is trivial. The first one is similar to a serial search tree traversal, with the only addition that the pointers followed by the traversal are stored in the provided stack. *install_copy()* simply changes the appropriate child pointer of *connection_point* to point to the newly created private copy, i.e., *priv_copy*. As already mentioned, the *lookup()* method simply calls the corresponding *lookup()* method of the serial search tree. The *insert()* method uses *traverse_with_stack()*, *insert_with_copy()* and *install_copy()* methods after the acquisition of the *updaters_lock*.

```
template <typename K, typename V>
class Map {
public:
  ...

  //> Methods necessary to support RCU-SGL
  V traverse_with_stack(K& key, void **stack, int *stack_top);
  void *insert_with_copy(K& key, V& val, void **stack, int *stack_top,
                         void **priv_copy);
  void *delete_with_copy(K& key, void **stack, int *stack_top,
                         void **priv_copy);
  void install_copy(K& key, void *connection_point, void *priv_copy);
};
```

Listing 3.4: Additions to the *Map* interface to enable support for RCU-SGL

```
template <typename K, typename V>
class rcu_sgl : public Map<K,V> {
private:
  pthread_spinlock_t updaters_lock;
  Map<K,V> *serial_st;
public:
  rcu_sgl(Map<K,V> *serial_st) {
    this->serial_st = serial_st;
    pthread_spin_init(&updaters_lock, PTHREAD_PROCESS_SHARED);
  }

  V lookup(K& key) { return serial_st->lookup(key); }

  V insert(K& key, V& val) {
    void *stack[MAX_STACK_LEN];
    void *connection_point, *priv_copy;
    int stack_top;

    LOCK(&updaters_lock);
    V ret = serial_st->traverse_with_stack(key, stack, &stack_top);
    if (ret != NO_VALUE) {
      UNLOCK(&updaters_lock);
      return ret;
    }
    connection_point = serial_st->insert_with_copy(key, val, stack, &stack_top,
                                                   *priv_copy);
    serial_st->install_copy(key, connection_point, priv_copy);
    UNLOCK(&updaters_lock);
    return NO_VALUE;
  }

  //> remove() is exactly similar to insert(), except for the call to
  //> insert_with_copy() which is replaced by delete_with_copy().
  V remove(K& key) { ... }
};
```

Listing 3.5: A search tree with RCU-SGL synchronization.

```
void *insert_with_copy(const K& key, const V& value, void **stack, int *stack_top, void **privcopy) {
  node_t *connection_point;
  node_t **node_stack = (node_t **)stack;

  //> Initiate the private copy with the new node.
  *privcopy = new node_t(key, value);
  connection_point = *stack_top >= 0 ? node_stack[*stack_top--] : NULL;

  //> Empty tree case
  if (*stack_top < 0) return connection_point;

  while (*stack_top >= -1) {
    //> If we've reached or passed root of the tree, return.
    if (!connection_point) break;

    //> If no height change occurs we can return.
    if ((*privcopy)->height + 1 <= connection_point->height) break;

    //> Copy the current node and link it to the local copy.
    node_t *curr_cp = node_copy(connection_point);

    curr_cp->height = (*privcopy)->height + 1;
    if (key < curr_cp->key) curr_cp->left = *privcopy;
    else                    curr_cp->right = *privcopy;
    *privcopy = curr_cp;

    // Move one level up
    connection_point = *stack_top >= 0 ? node_stack[*stack_top--] : NULL;

    // Get current node's balance
    node_t *sibling;
    int curr_balance;
    if (key < curr_cp->key) {
      sibling = curr_cp->right;
      curr_balance = node_height(curr_cp->left) - node_height(sibling);
    } else {
      sibling = curr_cp->left;
      curr_balance = node_height(sibling) - node_height(curr_cp->right);
    }

    if (curr_balance == 2) {
      int balance2 = node_balance((*privcopy)->left);

      if (balance2 == 1) {
        *privcopy = rotate_right(*privcopy);
      } else if (balance2 == -1) {
        (*privcopy)->left = rotate_left((*privcopy)->left);
        *privcopy = rotate_right(*privcopy);
      }
      break;
    } else if (curr_balance == -2) {
      int balance2 = node_balance((*privcopy)->right);

      if (balance2 == -1) {
        *privcopy = rotate_left(*privcopy);
      } else if (balance2 == 1) {
        (*privcopy)->right = rotate_right((*privcopy)->right);
        *privcopy = rotate_left(*privcopy);
      }
      break;
    }
  }

  return connection_point;
}
```

Listing 3.6: Implementation of *insert_with_copy()* for an internal AVL tree.

3.3.2 RCU with fine-grained locking

Arbel et. al [AA14] introduced Citrus, an RCU based search tree which uses fine-grained locking to synchronize updaters. This way updaters can run concurrently allowing for much better performance. However, the problem with fine-grained locks is the difficulty to support complex operations such as rebalancing. For this reason, Citrus is an unbalanced tree.

```
int validate(node_t *prev, node_t *curr, int direction) {
  int result;
  result = !(prev->marked);
  if (direction == 0) result = result && (prev->left == curr);
  else                result = result && (prev->right == curr);
  if (curr != NULL) result = result && (!curr->marked);
  return result;
}

int do_insert(const K& key, const V& value, node_t *prev, node_t *curr,
              int direction) {
  node_t *new_node;
  LOCK(&prev->lock);
  if(!validate(prev, curr, direction)) {
    UNLOCK(&prev->lock);
    return 0;
  }
  new_node = new node_t(key, value);
  if (direction == 0) prev->left = new_node;
  else                prev->right = new_node;
  UNLOCK(&prev->lock);
  return 1;
}

V insert(K& key, V& value) {
  node_t *prev, *curr, *new_node;

  while(1) {
    rcu_read_lock();
    int direction = traverse_with_direction(key, &prev, &curr);
    rcu_read_unlock();

    // Key already in the tree
    if (curr != NULL) return curr->value;

    if (do_insert(key, value, prev, curr, direction) == 1) return NO_VALUE;
  }
}

int do_remove(node_t *prev, node_t *curr, int direction) {
  LOCK(&prev->lock);
  LOCK(&curr->lock);
  if(!validate(prev, curr, direction)) {
    UNLOCK(&prev->lock);
    UNLOCK(&curr->lock);
    return 0;
  }

  if (!curr->left) {
    curr->marked = true;
    if (direction == 0) prev->left = curr->right;
    else                prev->right = curr->right;
    UNLOCK(&prev->lock);
    UNLOCK(&curr->lock);
    return 1;
  } else if (!curr->right) {
    curr->marked = true;
    if (direction == 0) prev->left = curr->left;
    else                prev->right = curr->left;
    UNLOCK(&prev->lock);
    UNLOCK(&curr->lock);
    return 1;
  }

  node_t *prevSucc = curr;
  node_t *succ = curr->right;
  node_t *next = succ->left;
  while (next != NULL){
    prevSucc = succ;
    succ = next;
    next = next->left;
  }

  int succDirection = 1;
  if (prevSucc != curr){
    LOCK(&prevSucc->lock);
    succDirection = 0;
  }
  LOCK(&succ->lock);
  if (validate(prevSucc, succ, succDirection) && validate(succ, NULL, 0)) {
    curr->marked=1;
    node_t *new_node = new node_t(succ->key, succ->value);
    new_node->left = curr->left;
    new_node->right = curr->right;
    LOCK(&new_node->lock);
    if (direction == 0) prev->left = new_node;
    else                prev->right = new_node;
    urcu_synchronize();
    succ->marked = true;
    if (prevSucc == curr) new_node->right = succ->right;
    else                  prevSucc->left = succ->right;
    UNLOCK(&prev->lock);
    UNLOCK(&new_node->lock);
    UNLOCK(&curr->lock);
    if (prevSucc != curr) UNLOCK(&prevSucc->lock);
    UNLOCK(&succ->lock);
    return 1;
  }
  UNLOCK(&prev->lock);
  UNLOCK(&curr->lock);
  if (prevSucc != curr) UNLOCK(&prevSucc->lock);
  UNLOCK(&succ->lock);
  return 0;
}

V remove(K& key) {
  node_t *prev, *curr;
  int direction;

  while(1) {
    rcu_read_lock();
    direction = traverse_with_direction(key, &prev, &curr);
    rcu_read_unlock();

    // Key not found
    if (!curr) return this->NO_VALUE;

    const V del_val = curr->value;
    if (do_remove(prev, curr, direction) == 1) return del_val;
  }
}
```

Listing 3.7: Implementation of Citrus, a concurrent unbalanced internal search tree with RCU and fine-grained locking.

3.4 HTM-based Search Trees

3.4.1 Coarse-grained HTM with single lock fallback

The most straightforward way to apply HTM on a search tree is to enclose each operation in an HTM transaction, essentially replacing the single global lock of the coarse-grained locking implementation. As already discussed in Chapter 2, HTM provides no guarantees that a transaction will commit, it is thus necessary for the programmer to provide an alternative non-transactional fallback path that executes after a number of transactional aborts. In coarse-grained HTM implementations this fallback path simply acquires the lock, aborting any other concurrent transactions, and serializes the operation. Although trivial to implement, these coarse-grained approaches do not provide high performance since they suffer from large numbers of transactional aborts due to the large size of their transactions, both in terms of memory locations and time duration.

3.4.2 3-Path HTM

The classic coarse-grained HTM approach consists of 2 execution paths for each operation; the transactional path, which executes the operation inside a transaction, and the non-transactional fallback path which acquires a single global lock. When an operation enters the fallback path, no other operation can run concurrently, which greatly degrades performance. Brown et. al [Bro17] introduced 3-path HTM, where each operation consists of 3 execution paths, the fast, middle and slow paths. The fast path is the same as in the classic 2-path approach and executes the operation in a transaction. The middle path still uses a transaction but it is modified so as to be able to execute concurrently with the slow path. Finally, the slow path uses the lock-free LLX/SCX primitive to synchronize concurrent operations. The 3-path HTM approach allows concurrency between operations that execute the middle and slow paths, thus avoiding the excessive serialization of operations in the non-transactional fallback path of 2-path HTM. The advantage of 3-path HTM is that it adapts the high performance of 2-path HTM under low contention scenarios and still allows concurrent operations to execute under high contention.

3.4.3 Consistency Oblivious Programming with HTM

Consistency-Oblivious programming (COP) [AAS11, AK14] provides a way to use HTM with a smaller footprint than the coarse-grained HTM synchronization. A search tree operation in COP is divided into two parts: a read-only phase, and an updating read-write phase. The read-only phase runs without any synchronization and includes the traversal of the tree. This traversal may lead to a false location due to concurrent modifications on the path that the operation follows. The updating phase starts an HTM transaction with two objectives; it first validates that the traversal has led to the right location, and perform any writes needed by the operation.

Although COP indeed reduces the transaction size relatively to the coarse-grained HTM synchronization, it still has two drawbacks; first, lookup operations need to use transactions and hence pay their overhead, and second, the whole update operation (including the rebalance step) is enclosed in a single transaction, leading to a large transactional write set, which results in a high probability of conflicts.

CHAPTER 4

RCU-HTM

This chapter presents the main contribution of this thesis, i.e., *RCU-HTM*, a novel synchronization mechanism that can be applied to search trees to create highly efficient concurrent implementations. We first give a high level overview of *RCU-HTM* and then provide all the necessary information about how *RCU-HTM* can be used, together with two examples of *RCU-HTM* based search trees. We also provide an informal correctness proof by defining the linearization points of the *RCU-HTM* operations. Finally, we explain how we applied a memory reclamation scheme on *RCU-HTM*, which helps avoid excessive use of memory and makes *RCU-HTM* practical to use in real-life long-running parallel applications.

4.1 High Level Overview

RCU-HTM combines Read-Copy-Update (RCU) with Hardware Transactional Memory (HTM) in a novel way and takes advantage of their key performance characteristics, i.e., *asynchronized read-only operations*, and, *optimistic concurrent execution of updaters*, respectively. At the same time, *RCU-HTM* mitigates their limitations, i.e., serialization of updaters and large transaction sizes. *RCU-HTM* is, to the best of our knowledge, the first synchronization mechanism that sustains high performance over a wide range of execution scenarios, as our experimental evaluation validates, and combines it with wide applicability on all types of search trees.

RCU is used in the exact same way as in RCU implementations with coarse-grained locking [HW14]. Updaters perform their modifications on private copies of the appropriate nodes,

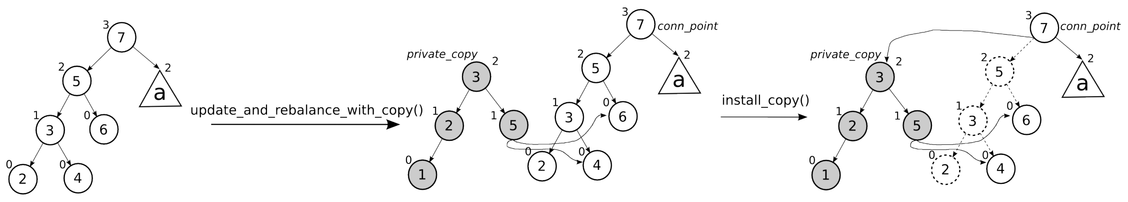
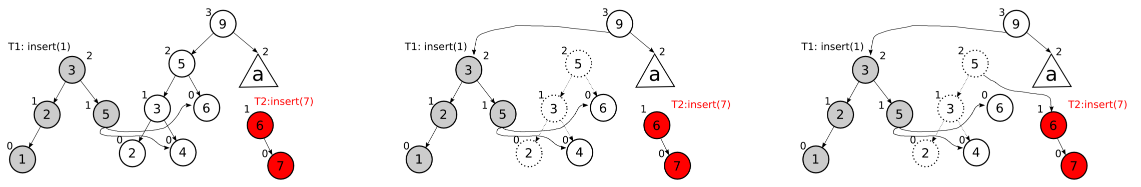


Figure 4.1: Insertion of key 1 followed by rebalancing in an RCU-based internal AVL tree.

instead of in-place, as depicted in Figure 4.1. When the private copy is ready, updaters install it in the shared tree in a single atomic step by modifying one child pointer of a single tree node, called the *connection point*. In this way, readers can safely traverse the tree without synchronization, as they will observe either the previous unmodified or the new modified version of the affected nodes; both cases produce correct results as they can be successfully linearized (the correctness proof can be found in Section 4.4).

While RCU already provides asynchronized read-only operations, the novelty of *RCU-HTM* lies in leveraging HTM to optimistically synchronize updaters. The most straightforward way of using HTM is to replace the updaters' global lock used in the coarse-grained locking approach [HW14], with an HTM transaction. However, this solution has two drawbacks: first, it creates large transactions which are subject to conflict and capacity aborts, and second, update operations that do not modify the tree, still pay the overheads of using transactions. To avoid these, *RCU-HTM* performs the traversal phase and the creation of the private copy of update operations outside of its transactions. It encloses only the installation of the private copy, i.e., the update of a single child pointer, along with a validation step that is necessary to ensure that the modified nodes can safely be replaced by their new version.

To illustrate why the validation step is necessary, Figure 4.2 presents an erroneous concurrent execution of two insert operations when no validation is performed. In Figure 4.2a both threads execute *insert_and_rebalance_with_copy()* and create their private copies. Thread T1 executes *install_copy()* first and replaces 3 nodes. Afterwards, thread T2 installs its copy under the original node 5, since this is the version of the node it read during its traversal phase. However, the node with key 5 has been replaced in the meanwhile by T1's new modified version and the modification of T2 is thus discarded. We avoid such erroneous executions in *RCU-HTM* by adding a validation step which ensures that the nodes to be replaced have not been modified since they were read. To achieve this, when we traverse the tree and create the modified copy, we maintain a validation set which contains the pointers of the access path plus the sibling pointers of the copied nodes. We then validate that all these pointers have not been modified in the meanwhile. In the example of Figure 4.2, thread T2 would fail in the validation step, because the left pointer of node 9 (which



(a) Both threads execute *insert_and_rebalance_with_copy()*. (b) T1 executes *install_copy()*. (c) T2 executes *install_copy()*.

Figure 4.2: An erroneous execution of two threads inserting keys in an AVL tree. Thread T2 installs its copy on a removed node so its modification does not become visible to other threads.

is part of its access path) has changed. This would cause T2 to restart its operation instead of installing its copy on the removed path of the tree.

RCU-HTM is able to deliver two key properties of concurrent data structures: a) applicability to multiple types of search trees, and, b) high and sustainable performance accross different execution scenarios (i.e., different combinations of key size, tree size, mix of operations and number of concurrent threads). These properties are further analyzed below:

Applicability of *RCU-HTM*. *RCU-HTM* can be applied to any search tree where RCU is applicable; that is, any search tree whose update operations can be performed by copying the affected set of nodes and then installing the modified copy by swapping a single child pointer. Examples of such types of search trees are binary search trees (unbalanced/relaxed-balanced/balanced, internal/partially-external/external), B-trees, B+-trees, (a-b)-trees and radix trees. In our current implementations of *RCU-HTM* search trees, we do not use parent pointers to perform the reverse traversals of the trees but we store the access path in a stack. This is not restrictive, since all the search trees we encountered could easily be implemented in such a way.

Another advantage of *RCU-HTM* is that the procedure of applying it, is very similar across different types of search trees, enabling a programmer who has implemented one *RCU-HTM* based search tree to use a very similar procedure for any other type of search tree. This facilitates the easy and timely implementation of multiple concurrent *RCU-HTM* trees. This is not true for other highly efficient fine-grained synchronization mechanisms, such as fine-grained locking and lock-free approaches, where each type of search tree requires a complete rethinking of how and when should locks be acquired or which pointers can be swapped using CAS operations.

Performance of *RCU-HTM*. The main performance benefits of *RCU-HTM* come from its completely asynchronized read-only operations, both lookups and unsuccessful updates, i.e., insertions that find the key already in the tree and deletions that do not locate the key to be deleted. Lookups do not lock any node or perform any CAS operation or HTM transaction nor do they perform any additional checks during the traversal of the tree. Their performance is thus similar to their serial counterpart. Unsuccessful update operations are also completely asynchronized.

The only added overhead compared to the serial version, is that we store the traversed pointers in the validation set. Although this is not used, since unsuccessful updates do not need to perform the validation step, we do not know this a priori and therefore still need to store those pointers. On the other hand, successful update operations pay the overhead of copying nodes and validating the stored pointers. The validation overhead though is low, since these pointers have been recently accessed and can be found in the highest levels of the cache hierarchy with high probability. As our evaluation shows, these overheads are outweighed by the performance benefits of *RCU-HTM*.

4.2 How to use *RCU-HTM*

An *RCU-HTM* based concurrent search tree is represented with the *rcu_htm* class which is a subclass of our main *Map* class. Listing 4.1 presents a simple example of creating and using an *RCU-HTM* AVL tree. Any other search tree that implements the methods presented in section 4.3 can be used in the same way. The only difference, in comparison to any other *Map* object, is that the constuctor of the *RCU-HTM* tree takes as argument an instance of a tree which needs to implement the methods that we describe in the following sections. Apart from that, an instance of the *rcu_htm* class can be used in exactly the same way as an instance of the base *Map* class.

```
#include "rcu-htm.h"
#include "avl.h"

void main()
{
  avl<int, void*> *avl_tree;
  rcu_htm<int, void*> *rcuhtm_tree;

  avl_tree = new avl<int, void*>();
  rcuhtm_tree = new rcu_htm<int, void*>(avl_tree);

  for (int i=0; i < 1000; i++)
    rcuhtm_tree->insert(i, (void *)i);

  void *ret = rcuhtm_tree->find(100);
  assert(ret == (void *)100); // Value found

  rcuhtm_tree->remove(100);

  void *ret = rcuhtm_tree->find(100);
  assert(ret == NO_VALUE); // Value not found
}
```

Listing 4.1: An example using an AVL *RCU-HTM* tree.

4.3 The *RCU-HTM* class

The source code of *rcu_htm* class is shown in Listing 4.2. Two private fields are added; *updaters_lock* is the lock that is used by the updater threads when they need to enter the non-transactional fallback execution path, and, *seq_ds* is the data structure that is protected by the

RCU-HTM technique. For the two methods *contains()* and *find()*, *RCU-HTM* just calls the respective serial method, since *RCU-HTM* traversal operations do not use any synchronization. Regarding the *rangeQuery()* method we explain how it can be implemented in Chapter 6.

```
template <typename K, typename V>
class rcu_htm : public Map<K,V> {
private:
  pthread_spinlock_t updaters_lock;
  Map<K,V> *seq_ds;

public:
  rcu_htm(Map<K,V> *seq_ds) {
    this->seq_ds = seq_ds;
    pthread_spin_init(&updaters_lock, PTHREAD_PROCESS_SHARED);
  }

  bool          contains(K& key) { return seq_ds->contains(key); }
  pair<V,bool> find(K& key) { return seq_ds->find(key); }
  int           rangeQuery(K& key1, K& key2, vector<pair<K,V>> kv_pairs);
  V             insert(K& key, V& val);
  pair<V,bool> remove(K& key);
};
```

Listing 4.2: The *RCU-HTM* class.

```
V traverse_with_stack(K& key, void **stack, int *stack_indexes, int *stack_top);

void install_copy(void *connpoint, void *privcopy, int *stack_indexes,
                  int connpoint_stack_index);

void validate_copy(void **stack, int *stack_indexes, int stack_top);

void *insert_with_copy(K& key, V& value,
                       void **stack, int *stack_indexes, int stack_top,
                       void **privcopy, int *connpoint_stack_index);

void *remove_with_copy(K& key,
                       void **stack, int *stack_indexes, int *stack_top,
                       void **privcopy, int *connpoint_stack_index);
```

Listing 4.3: The methods that need to be implemented by any data structure that will be used with *RCU-HTM*.

The methods *insert()* and *remove()* of *rcu_htm* require that the underlying sequential data structure implements a set of methods. The exact interfaces of these methods are presented in Listing 4.3 and are the following:

- *traverse_with_stack()* : This method performs the traversal of the tree. It is very similar to the classic tree traversal with the addition that we also maintain a stack of pointers to the nodes that have been accessed during the traversal (i.e., the *stack* argument) as well as the index of the child that has been chosen on each node (i.e., the *stack_indexes* argument).

- *insert_with_copy()* : This method generates a private copy of the part of the tree that will be affected by the insertion of the given key-value pair. It does not modify the tree in any way as any node that needs to be modified is copied. The private copy is a subtree which is pointed to by *privcopy*. The method returns a pointer to the *connection point*, i.e., the node of the original tree where the generated private copy will be attached.

- *remove_with_copy()* : This method, similarly to *insert_with_copy()*, generates a private copy of the affected part of the tree with the given key removed.

- *validate_copy()* : This method validates that the part of the tree that will be replaced by the operation has remained unchanged since the copy was created. If some part of the tree has changed, the operation is aborted and starts again.

- *install_copy()* : This method is responsible for installing the copy of the subtree that has been generated by the call to either *insert_with_copy()* or *remove_with_copy()*. It is as simple as changing a single node's child pointer.

These methods are not hard to implement and can be debugged in a serial manner without the need to take into account concurrent threads' interleavings. The programmer needs to ensure that the node copies are created when necessary and that the validation set contains all the nodes that will be replaced by the operation. In our future work we aim to facilitate this even further by creating a library or a compiler plugin to make this procedure automatic.

With these five methods available, the *insert()* and *remove()* operations of the respective *RCU-HTM* concurrent search tree are implemented as presented in Listing 4.4. We only show *insert()*, since *remove()* is implemented in exactly the same way by just replacing the calls to *insert_with_copy()* with *remove_with_copy()*. The *RCU-HTM* update operation performs the traversal and the creation of the private copy without any synchronization (lines 14–23). Then *validate_and_install_copy()* validates, in a single atomic step using an HTM transaction, that the nodes to be replaced have not been modified by another concurrent updater and installs the private copy in the shared tree. If *RCU-HTM* continuously fails to complete the operation (i.e., the operation has been restarted for MAX_OPERATION_RETRIES times due to transactional aborts), we fallback to the plain RCU-based algorithm, to ensure forward progress, where updaters are serialized using a single global lock (lines 33–48).

```
1  template <typename K, typename V>
2  V rcu_htm<K,V>::insert(K& key, V& val)
3  {
4    void *node_stack[MAX_STACK_LEN];
5    int stack_indexes[MAX_STACK_LEN], stack_top, index;
6    void *connection_point, *tree_cp_root;
7    int retries = -1;
8    int connection_point_stack_index;
9
10   //> First try with the RCU-HTM way ...
11   while (++retries < MAX_OPERATION_RETRIES) {
12     ht_reset(tdata->ht);
13
14     //> Asynchronized traversal. If key is there we can safely return.
15     V ret = seq_ds->traverse_with_stack(key, node_stack,
16                                           _stack_indexes, &stack_top);
17     assert(stack_top < MAX_STACK_LEN);
18     if (ret != this->NO_VALUE) return ret;
19
20     connection_point = seq_ds->insert_with_copy(key, val,
21                                       node_stack, stack_indexes, stack_top,
                                      &tree_cp_root,
                                      &connection_point_stack_index);
    bool installed = validate_and_install_copy(connection_point, tree_cp_root,
                                               node_stack, stack_indexes,
                                               stack_top,
                                               connection_point_stack_index);
    if (installed) return this->NO_VALUE;
  }

  //> ...otherwise fallback to the coarse-grained RCU
  ht_reset(tdata->ht);
  pthread_spin_lock(&updaters_lock);
  V ret = seq_ds->traverse_with_stack(key, node_stack,
                                            stack_indexes, &stack_top);
  assert(stack_top < MAX_STACK_LEN);
  if (ret != this->NO_VALUE) {
    pthread_spin_unlock(&updaters_lock);
    return ret;
  }
  connection_point = seq_ds->insert_with_copy(key, val,
                                  node_stack, stack_indexes, stack_top,
                                  &tree_cp_root,
                                  &connection_point_stack_index);
  seq_ds->install_copy(connection_point, tree_cp_root,
                       stack_indexes,
                       connection_point_stack_index);
  pthread_spin_unlock(&updaters_lock);
  return this->NO_VALUE;
}

template <typename K, typename V>
bool rcu_htm<K,V>::validate_and_install_copy(void *connpoint, void *tree_cp_root,
                               void **node_stack, int *stack_indexes,
                               int stack_top, int connection_point_stack_index)
{
  unsigned int status;
  int validation_retries = -1;

  while (++validation_retries < MAX_VALIDATION_RETRIES) {
    while (updaters_lock != LOCK_FREE) ;

    status = TX_BEGIN(0);
    if (status == TM_BEGIN_SUCCESS) {
      if (updaters_lock != LOCK_FREE)
        TX_ABORT(ABORT_GL_TAKEN);

      seq_ds->validate_copy(node_stack, stack_indexes, stack_top);
      seq_ds->install_copy(connpoint, tree_cp_root,
                           stack_indexes,
                           connection_point_stack_index);
      TX_END(0);
      return true;
    } else if (ABORT_IS_EXPLICIT(status) &&
             ABORT_CODE(status) == ABORT_VALIDATION_FAILURE) {
      return false;
    }
  }
  return false;
}
```

Listing 4.4: Template code for all *RCU-HTM* implementations.

4.4 Correctness

In this section we discuss the correctness of the *RCU-HTM* synchronization mechanism. We use the well-known correctness condition of linearizability [HW90] and define the linearization point for each operation, i.e., the point at which the operation's modifications become visible to other threads. The linearization points of the *RCU-HTM* technique are similar for any type of search tree that is applied to, constituting one more advantage relative to other synchronization mechanisms that may require a different lengthy correctness proof for different search trees. The only thing a programmer who implements an *RCU-HTM* search tree needs to do, is to make sure that all the necessary pointers have been added in the validation set. We focus on the AVL example given in Section 4.5 but the same correctness reasoning stands for any *RCU-HTM* search tree.

Lookup operation. The reasoning about the linearizability of lookup in *RCU-HTM* is identical to other RCU-based implementations [AA14, HW14]. In all RCU-based algorithms, including *RCU-HTM*, updaters commit their copies by modifying a single memory location, i.e., the appropriate child pointer of the connection point. Since single-word reads and writes are atomic, readers observe either the old or the new version of the data structure. Moreover, because *RCU-HTM* avoids performing any rotations directly on the tree, traversals never follow a wrong path.

RCU-HTM lookups are identical to those of the serial version of the tree. A lookup operation that observes an empty tree is linearized at the point where the root of the tree is read as *NULL*. When the tree is not empty, the linearization point is at at the point when we read the appropriate child of the parent of the leaf node that contains the key. There is a time window between the read of this child pointer and the point at which the lookup operation returns, during which the leaf node may have been removed from the tree by some concurrent insertion or deletion. Even then, however, the lookup is safely linearized before this update operation.

Update operations. We can easily prove the correctness of the update operations if we consider the main three characteristics of *RCU-HTM* design. First, HTM guarantees that all operations enclosed inside a transaction are executed *atomically*, i.e., the validation and installation of the modified copy in the shared tree (lines 63–71 of *validate_and_install_copy()* in Listing 4.4) can be seen as one indistinguishable operation, which greatly facilitates our proof. Second, when an updater resorts to the fallback path with the updaters' lock (due to multiple consecutive failed transactional attempts), no other updater executes concurrently. This is guaranteed by lines 61 and 65 of *validate_and_install_copy()*. At line 61, an updater waits until the updaters' lock is released before starting a transaction, and at line 65, the updaters' lock is checked and, if it is locked, the transaction immediately aborts. By reading the value of the updaters' lock at line 65, we add it in the transaction's read-set in order to trigger a conflict abort, in case another updater acquires the updaters' lock during the transaction's lifetime. Such a conflict abort will cause any updater that executes a transaction to retry from line 60 and will consequently spin at line 61 until the updaters' lock is released. Third, operations executed by one thread can never be observed by other threads "half-done", i.e., a thread will never read a node which is being concurrently rotated. This is true because updaters install their copies by changing a single child pointer and the modification of a single memory word is atomic. Thus, a thread will see either the whole result of another operation (i.e., the whole modified subtree copy) or nothing (i.e., it will read the old version of the subtree). This allows *RCU-HTM* to avoid problematic situations in the asynchronized reverse traversal performed by updaters. During this reverse traversal, an updater may observe a different tree than the version it observed during the root to leaf traversal. Despite this inconsistent view of the tree, the updater will safely proceed (e.g., without causing a segmentation fault due to reading a stale sibling pointer) to the validation step, which will then fail and cause the updater to restart its operation.

Unsuccessful update operations (i.e., inserts that find the key in the tree and deletes that do not find it) do not modify the tree structure and are linearized similarly to a lookup operation, i.e., at the point when the node containing the searched key is read. Successful update operations (i.e., inserts that do not find the key in the tree and deletes that do find it) modify the tree and are linearized in one of two points depending on whether they managed to complete their operation using transactions or resorted to the fallback path. As already mentioned, an updater that resorts to the fallback path by acquiring the updaters' lock does not allow concurrent updaters and its operation is linearized at the time when the lock is acquired, i.e., inside the *insert()* in Listing 4.4 at line 33. An updater that executes the common transactional path runs concurrently with other updaters; however, since HTM guarantees that all the accesses inside a transaction are atomic, these update operations are linearized at line 72 of Listing 4.4, when a hardware transaction commits installing the private copy of the operation in the tree.

4.5 *RCU-HTM* Search Tree Examples

In this section we present two example implementations of *RCU-HTM* based trees. The first is an internal AVL binary search tree and the second one is a B+-tree. The full C++ code of the two search trees along with all the other *RCU-HTM* trees can be found in our library's github repo [1].

4.5.1 Example 1: Internal AVL binary search tree

The code for the internal AVL binary search tree is presented in Listing 4.5.

Traverse_with_stack() follows the classic tree traversal procedure with the only addition that we store the traversed nodes in the *stack* array as well as the direction that was taken on each

[1] `https://www.github.com/jimsiak/concurrent-maps-cpp`

node (i.e., left or right child pointer) in the *stack_indexes* array. *stack_top* indicates the length of the traversal path.

Install_copy() is also very simple. It just checks whether *connection_point* is NULL. If this is the case, the root of the tree needs to be updated to point to the private copy. Otherwise, the private copy is attached as the appropriate child of the node that is the *connection_point*.

Validate_copy() performs the validation of the nodes to be replaced. The validation in our implementation is split in two phases. In the first one in lines 44–50 we validate that the access path of the operation has not been modified. This ensures that no other concurrent operation has in the meanwhile removed any of the nodes in this path. The second validation in lines 52–59 ensures that all the nodes that have been added in the validation set have also remained unmodified. We implement the validation set as a simple statically sized hash table, with *HT_LEN* buckets each of a predefined size. For all our search trees we found that a hash table with 16 64-length buckets suffices. Each thread has its own hash table which is stored as the *ht* field of a per thread global structure called *tdata*.

Insert_with_copy() proceeds in the following way. It initiates the private copy with a new node that contains the key-value pair (lines 70–73). It then traverses the tree in the opposite direction (i.e., from the reached node up to the root) to perform the appropriate rebalancing actions. At each step of this reverse traversal, it copies the current node and attaches this copy to the private copy (lines 90–98). Then it performs any necessary rotation on the private copy (lines 115–139). The reverse traversal ends in three cases: a) when it reached the root of the tree (lines 83–84), b) when it encounters a node whose height has not been updated (lines 86–88), and, c) when a rotation has been performed which means that no further rebalancing is required (lines 126 and 138). When the operation is finished *privcopy* points to the new modified version of the affected part of the tree.

```
1 V traverse_with_stack(K& key, void **stack, int *stack_indexes, int *stack_top)
2 {
3   node_t *parent, *leaf;
4   node_t **node_stack = (node_t **)stack;
5
6   parent = NULL;
7   leaf = root;
8   *stack_top = -1;
9
10  while (leaf) {
11    node_stack[++(*stack_top)] = leaf;
12    stack_indexes[*stack_top] = (key <= leaf->key) ? 0 : 1;
13
14    if (leaf->key == key) break;
15    parent = leaf;
16    leaf = (key < leaf->key) ? leaf->left : leaf->right;
17  }
18
19  if (*stack_top >= 0 && node_stack[*stack_top]->key == key)
20    return node_stack[*stack_top]->value;
21  else
22    return NO_VALUE;
23 }
24
```

```
void install_copy(void *connpoint, void *privcopy, int *stack_indexes, int connpoint_stack_index)
{
  node_t *connection_point = (node_t *)connpoint;
  node_t *tree_copy_root = (node_t *)privcopy;
  if (connection_point == NULL) {
    root = tree_copy_root;
  } else {
    int index = stack_indexes[connpoint_stack_index];
    if (index == 0) connection_point->left = tree_copy_root;
    else            connection_point->right = tree_copy_root;
  }

}

void validate_copy(void **stack, int *stack_indexes, int stack_top)
{
  node_t **node_stack = (node_t **)stack;
  node_t *n1, *n2;

  for (int i=0; i < stack_top; i++) {
    n1 = node_stack[i];
    int index = stack_indexes[i];
    n2 = (node_t *)((index == 0) ? n1->left : n1->right);
    if (n2 != node_stack[i+1])
        TX_ABORT(ABORT_VALIDATION_FAILURE);
  }

  for (int i=0; i < HT_LEN; i++) {
    for (int j=0; j < tdata->ht->bucket_next_index[i]; j+=2) {
      node_t **np = (node_t **)tdata->ht->entries[i][j];
      node_t  *n  = (node_t *)tdata->ht->entries[i][j+1];
      if (*np != n)
        TX_ABORT(ABORT_VALIDATION_FAILURE);
    }
  }

}

void *insert_with_copy(K& key, V& value, void **stack,
                       int *stack_indexes, int stack_top, void **privcopy,
                       int *connpoint_stack_index)
{
  node_t *tree_copy_root, *connection_point;
  node_t **node_stack = (node_t **)stack;

  //> Start the tree copying with the new node.
  tree_copy_root = new node_t(key, value);
  *connpoint_stack_index = stack_top;
  connection_point = stack_top >= 0 ? node_stack[stack_top--] : NULL;

  //> Empty tree case
  if (stack_top < 0) {
    *privcopy = (void *)tree_copy_root;
    return (void *)connection_point;
  }

  while (stack_top >= -1) {
    //> If we've reached and passed root return.
    if (!connection_point)
      break;

    //> If no height change occurs we can break.
    if (tree_copy_root->height + 1 <= connection_point->height)
      break;

    //> Copy the current node and link it to the local copy.
    node_t *curr_cp = node_new_copy(connection_point);
    ht_insert(tdata->ht, &connection_point->left, curr_cp->left);
    ht_insert(tdata->ht, &connection_point->right, curr_cp->right);

    curr_cp->height = tree_copy_root->height + 1;
    if (key < curr_cp->key) curr_cp->left = tree_copy_root;
    else                    curr_cp->right = tree_copy_root;
    tree_copy_root = curr_cp;

    // Move one level up
    *connpoint_stack_index = stack_top;
    connection_point = stack_top >= 0 ? node_stack[stack_top--] : NULL;

    // Get current node's balance
    node_t *sibling;
    int curr_balance;
    if (key < curr_cp->key) {
      sibling = curr_cp->right;
      curr_balance = node_height(curr_cp->left) - node_height(sibling);
    } else {
      sibling = curr_cp->left;
      curr_balance = node_height(sibling) - node_height(curr_cp->right);
    }

    if (curr_balance == 2) {
      int balance2 = node_balance(tree_copy_root->left);

      if (balance2 == 1) { // LEFT-LEFT case
        tree_copy_root = rotate_right(tree_copy_root);
      } else if (balance2 == -1) { // LEFT-RIGHT case
        tree_copy_root->left = rotate_left(tree_copy_root->left);
        tree_copy_root = rotate_right(tree_copy_root);
      } else {
        assert(0);
      }
      break;
    } else if (curr_balance == -2) {
      int balance2 = node_balance(tree_copy_root->right);

      if (balance2 == -1) { // RIGHT-RIGHT case
        tree_copy_root = rotate_left(tree_copy_root);
      } else if (balance2 == 1) { // RIGHT-LEFT case
        tree_copy_root->right = rotate_right(tree_copy_root->right);
        tree_copy_root = rotate_left(tree_copy_root);
      } else {
        assert(0);
      }
      break;
    }
  }

  *privcopy = (void *)tree_copy_root;
  return (void *)connection_point;
}
```

Listing 4.5: The methods necessary to apply *RCU-HTM* for an internal AVL binary search tree. The *remove_with_copy()* method is implemented in a similar fashion to *insert_with_copy()* and we omit it here for brevity.

4.5.2 Example 2: B+-tree

The code for the B+-tree is presented in Listing 4.6. *Install_copy()* and *validate_copy()* are identical to the respective methods in the internal AVL trees. In fact, these two methods could be part of the generic code of *RCU-HTM*, instead of being search tree specific. However, the *RCU-HTM* base class would then need to be aware of the internal structure of the node of the search tree it is applied to. Therefore, to keep the *RCU-HTM* base class as generic as possible, we leave these two methods in the implementation of each different search tree.

Similarly to the internal AVL tree, *traverse_with_stack()* here follows the classic B+-tree traversal procedure; the only addition is that we store the traversed nodes in the *stack* array together with the direction that was taken on each node (i.e., the index of the child that was chosen on each node) in the *stack_indexes* array. *stack_top* indicates the length of the traversal path.

In the B+-tree, *insert_with_copy()* is simpler than the one in the AVL case, because here rebalancing is done only when a node is full. This method performs a reverse traversal of the tree until it either reaches the root (lines 76–83) or encounters a node where splitting is not necessary (lines 97–102). While reverse traversing the tree, the appropriate nodes are copied and attached to the private copy (lines 88–92).

```
 1 V traverse_with_stack(K& key, void **stack, int *stack_indexes, int *stack_top)
 2 {
 3   node_t **node_stack = (node_t **)stack;
 4   int index;
 5   node_t *n;
 6 
 7   *stack_top = -1;
 8   n = root;
 9   if (!n) return NO_VALUE;
10 
11   while (!n->leaf) {
12     index = n->search(key);
13     node_stack[++(*stack_top)] = n;
14     stack_indexes[*stack_top] = index;
15     n = (node_t *)n->children[index];
16   }
17   index = n->search(key);
18   node_stack[++(*stack_top)] = n; stack_indexes[*stack_top] = index;
19 
20   if (*stack_top >= 0 && index < n->no_keys && n->keys[index] == key)
21     return (V)n->children[index+1];
22   else
23     return NO_VALUE;
24 }
25 
26 void install_copy(void *connpoint, void *privcopy, int *stack_indexes, int connpoint_stack_index)
27 {
28   node_t *connection_point = (node_t *)connpoint;
29   node_t *tree_copy_root = (node_t *)privcopy;
30   if (connection_point == NULL) {
31     root = tree_copy_root;
32   } else {
33     int index = stack_indexes[connpoint_stack_index];
34     connpoint->children[index] = tree_copy_root;
35   }
36 }
37 
38 void validate_copy(void **stack, int *stack_indexes, int stack_top)
39 {
40   node_t **node_stack = (node_t **)stack;
41   node_t *n1, *n2;
42 
43   for (int i=0; i < stack_top; i++) {
44     n1 = node_stack[i];
45     int index = stack_indexes[i];
46     n2 = (node_t *)n1->children[index];
47     if (n2 != node_stack[i+1])
48       TX_ABORT(ABORT_VALIDATION_FAILURE);
```

```
  }

  for (int i=0; i < HT_LEN; i++) {
    for (int j=0; j < tdata->ht->bucket_next_index[i]; j+=2) {
      node_t **np = (node_t **)tdata->ht->entries[i][j];
      node_t  *n  = (node_t *)tdata->ht->entries[i][j+1];
      if (*np != n)
        TX_ABORT(ABORT_VALIDATION_FAILURE);
    }
  }
}

void *insert_with_copy(K& key, V& val, void **stack,
                       int *stack_indexes, int stack_top, void **privcopy,
                       int *connpoint_stack_index)
{
  node_t **tree_copy_root = (node_t **)privcopy;
  node_t **node_stack = (node_t **)stack;

  node_t *cur = NULL, *cur_cp = NULL, *cur_cp_prev;
  node_t *connection_point;
  int index, i;
  K key_to_add = key;
  void *ptr_to_add = val;

  while (1) {
    //> We surpassed the root. New root needs to be created.
    if (stack_top < 0) {
      node_t *new_node = new node_t();
      new_node->insert_index(0, key_to_add, ptr_to_add);
      new_node->children[0] = cur_cp;
      *tree_copy_root = new_node;
      break;
    }

    cur = node_stack[stack_top];
    index = stack_indexes[stack_top];

    //> Copy current node
    cur_cp_prev = cur_cp;
    cur_cp = cur->copy();
    for (i=0; i <= cur_cp->no_keys; i++)
      ht_insert(tdata->ht, &cur->children[i], cur_cp->children[i]);

    //> Connect copied node with the rest of the copied tree.
    if (cur_cp_prev) cur_cp->children[index] = cur_cp_prev;

    //> No split required.
    if (cur_cp->no_keys < 2 * BTREE_NODE_DEGREE) {
      cur_cp->insert_index(index, key_to_add, ptr_to_add);
      *tree_copy_root = cur_cp;
      break;
    }

    ptr_to_add = cur_cp->split(key_to_add, ptr_to_add, index, &key_to_add);

    stack_top--;
  }

  *connpoint_stack_index = stack_top - 1;
  connection_point = stack_top <= 0 ? NULL : node_stack[stack_top-1];
  return (void *)connection_point;
}
```

Listing 4.6: The methods necessary to apply *RCU-HTM* for a B+-tree. The *remove_with_copy()* method is implemented in a similar fashion to *insert_with_copy()* and we omit it here for brevity.

4.6 Memory Reclamation

Several approaches have been proposed to reclaim the memory used by concurrent data structures that allow optimistic readers, i.e., where concurrent threads read parts of the data structure which may have been deleted by other threads in the meanwhile [MS98, Mic02, Fra04, HLMM05, HMBW07, Bro15, BGHZ16, WIC^{+}18]. This is a challenging task which is typically considered orthogonal to the research around concurrent data structures and is usually omitted from related work. However, since *RCU-HTM* stresses memory with its node copies, we augment our approach with memory reclamation to validate that any benefit from *RCU-HTM* is not cancelled by excessive memory use. In this section we explain how an epoch-based reclamation scheme can be integrated with *RCU-HTM* without compromising its high performance, as our experimental results validate.

More specifically, we integrate DEBRA [Bro15], a state-of-the-art epoch-based reclamation technique with low overhead. Alternative techniques like EBR [Fra04], QSBR [MS98], QSense [BGHZ16] and IBR [WIC^{+}18] can also be integrated to *RCU-HTM*. In future work we intend to integrate and evaluate these techniques as well. To make this work more self-contained, we describe the necessary concepts of DEBRA herein. DEBRA defines four operations: *leaveQstate()*, *enterQstate()*, *retire()* and *isQuiescent()*. *leaveQstate()* is invoked before the execution of each tree operation and *enterQstate()* at the end of the operation. These two function invocations define the end and start of a thread's *quiescent period* respectively. A thread *t* is quiescent when it does not execute some tree operation, thus not accessing any tree nodes. In DEBRA each thread has a *quiescent bit* which indicates whether the thread is in a quiescent state or not.

An epoch is defined as a time window during which each and every thread has been in a quiescent state at least once. To keep track of epochs, DEBRA has a global variable which stores the current epoch number *e*. Whenever a thread leaves a quiescent state, it reads *e* and stores it in a local variable. Next, it attempts to determine whether the global epoch *e* can be advanced, which is the case if each thread is in a quiescent state or its locally stored epoch is less than or equal to *e*. Thread *t* incrementally checks all the other threads, amortizing the cost over n *leaveQstate()* invocations. A local variable *checked* keeps track of the number of threads that *t* has already checked and once all threads are checked, thread *t* performs a CAS to increment the current epoch.

To remove a node from the tree, a thread *t* invokes the *retire()* function. Each thread has three private *bags* of nodes that have been recently removed from the tree. At any point, one of these bags is the *currentBag* and whenever *t* removes a node from the tree, it adds it to the currentBag. When *t* observes an epoch change (i.e., its locally stored epoch is less than the global one) it changes its currentBag to the oldest of the three bags. The previous contents of this bag can be freed to the operating system since it is guaranteed that no other thread holds references on them. More details about the correctness of DEBRA can be found in the original paper [Bro15].

The bags of tree nodes that each thread occupies are implemented as singly-linked lists of *blocks*. Each block contains up to B pointers to tree nodes and a next pointer that points to its successive block in the list. In our experiments we have set B to 256 as also indicated by the authors in the paper [Bro15]. Moreover, each thread has a private *allocation bag* which is used for allocating tree nodes. When the contents of a reclamation bag are safe to be freed (i.e., when the epoch has advanced) we move all its blocks to the allocation bag to allow the nodes that they include to be re-used. If the allocation bag exceeds a threshold we free its contents. In our experiments we have set this threshold to 6MB per thread. When allocating a tree node, if the allocation bag is empty we allocate more memory from the operating system.

CHAPTER 5

Experimental Evaluation

5.1 Experimental Setup

We conduct all our experiments on an Intel Broadwell-EP server with an Intel Xeon E5-2699 v4 processor with 22 physical cores and 44 hardware threads. We set the processor to run at a fixed frequency of 2.2GHz with TurboBoost mode disabled. Each core has private 32KB L1 and 256KB L2 caches, while a 56MB L3 cache is shared by all cores. The server has 256GB of RAM running at 2134MHz. The OS is Debian 8.3 with kernel version 4.7.0. In order to manage hardware transactions, we use the processor's transactional synchronization extensions (TSX) [1]. We only use the restricted transactional memory (RTM) mode of TSX, which provides more flexibility and allows us to retry a transaction a number of times before resorting to the non-transactional fallback path.

5.2 Search Tree Implementations

Our library of search trees includes several implementations which can be found in the author's github page [2]. We use 9 serial search trees as our baseline, which are shown in Table 5.1, and we implement concurrent versions of them using the aforementioned synchronization mechanisms,

[1]https://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions
[2]https://www.github.com/jimsiak/concurrent-maps-cpp

Serial Search Trees				
#	**Name**	**Node Type**	**Balancing**	**Type**
1	*btree*	B-tree	Yes	External
2	*abtree*	B-tree	Relaxed	External
3	*bst-unb-int*	Binary	Unbalanced	Internal
4	*bst-unb-pext*	Binary	Unbalanced	Partially External
5	*bst-unb-ext*	Binary	Unbalanced	External
6	*bst-avl-int*	Binary	AVL	Internal
7	*bst-avl-pext*	Binary	AVL	Partially External
8	*bst-avl-ext*	Binary	AVL	External
9	*treap*	Binary internal / Fat leaves	Unbalanced	External

Table 5.1: Baseline serial search trees used in our experimental evaluation.

including *RCU-HTM*. The full list of concurrent search trees used in our experiments are presented in Table 5.2. Coarse-grained locking and HTM mechanisms are applied on all the serial implementations. The same is true for RCU with coarse-grained locking and for our *RCU-HTM* mechanism, yet we do not include the *RCU-HTM* treap in our evaluation since it does not provide any additional useful insights.

5.3 Benchmarks

We use three benchmarks; one artificial microbenchmark and two macrobenchmarks which resemble the way that concurrent search trees are used as indexes on database systems. For all benchmarks the following are true:

- We pin each software thread on a dedicated hardware thread in such a way as to first utilize all the physical cores of the machine, i.e., up to 22 threads, and only for more than 22 threads we utilize the processor's hyperthreads.
- We validate that the tree structure is in a consistent state, after the end of any benchmark execution, by checking first that the keys are ordered properly and then performing any other appropriate structural checks depending on the type of the tree (e.g., in an AVL tree we check whether the AVL variants still hold for every node). Moreover, we validate that the number of elements in the tree is correct given the total number of insertions and deletions that were performed.
- We use the scalable memory allocator call jemalloc [3] since the standard libc memory allocator does not perform well under multi-threaded workloads.

[3]`http://jemalloc.net/`

Name	**#serial**	**Node Type**	**Balancing**	**Type**
Coarse-grained Locking				
One version for each serial search tree.				
Contention-Adapting Locking				
bst-unb-int-ca-locks	3	Binary	Unbalanced	Internal
bst-unb-pext-ca-locks	4	Binary	Unbalanced	Partially External
bst-unb-ext-ca-locks	5	Binary	Unbalanced	External
treap-ca-locks [SW15]	9	Binary internal / Fat leaves	Unbalanced	External
Fine-grained Locking				
bst-unb-hohlocks	5	Binary	Unbalanced	External
avl-int-drachsler [DVY14]	6	Binary	Relaxed AVL	Internal
avl-pext-bronson [BCCO10]	7	Binary	Relaxed AVL	Partially External
avl-pext-cf [CGR13]	7	Binary	Relaxed AVL	Partially External
Lock-Free				
abtree-llxscx [Bro17]	2	B-tree	Relaxed	External
ist-brown [BPA20]	2	B-tree	Relaxed	External
bst-ext-natarajan [NM14]	5	Binary	Unbalanced	External
bst-ext-ellen [EFRvB10]	5	Binary	Unbalanced	External
bst-ext-llxscx [Bro17]	5	Binary	Unbalanced	External
RCU with coarse-grained locks				
One version for each serial search tree, except from 9.				
RCU with fine-grained locks				
rcu-fgl [AA14]	3	Binary	Unbalanced	Internal
Coarse-grained HTM				
One version for each serial search tree.				
3-Path HTM				
abtree-3path [Bro17]	2	B-tree	Relaxed	External
bst-ext-3path [Bro17]	5	Binary	Unbalanced	External
COP-HTM				
avl-int-cop [AK14]	6	Binary	AVL	Internal
avl-ext-cop [AK14]	8	Binary	AVL	External
RCU-HTM				
One version for each serial search tree, except from 9.				

Table 5.2: Concurrent serial search trees used in our experimental evaluation. We also experimented with coarse-grained locking trees with reader-writer locks, but the results were very similar to when using a spinlock. In total, our evaluation includes 52 concurrent search trees.

- We perform no memory reclamation during our experiments, similarly to prior work [AA14, NM14, Bro17] on concurrent data structures. In Section 5.6 we evalute the performance of our *RCU-HTM* trees when an epoch-based memory reclamation scheme is applied.

- We run every experiment in our evaluation 10 independent times and report the geometric mean. We did not observe a significant variance for any of our experiments.

Microbenchmark parameters	
Parameter	**Values**
Size of keys	8, 64 and 256 bytes
Number of keys in the tree	100, 1K, 10K, 1M, 10M keys
Mix of operations	100-0-0, 90-5-5, 80-10-10, 50-25-25, 20-40-40, 0-50-50
Number of threads	1, 2, 4, 8, 16, 22, 44
Total execution scenarios	$3 * 5 * 6 * 7 = \mathbf{630}$

Table 5.3: The parameters of our artificial microbenchmark. In the operations mix parameter, l-i-r indicates percentage of *lookup()*, *insert()* and *remove()* operations respectively.

5.3.1 Microbenchmark

We have implemented an artificial microbenchmark to evaluate the performance and scalability of the concurrent search trees using various configurations of key size, number of keys in the tree, mix of operations and number of threads, for a total of 630 different execution scenarios for each search tree. The various configurations of these parameters are given in Table 5.3. Although this is not a real-life benchmark, it allows the analysis and evaluation of the different search trees under various execution scenarios and is a benchmark that is very commonly used in related work on concurrent data structures [BCCO10, Bro17, AM15]. The benchmark execution comprises the following:

- A warmup phase, during which a single thread inserts random keys into the tree until it is filled with half of the keys of the key range. Since, in all our operation mixes, insert and remove operations are performed with the same probability, the size of the tree does not significantly fluctuate during our executions.
- An execution phase, during which we spawn the worker threads, which repeatedly perform lookup, insert or remove operations with randomly selected keys. The execution phase lasts for a predefined time duration, which we currently set to 5 seconds. We have validated that longer time durations produce similar results.

5.3.2 TPC-C

TPC-C is a realistic on-line transaction processing (OLTP) benchmark [4]. It simulates the operation of an order-entry system where a population of terminal operators executes transactions against a database. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. We use the TPC-C implementation of the DBx1000 database management system [YBP$^+$14] [5] and have modified their code to use our concurrent search trees as index to the tables.

[4] `http://www.tpc.org/tpcc/`

[5] `https://github.com/yxymit/DBx1000`

TPC-C tables and indexes			
Table Name	**Index Name**	**#elements**	**Access pattern**
Warehouse	WAREHOUSE_IDX	W	Read-only
District	DISTRICT_IDX	$W * 10$	Read-only
Customer	CUSTOMER_ID_IDX	$W * 30K$	Read-only
	CUSTOMER_LAST_IDX	$W * 30K$	Read-only
Item	ITEM_IDX	$100K$	Read-only
Stock	STOCK_IDX	$W * 100K$	Read-only
Order-line	ORDERLINE_IDX	$\geq W * 300K$	Insert-only
	ORDERLINE_WD_IDX	$\geq W * 300K$	Insert-only
Order	ORDER_IDX	$\geq W * 30K$	Insert-only
New-order	NEWORDER_IDX	-	Unused
History	-	-	Unused

TPC-C transactions			
Transaction	**Index accesses**	**Transaction**	**Index accesses**
Payment	read from WAREHOUSE_IDX read from DISTRICT_IDX read from CUSTOMER_ID_IDX or read from CUSTOMER_LAST_IDX	*NewOrder*	read from WAREHOUSE_IDX read from CUSTOMER_ID_IDX read from DISTRICT_IDX for 10-15 times: read from ITEM_IDX read from STOCK_IDX insert into ORDER_IDX insert into ORDERLINE_IDX insert into ORDERLINE_WD_IDX

Table 5.4: The tables with their respective indexes and the transactions executed in the TPC-C benchmark. In our experiments we set W to 10.

The database consists of 9 tables for which 10 indexes are created. The complete list of tables, indexes and the transactions that are executed in our implementation are presented in Table 5.4. This is a very insightfull benchmark for our experimental evaluation for two reasons; first, it is a classic database benchmark and closely resembles a real-life application, and, second, the different access patterns and sizes of the indexes represent different kind of workloads for the underlying data structure.

5.3.3 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) [6] is a benchmark that represents a single table database with a single index on top of it. The information about the table, the index and the executed transactions is given in Table 5.5. YCSB includes a single transaction which executes only lookups in the index, so it does not provide much additional insights in our evaluation, however we included it since it is a very commonly used benchmark. Similarly to TPC-C, we use the YCSB implementations of the DBx1000 database management system [YBP+14].

[6] `https://en.wikipedia.org/wiki/YCSB`

YCSB tables and indexes			
Table Name	**Index Name**	**#elements**	**Access pattern**
MAIN_TABLE	MAIN_INDEX	24M	Read-only

YCSB transactions	
Transaction	**Index accesses**
Read-only Transaction	for 16 times: read from *MAIN_INDEX*

Table 5.5: The tables with their respective indexes and the transactions executed in the YCSB benchmark.

5.4 Experimental Results: Microbenchmark

The microbenchmark allows us to evaluate the concurrent search trees under several different execution scenarios using various combinations of the four parameters, namely, the size of the keys, the number of keys in the tree, the mix of operations and the number of concurrent threads.

We analyze our results in two directions. The first one evaluates *RCU-HTM* when the underlying baseline serial search tree is predefined. The aim of this kind of evaluation is to assess whether *RCU-HTM* can provide a concurrent version of any type of search tree without incurring high overheads when compared to the other synchronization mechanisms that can be applied to the specific serial search tree. The second direction evaluates whether *RCU-HTM* provides high performance on each and every one of the execution scenarios under consideration. Given that *RCU-HTM* can be relatively easily applied to all search trees, our aim is to validate that, for every execution scenario, there is at least one *RCU-HTM* based search tree that performs better, or at least very close to, the other alternatives.

5.4.1 Evaluation of *RCU-HTM* with a predefined baseline serial search tree

We aim to show that *RCU-HTM* is a competitive synchronization mechanism regardless of the baseline serial search tree on which it is applied. This, in tandem with the simplicity of applying *RCU-HTM* on different search trees, makes *RCU-HTM*, to the best of our knowledge, the first synchronization mechanism that is widely applicable and at the same time highly efficient.

To compare *RCU-HTM* with the other synchronization mechanisms, we find the maximum achieved throughput, for each execution point and for each baseline serial data structure, and normalize all other throughputs to this maximum. This gives us a percentage of how close to the best each implementation performs. Figure 5.1 presents the results in boxplots which summarize multiple execution scenarios.

B+-trees. Regarding B+-trees we have 5 concurrent implementations, three that use coarse-grained locking and HTM synchronization, one with RCU-SGL and one with *RCU-HTM*. In this case, *RCU-HTM* provides the most stable performance and in the majority of executions its

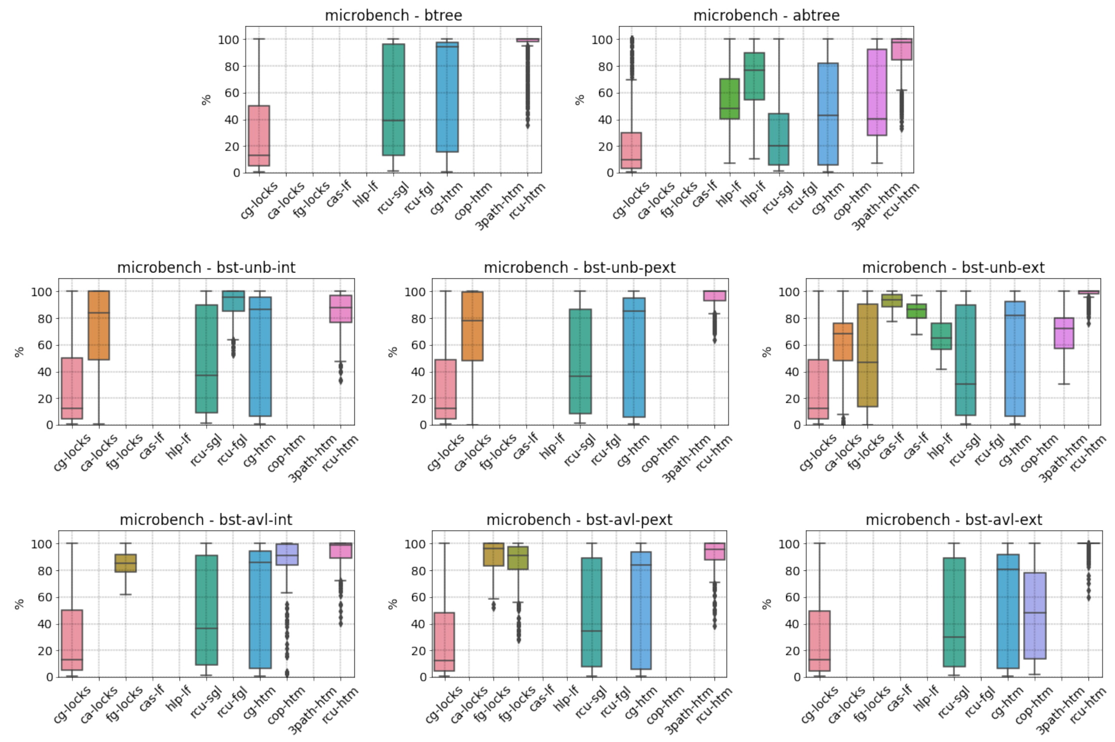


Figure 5.1: Evaluation of the synchronization mechanisms when applied to a specific baseline serial search tree. For each execution scenario we find the synchronization mechanism with the highest throughput and use it to normalize the rest throughputs. This way, the y axis shows the percentage difference of the throughput of each synchronization compared to the best achieved throughput for the specific serial search tree and execution scenario. Every boxplot include 630 execution scenarios.

throughput is more than 80% close to the best one achieved. There are some outliers where *RCU-HTM* is up to 40% close to the best; all these outlier points are executions with a small number of threads, i.e., one or two threads, where *RCU-HTM* performs worse that the coarse-grained implementations due to the overheads of copying nodes and validating the private copies. However, for more than two threads *RCU-HTM* allows more concurrency than coarse-grained B+-trees and outperforms them.

(a-b)-trees. In the case of (a-b)-trees we have more synchronization mechanisms that can be applied. Apart from the coarse-grained implementations, we also have three lock-free approaches by Brown et. al [Bro17, BPA20]. In this case, *RCU-HTM* implementations perform between 33%-100% relative to the best implementation at each execution point. Similarly to B+-trees the cases where *RCU-HTM* performs much worse than the other implementations is in cases with a small number of threads.

Unbalanced binary search trees. When applied to unbalanced versions of binary search trees *RCU-HTM* is highly performant and competitive with other synchronization mechanisms in most of the execution scenarios. The three versions of *RCU-HTM* based binary search trees (i.e., internal, partially-external and external) have throughput more than 70% in the 75% of the cases.

AVL binary search trees. In all versions of AVL binary search trees (i.e., internal, partially-external, external) *RCU-HTM* implementations are clearly among the best ones on the vast majority of execution points. In the internal version *RCU-HTM* is 40%-100% close to the best one, however, in the majority of executions it is more than 70% close. The same is true for partially-external versions. In the external AVL trees *RCU-HTM* performs even better, achieving performance more than 90% close to the best one in the majority of execution scenarios.

Overall *RCU-HTM* evaluation. Overall, *RCU-HTM* is a competitive synchronization mechanism for each and every baseline serial data structure. Other synchronization mechanisms either fail to provide robust performance over all the different execution scenarios (i.e., coarse-grained synchronization mechanisms) or are limited to specific serial data structures and are hard to be implemented (i.e., fine-grained synchronization mechanisms). *RCU-HTM* is a solution that is both applicable to any serial search tree and also maintains high performance on all execution scenarios.

5.4.2 Overall scalability evaluation of *RCU-HTM*

In the previous section we compared *RCU-HTM* with the other synchronization mechanisms when the baseline serial data structure was predefined. In this section we perform an overall scalability evaluation to show that by exploiting *RCU-HTM*, we can provide an efficient concurrent search for every execution scenario.

Figures 5.2 and 5.3 present the performance of the five best performing search trees for different combinations of the size of key, number of keys in the tree and operations mix. We omit some of the plots for brevity as they do not provide any additional insights. As all search trees scale at least up to 22 threads, we nominate as best performing implementations those that exhibit the highest throughput for 22 threads. We make the following observations:

1 *RCU-HTM* is among the five best performing trees in the majority of cases. No other competitive concurrent search tree or other synchronization mechanism is more consistently among the top five implementations. The only cases where *RCU-HTM* is not among the best implementations are in the 64 byte keys with 0% lookups. In these situations the size of nodes is increased and copying such large nodes slows down the performance of *RCU-HTM*. However, since keys are immutable in *RCU-HTM*, it is possible to avoid storing the keys in the node, by storing pointers to these keys. This way, we can decrease the size of the node and avoid the

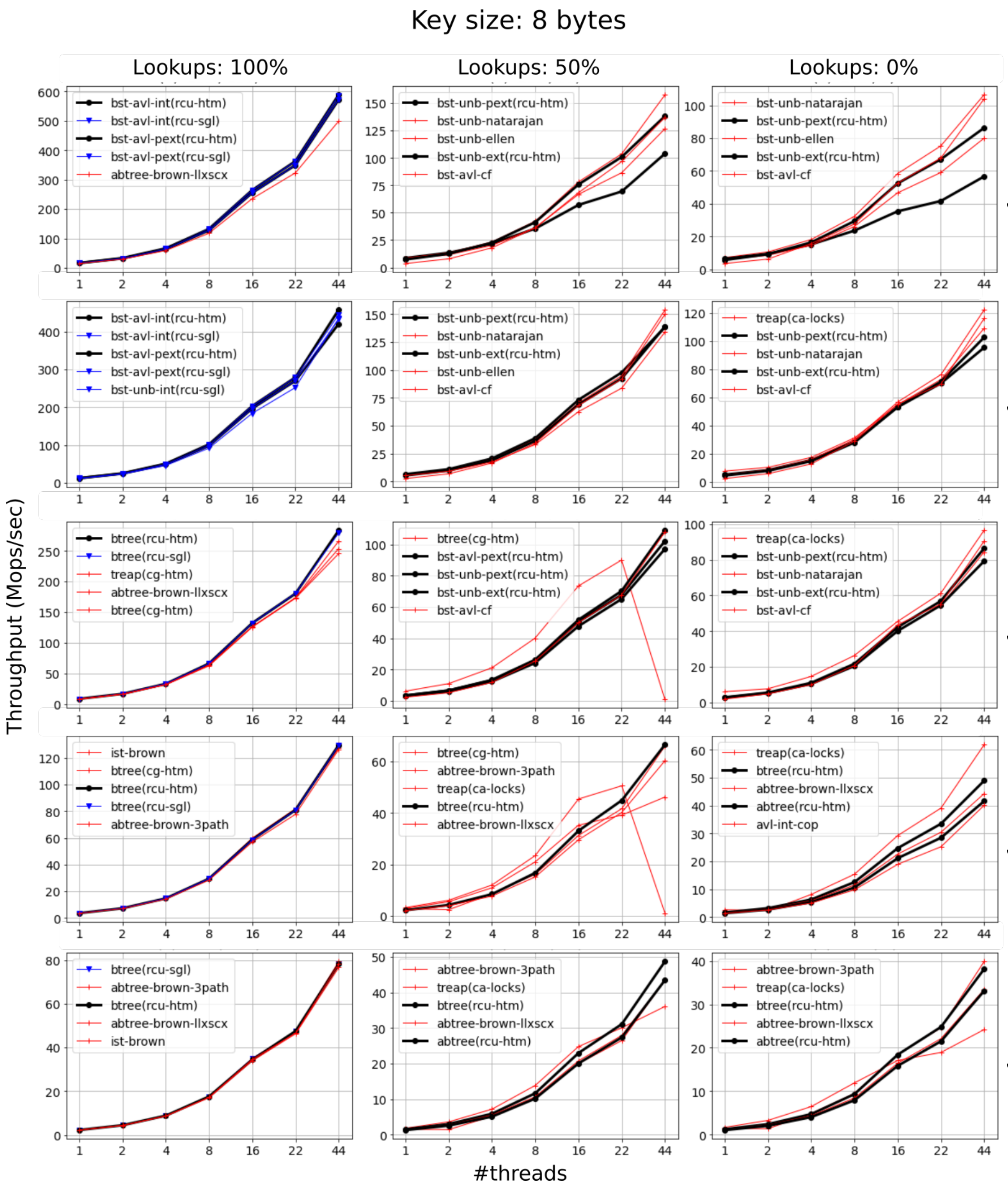


Figure 5.2: Scalability evaluation of concurrent search trees with 8 bytes key size. Each plot includes the five best performing search trees. The rows represent different tree sizes and the columns different mix of operations. Notice the differences in the y-axis range between the figures.

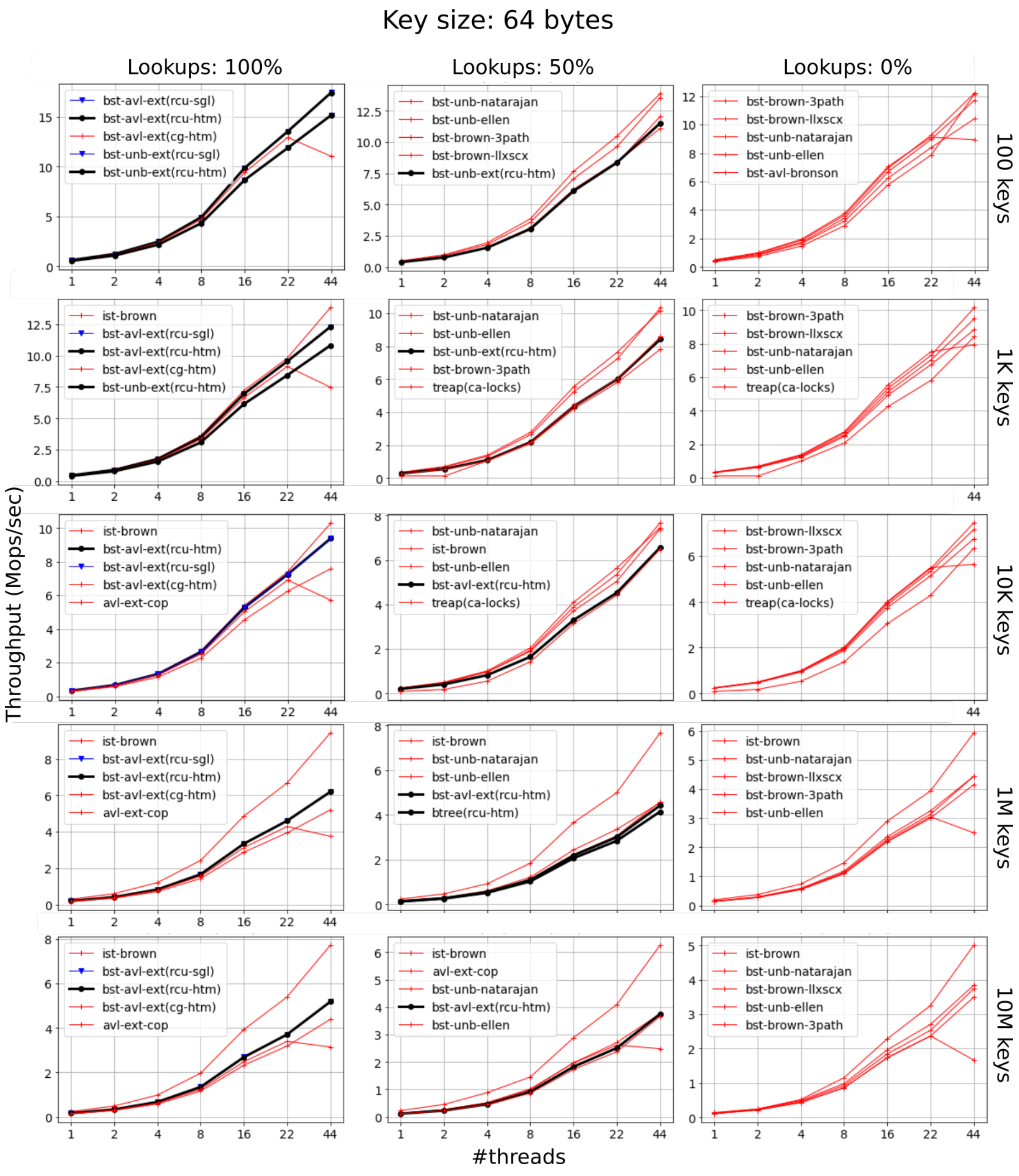


Figure 5.3: Scalability evaluation of concurrent search trees with 64 bytes key size. Each plot includes the five best performing search trees. The rows represent different tree sizes and the columns different mix of operations. Notice the differences in the y-axis range between the figures.

overheads of copying the whole keys. Some preliminary experiments have shown that this can greatly reduce these performance overheads.

2 In read-only scenarios, with 100% lookup operations, *RCU-HTM* either provides the best performance or very close to it. Only in the cases with 64 bytes keys for 1M and 10M keys it is somewhat slower than ist-brown. The reason for this is that ist-brown uses interpolation search inside the nodes, and thus performs fewer key comparisons on average. It is trivial to use interpolation search in our *RCU-HTM* trees and we indeed tried this and got similar throughput to ist-brown. We do not report these results in this thesis since we consider such kind of optimizations orthogonal to *RCU-HTM*.

3 In read-write scenarios, *RCU-HTM* maintains its very high performance. In the five cases with 64 bytes keys and 0% lookups where it is not among the top five implementations, it still is competitive and performs at least 60% close to the best implementations in all execution scenarios.

Overall, our scalability results validate that for every execution scenario we can exploit *RCU-HTM* and implement a concurrent search tree with very high performance. This makes *RCU-HTM*, to the best of our knowledge, the first synchronization mechanism that manages to combine programmability/applicability with high performance for a wide spectrum of execution scenarios.

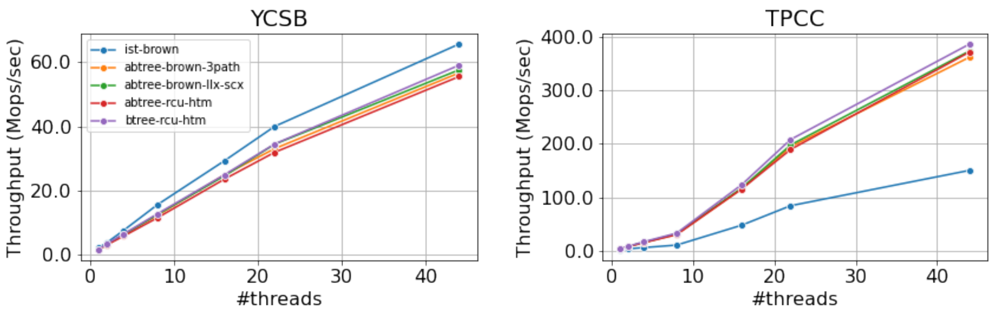


Figure 5.4: Performance of concurrent B+-trees and (a-b)-trees for YCSB and TPC-C benchmarks.

5.5 Experimental Results: TPC-C and YCSB

In this section we evaluate *RCU-HTM* in two benchmarks that resemble real-life database applications and are widely used for the evaluation of database management systems. In both benchmarks the keys that are stored in the trees are 8 bytes long. Figure 5.4 presents the performance of lock-free and *RCU-HTM* based B+-trees and (a-b)-trees since these are the best ones in both benchmarks.

In YCSB, where only lookups are performed in a search tree that contains 25M keys, ist-brown performs slightly better than *RCU-HTM* B+-tree thanks to the interpolation search that we described earlier. However, even without the interpolation search optimization, *RCU-HTM* B+-tree achieves a throughput of 60 Mops/sec, very close to the 64 Mops/sec of ist-brown. The results in YCSB are similar to those in the microbenchmark, in the case of 10M trees and 100% lookups.

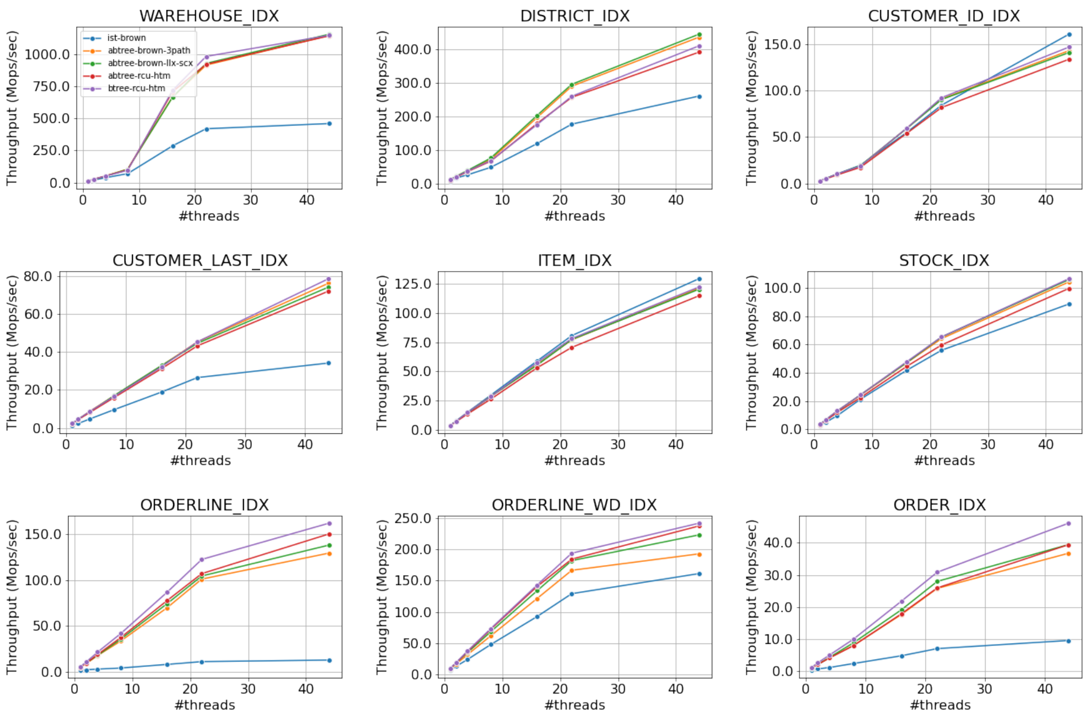


Figure 5.5: Scalability of concurrent B+-trees and (a-b)-trees for each index of the TPC-C benchmark.

In TPC-C, which uses 9 search trees with different sizes and mix of operations, the *RCU-HTM* B+-tree provides the highest throughput. The more complex access pattern of TPC-C requires a search tree that consistently provides high performance across various tree sizes and mix of operations. Figure 5.5 presents the performance of concurrent B+-trees and (a-b)-trees on each index of the TPC-C benchmark. *RCU-HTM* maintains its performance on all indexes, thus leading to increased high aggregated performance for the whole benchmark. ist-brown, which was the top performing tree in YCSB, provides high performance on the three large indexes with a read-only access pattern (i.e., CUSTOMER_ID_IDX, ITEM_IDX and STOCK_IDX) but its performance degrades in small indexes (i.e., WAREHOUSE_IDX, DISTRICT_IDX and CUSTOMER_LAST_IDX) and in indexes where only insert operations are performed (i.e., ORDERLINE_IDX, ORDERLINE_WD_IDX and ORDER_IDX). This performance variation under different access patterns results in poor aggregated performance.

Overall, the evaluation with YCSB and TPC-C validates that *RCU-HTM* can be used in real-life applications and provides high performance for a wide variety of execution scenarios. It is consistently among the top performing implementations and this results in high performance even in applications that have multiple search trees with different access pattern for each one, as was the case for the TPC-C benchmark.

5.6 Experimental Results: Memory Reclamation

In this section we evaluate the performance of *RCU-HTM* with the DEBRA memory reclamation scheme integrated. Although, we have applied DEBRA to all *RCU-HTM* based search trees, we limit our evaluation here to an internal AVL tree, namely *avl-int-rcu-htm* as for all other cases we drew similar conclusions. Figure 5.6 presents the performance of avl-int-rcu-htm with and without memory reclamation enabled for three tree sizes and three operation mixes. The two omitted tree sizes (2K and 2M keys) do not offer any different conclusion. As already mentioned, each thread occupies up to 6MB of space at any time.

As expected, and visualized by read-only workloads, lookup operations are only slightly affected by the reclamation scheme, since they do not allocate or reclaim any nodes; the only action they perform is to modify the quiescent bit and, infrequently, increase the global epoch counter. In the other two workloads, with update operations taking place, the reclamation scheme adds overhead which reaches up to 30% in the case of 20K keys with 0% lookups. However, as the figure showcases, in most cases the tree with the memory reclamation scheme enabled performs very close to the one without it.

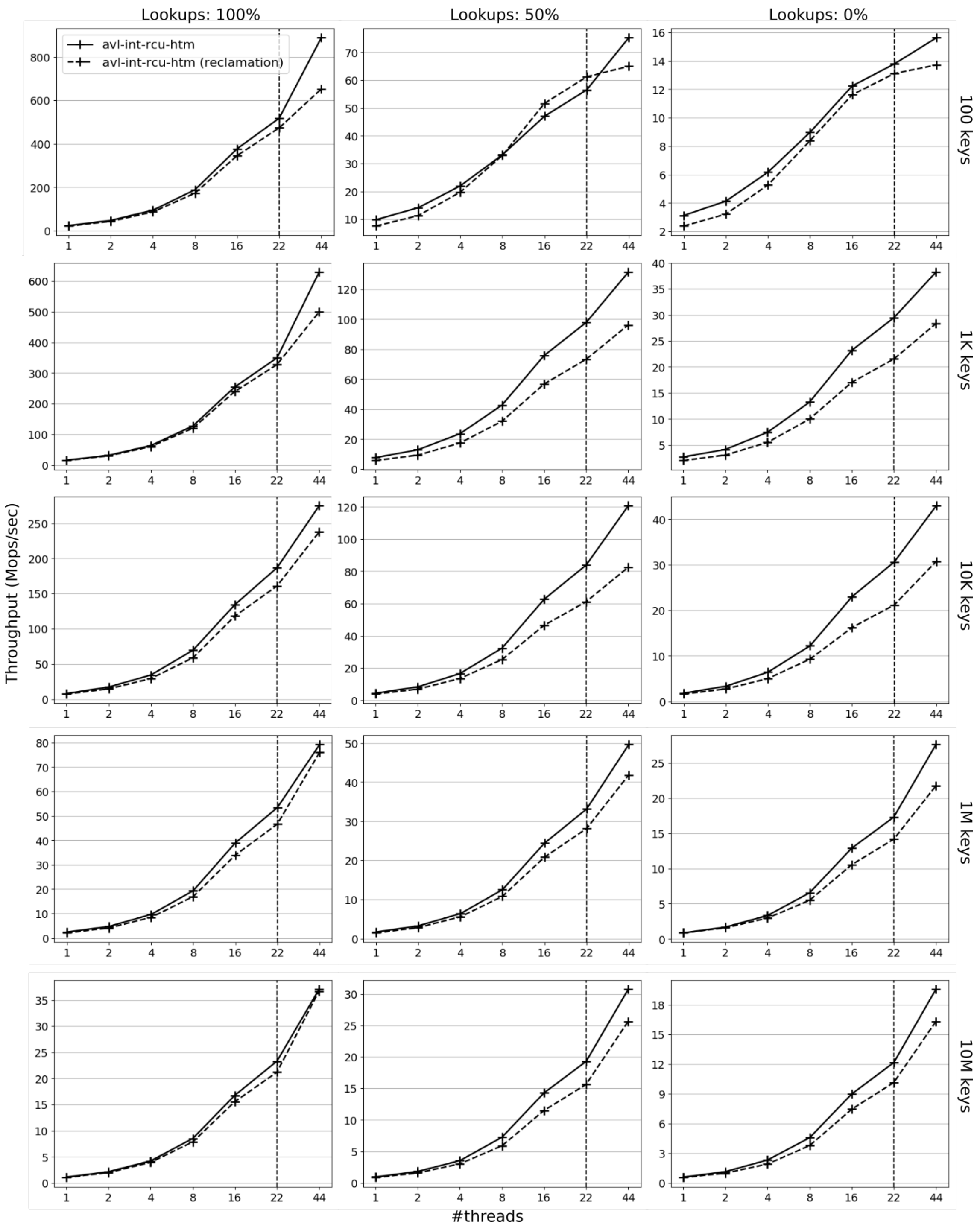


Figure 5.6: Performance of avl-int-rcu-htm when memory reclamation is performed.

CHAPTER 6

Range Query Operations in RCU-HTM

There has been plenty of research around concurrent maps, however most of the related work lacks support for range queries (RQs). Only a rather small number of concurrent maps provide linearizable RQs [Cha17, ASS13, BCCO10, SW16, Win17, BBB^{+}17, BA12]. The design and implementation of a concurrent map that supports RQs is challenging due to the difficulty to guarantee their correct execution when interleaved with concurrent update operations, i.e., inserts and deletes. The results of an RQ typically span several parts of the underlying data structure and the access to all of them needs to use some kind of synchronization, such as locks and/or atomic operations. Concurrent maps that use fine-grained locking are hard to efficiently support RQs due to the high overheads of obtaining locks on all the nodes included in the range. Non-blocking approaches, which use atomic operations such as Compare-And-Swap (CAS), are even more challenging since the hardware provided atomic operations can only be used to synchronize the access to a very limited number of memory locations, typically one or two. Transactional memory (TM) [HM93] and Read-Copy-Update (RCU) [MS98] can facilitate the implementation of RQs, however, they both incur high overheads both on the RQs and on the update operations.

RCU-HTM can be easily extended to support efficient concurrent RQs. In this chapter we augment an *RCU-HTM* based B+-tree with support for RQs and show that along with its performance benefits, *RCU-HTM* also greatly facilitates the support for RQs. This is achieved thanks to the use of node copying for performing the B+-tree modifications and through the use of HTM that allows multiple memory operations (reads and/or writes) to be performed in a single atomic

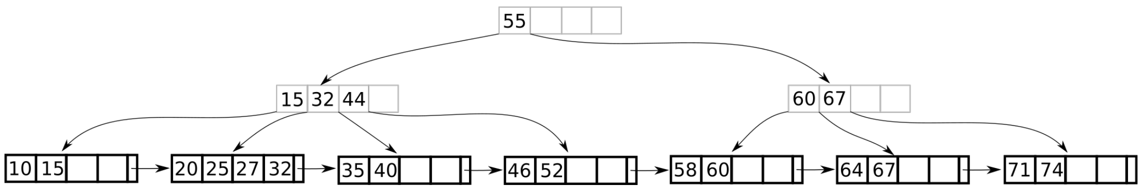


Figure 6.1: Example of a B+-tree. Only a part of the tree is depicted. Gray nodes are the internal nodes that are used only for directing traversals to the appropriate leaves.

step. With *RCU-HTM*, updaters work on copies of the affected parts of the underlying data structure rather than modifying them in place. When their modified copy is ready they install it in the shared data structure in a single atomic step using an HTM transaction. This allows readers to proceed safely without using any synchronization. As we show in our work, this also allows RQs to use an HTM transaction to quickly get a snapshot of the leaf nodes that are to be included in the RQ's result.

In a nutshell, an RQ in our *RCU-HTM* based B+-tree proceeds with the following steps. First, we traverse the tree until we reach the leaf node that includes the lowest key in the searched range. Then, we start an HTM transaction, which uses the leaves' sibling pointers to locate all the leaves that contain keys inside the range. During this transaction we only store pointers to these nodes and we do not have to copy them, since the keys of a node in our tree never change (when a node's key needs to change a copy is created which replaces the old node). As our evaluation reveals, apart from their simplicity, RQs in our B+-tree provide high performance even under workloads with high percentage of update operations. More specifically, we evaluate our B+-tree implementation under different execution scenarios and against state-of-the-art concurrent maps that also support RQs. We find that *RCU-HTM* greatly facilitates the implementation of linearizable and efficient RQs.

6.1 Background

6.1.1 Range Queries

A range query (RQ) operation in a map data structure returns the set of key-value pairs whose key is between a range of keys $[lowKey, highKey]$. RQs are typically met and are of significant importance in database and key-value store systems. Maps with RQ support can be implemented with a wide variety of underlying data structures, such as hash tables, singly-linked lists, skiplists, binary search trees, B+-trees, etc. Each data structure has different performance characteristics regarding RQs. Hash tables, for example, achieve low performance because the key-value pairs are not kept in sorted order, so an RQ translates into a lookup operation for each and every key in $[lowKey, highKey]$[1].

Data structures like lists, skiplists and trees maintain the set of key-value pair in sorted order and can support simpler and more efficient RQs. Lists and skiplists perform well when the stored set of key-value pairs is relatively small, but for large sets, trees typically provide higher performance due to the lower worst-case performance guarantees. Search trees support RQs either by performing a breadth-first traversal of the tree or by augmenting each node with a sibling pointer which leads to the node with the key that is immediately higher than this node's key and scanning this chain of sibling pointers starting from the first node with key higher or equal to $lowKey$ and ending at the last node with key lower or equal to $highKey$. External trees, which keep the key-value pairs only in the leaves and internal nodes contain only keys to be used for routing purposes, simplify the addition of sibling pointers since they only need to be added to the leaves. Moreover, trees with fat nodes, i.e., nodes that contain more than one key-value pairs, offer an advantage for RQs due to the improved locality of accesses for keys in the same node.

6.1.2 B+-trees

B+-trees are balanced external trees with fat nodes which makes them very good candidates for implementing a map with RQ support and for this they are used as indexes in several database management systems and in key-value stores. An example B+-tree is depicted in Figure 6.1. B+-trees are external trees; the data is stored in the leaves and the internal nodes contain only keys and are used for routing the traversals to the appropriate leaves. They support efficient implementations of the three basic map operations, namely, *lookup*, *insert* and *delete* as well as very fast and simple RQs. To facilitate RQs, every leaf contains a sibling pointer to reference its right sibling. RQs start with a traversal to locate the leaf that contains the first key in the range. Then, it horizontally scans the leaves, using the sibling pointers, until a key that is out of the requested range is reached.

6.1.3 Concurrent RQs in B+-trees

While RQs in a serial version of a B+-tree are simple, in a concurrent setup the correct implementation of an RQ is challenging. Concurrent updaters may modify keys that are in the way of the horizontal scan of the RQ and this may lead to inconsistent execution. An example of an erroneous execution of two RQs, concurrently with two updates, is given in Figure 6.2. Threads T1 and T2 perform an RQ for the same range of keys, [32-54]. Threads T3 and T4 insert keys 42 and 53, respectively. T1 and T2 follow the same path of leaves, however, the order in which

[1]For some key types, e.g., strings, it is not possible to enumerate all the possible keys inside a given range. In these cases hash tables are incapable of supporting RQs

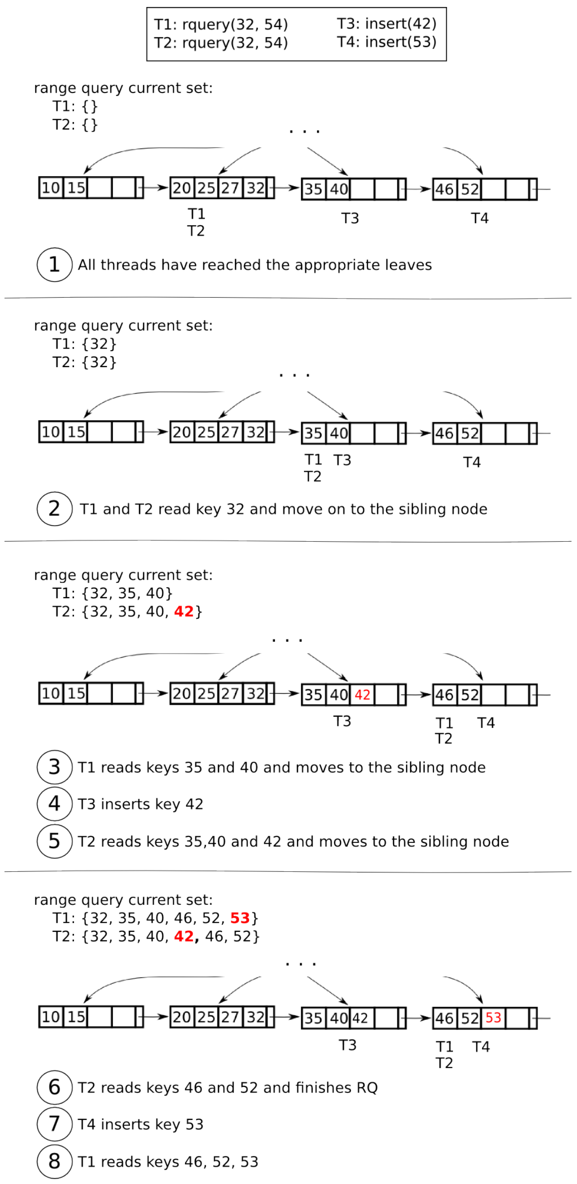

T1: rquery(32, 54)
T2: rquery(32, 54)
T3: insert(42)
T4: insert(53)
range query current set:
T1: {}
T2: {}
10 15
20 25 27 32
35 40
46 52
T1
T2
T3
T4
1 All threads have reached the appropriate leaves
range query current set:
T1: {32}
T2: {32}
T1 T3
T2
T4
2 T1 and T2 read key 32 and move on to the sibling node
range query current set:
T1: {32, 35, 40}
T2: {32, 35, 40, 42}
35 40 42
3 T1 reads keys 35 and 40 and moves to the sibling node
4 T3 inserts key 42
5 T2 reads keys 35,40 and 42 and moves to the sibling node
range query current set:
T1: {32, 35, 40, 46, 52, 53}
T2: {32, 35, 40, 42, 46, 52}
46 52 53
6 T2 reads keys 46 and 52 and finishes RQ
7 T4 inserts key 53
8 T1 reads keys 46, 52, 53

Figure 6.2: A non-linearizable execution of two RQs that run concurrently with two updates. The two RQ threads observe the two updates in different order. We only show the leaves that are involved in the four operations.

they read the sibling pointers of each leaf, causes them to observe a different ordering of the two inserts. RQs that use our proposed approach use an HTM transaction to get a consistent snapshot of this path of leaves. This way they avoid such inconsistent executions.

6.2 Previous Approaches

The importance of concurrent map data structures with RQ support resulted in a significant amount of research efforts towards this direction. The related work can be split in two categories, namely, hand-crafted data structures, that is data structures that were carefully designed and implemented to support RQs, and, general techniques that can be applied to several data structures and be extended with RQ support. Our approach to use *RCU-HTM* stands in the middle of the two categories; each and every *RCU-HTM* based data structure can be augmented with an RQ enclosed in an HTM transaction, as we explain in Section 6.3 for the case of B+-trees. However, we believe that in some of these data structures (e.g., binary search trees) the large memory footprint of the transactions will lead to low performance.

Hand-crafted data structures with RQ support. K-ary trees [BH11] are similar to B+-trees, in that multiple keys are stored in each node and the actual data is stored in the leaves. In contrast to B+-trees, the internal nodes are always full and the leaves can even be empty. Moreover, k-ary trees are unbalanced which makes them a good choice for applying lock-free synchronization. In [BA12] k-ary trees are augmented with RQ support.

Snaptree [BCCO10] is a partially external relaxed-balance AVL tree that uses fine-grained locks and supports snapshot operations, that is, getting a consistent snapshot of the whole tree, which can then be used to execute an RQ. As shown in [BA12], the snaptree adds high overhead to both updates and RQs since while taking the snapshot all concurrent updates are blocked. Another downside of snaptree is the necessity to take a snapshot of the whole data structure, even when the RQ concerns only a subset of the keys.

General techniques for supporting RQs. Transactional memory can be used in a straightforward way to implement a concurrent map with RQ support. All the operations, including the RQ, are executed inside a transaction and the TM system guarantees safe and correct concurrent execution. The problem with this approach is that software TMs (STMs) introduce very high overheads [ASS13] and hardware TMs (HTMs) have limitations that make transactions more likely to abort as they access more memory. Such coarse-grained TM concurrent maps are thus rarely satisfactorily efficient.

Read-Copy-Update (RCU) [MS98] is a technique that allows readers to traverse the data structure without using any synchronization. Updaters create copies of the parts of the data structure they need to modify and install their updated versions in a single atomic step. Readers can run concurrently with updaters, however, in the original RCU implementations updaters are serialized using a single lock. RQs can easily be executed if they also acquire this lock. However, this approach significantly decreases the concurrency of updaters and RQs.

Read-Log-Update (RLU) [MSFM15] combines locking, RCU and some techniques from STM and mitigates some of the limitations of plain RCU. With RLU, readers always see a snapshot of the data structure, so RQs can be easily implemented. However, as in RCU, updaters must block waiting for all concurrent operations to finish.

Contention-adaptive (CA) search trees [SW15] use a dynamically regulated number of locks to protect different parts of the data structure. The number of locks fluctuates depending on statistics about the contention which are collected at runtime. The data structure is split in dynamically sized sequential data structures and each lock protects one of these parts. When increased contention is observed in some individual part, this is split in two. Respectively, when decreased contention is observed, two parts are joined together. In order to keep track of the multiple sequential data structures, an additional tree structure is kept on top of them. CA trees were extended to support RQs [SW16], however they have two downsides. First, the additional tree structure increases the number of nodes that need to be accessed during a tree traversal. Second, an RQ may span several different sequential data structures, and locks need to be held for all of them. In [Win17] an optimization is proposed which uses immutable sequential data structures. This reduces the time during which an RQ must hold the appropriate locks.

The snap-collector [PT13] provides an object, which multiple threads can use to collaboratively build a snapshot of the data structure. A RQ can then use this snapshot to locate the keys that are inside the requested range. As was the case for snaptree, snap-collector is inefficient for small RQs since the snapshot includes the entire data structure. Moreover, it is not clear if snap-collector can be applied to more complex (and more efficient, at least regarding RQs) data structures than lists and skip-lists.

In [ARBM18] the authors exploit some characteristics of epoch-based reclamation techniques [] to implement an RQ provider which can be used by threads to execute consistent RQs. They provide three implementations of the RQ provider, a lock-based, a lockfree and one that uses HTM transactions. Their approach can be applied to a variety of data structures. The downside is the use of a global timestamp counter which is incremented by each RQ. As our evaluation shows, data structures that use this approach perform worse than our *RCU-HTM* based B+-tree.

6.3 RQs in an *RCU-HTM* B+-tree

6.3.1 Overview

We build on top of an *RCU-HTM* based B+-tree and extend it to support very simple, linearizable and efficient RQs. We exploit the fact that in an *RCU-HTM* B+-tree the keys of a leaf, and their associated values, never change[2]; when a key needs to be added/removed from a leaf, a copy of that node is created and replaces the old one. Based on that characteristic, an RQ can quickly take

[2]this is true for internal nodes as well, but this is irrelevant to our work

a snapshot of all the leaves involved in the RQ and then, without the need for synchronization, read all their key-values pairs.

Our RQs proceed in the following way. First, we traverse the tree until we reach the leaf node that includes the lowest key in the requested range. Then, we start an HTM transaction, which uses the leaves' sibling pointers to locate all the leaves that contain keys inside the range. During this transaction we only store pointers to these nodes and we do not have to copy them, as explained before. Inside the transaction we walk the list of sibling pointers and at each leaf we compare the highest key in the request range with the highest key of the leaf. By doing this, we avoid reading the whole array of keys which, for large node sizes, would result in adding multiple cache lines in the transactional read-set. By reading only the highest key, we add one cache line per leaf, thus we greatly decrease the memory footprint of the transaction.

6.3.2 Implementation

The C++ code for the RQ operation in our *RCU-HTM* B+-tree is given in Algorithm 2. The helper function *get_leaves()* starts from a leaf node and performs a horizontal scan of the leaves using the sibling pointers until a key larger than *key2* is encountered. To minimize the duration and the memory footprint of this scan, we do not scan all the keys on each node, we only compare *key2* with the highest key of the node. The helper function *get_keys()* scans all the leaves that contain keys inside the range and store the keys and their associated values in an array that can be returned to the caller of the range query operation.

The function *bptree_rcuhtm_rquery* first performs a traversal of the tree in line 24[3] to find the leaf that contains the first key in the requested range (or the leaf that would contain this key, if the key is not present in the map). If the reached leaf contains all the keys in the requested range, we can safely read and return the keys and their associated values without using transactions (lines 25–28). Otherwise, additional leaves need to be scanned, and to do so safely and guarantee linearizable execution this scan needs to be done atomically with respect to concurrent update operations. We achieve this atomicity either with HTM transactions (lines 29–33) or with a global lock that prevents the execution of concurrent updaters (lines 34–37). In lines 38–39, the array *rquery_leaves* stores pointers to all these leaves that contain keys in the requested key range. Since the keys (and the associated values) inside these leaves can never be modified, we can safely use *get_keys()* to scan these leaves without using any synchronization.

[3]*bptree_traverse()* performs a typical traversal of the B+-tree following child pointers until the appropriate leaf is reached.

ALGORITHM 2: Range Query operation in *RCU-HTM* B+-tree and helper functions.

```
// Per-thread heap-allocated data
__thread int *rquery_keys;
__thread void *rquery_values;
__thread bptree_node_t *rquery_leaves;

int get_keys (bptee_node_t *leaf, int key1, int key2)
    int i, j, nkeys = 0;
    for i = 0; i < nnodes; i++ do
        bptree_node_t *c = rquery_nodes[i];
        for j = 0; j < n->nkeys; j++ do
            if key1 < n->keys[j] && key2 >= n->keys[j] then
                rquery_keys[nkeys] = n->keys[j]; rquery_values[nkeys++] = n->values[j];
    return nkeys;

int get_leaves (bptree_node_t *leaf, int key2)
    bptree_node_t *c = leaf; // Currently examined leaf
    int nleaves = 0; // Number of leaves examined
    while c && c->keys[c->nkeys-1] < key2 do
        rquery_leaves[nleaves++] = c;
        c = c->sibling;
    if c then
        rquery_leaves[nleaves++] = c;
    return nleaves;

int bptree_rcuhtm_rquery (bptree *bpt, int key1, int key2)
    int nleaves, nkeys;
    int tx_retries = TX_MAX_RETRIES;
    bpt_node_t *leaf = bptree_traverse(bpt, key1);
    // If only one leaf is involved we can
    // avoid transactions
    if key2 < leaf->keys[leaf->nkeys-1] then
        rquery_leaves[0] = leaf;
        nkeys = get_keys_from_leaves(key1, key2, 1);
        return nkeys;
    // First try with HTM transactions
    while tx_retries-- > 0 do
        if TX_BEGIN() == TM_BEGIN_SUCCESS then
            nleaves = get_leaves(leaf, key2);
            TX_END();
            break;
    // If necessary, resort to the global lock
    if tx_retries <= 0 then
        lock_acquire(bptree->lock);
        nleaves = get_leaves(leaf, key2);
        lock_release(bptree->lock);
    // Now we can read the keys from the leaves.
    nkeys = get_keys_from_leaves(key1, key2, nleaves);
    return nkeys;
```

Label	Type	RQ synchronization
`btree-rcuhtm`	*RCU-HTM* B+-tree	HTM
`treap-ca`	Contention-adaptive treap [SW16]	Contention-adaptive locks [SW16]
`k-ary`	K-ary tree [BA12]	Non-blocking
`skiplist`	Fine-grained locking skip-list [HS08]	Lock-free RQ provider [ARBM18]
`citrus`	Citrus internal BST with fine-grained locking and RCU [AA14]	Lock-free RQ provider [ARBM18]
`lfbst`	Lock-free external BST [BER14]	Lock-free RQ provider [ARBM18]
`abtree`	Lock-free external (a,b)-tree	Lock-free RQ provider [ARBM18]

Table 6.1: Concurrent map implementations that were used in our evaluation.

6.4 Experimental Evaluation

We conduct our experiments on a dual socket Intel Broadwell-EP server with two Intel Xeon E5-2699 v4 processors each with 22 physical cores and 44 hardware threads, for a total of 44 and 88 physical cores and hardware threads respectively. We set the processors to run at a fixed frequency of 2.2GHz with TurboBoost mode disabled. Each core has private 32KB L1 and 256KB L2 caches, while a 56MB L3 cache is shared by all cores. The server has 256GB of RAM running at 2134MHz. The OS is Debian 8.3 with kernel version 4.7.0.

For our evaluation we used the C++ version of the benchmark code that was used in [ARBM18] which the authors have made publicly available [4]. Apart from the already provided concurrent maps, we implemented our *RCU-HTM* based B+-tree as well as the non-blocking k-ary tree [BA12] and the contention adaptive treap [SW16]. The complete list of concurrent map implementations that we used in our evaluation is presented in Table 6.1. All implementations were compiled using GCC 4.9.2 with -O3 optimizations enabled.

The benchmark methodology consists of the following:

- In the warmup phase a single thread inserts random keys into the data structure until it is filled with half of the keys of the key range.

- In the execution phase we spawn a number of worker threads, which repeatedly perform lookup, update (insert or delete) and RQ operations with randomly selected keys. The execution phase lasts for a predefined time duration, which we currently set to 5 seconds. We have validated that longer time durations produce similar results.

- We use different operation mixes and label each workload with L%-U%-R% where L, U and R are the proportions of lookups, updates and RQs, respectively. Updates are equally divided between inserts and deletes, thus the size of the data structure does not vary significantly throughout the execution. Our evaluation includes three different operation mixes; 0%-0%-100%, 20%-40%-2% and 0%-50%-50%.

[4]https://bitbucket.org/trbot86/implementations

- We use different key ranges which also determine the size of the data structure and, consequently, the level of contention. Our experiments include ranges with 20K, 2M and 20M keys.
- We used different RQ sizes, i.e., sizes of the requested key range. We execute experiments for 100, 1K, 10K and 100K RQ sizes.
- For the already provided implementations of skiplist, citrus, bst and abtree [ARBM18] we present the results of the lock-free RQ provider since this provided the best results.
- For the (a-b)-trees we set $a = 6$ and $b = 16$, as indicated by the authors in [Bro17] and in [ARBM18]. This means that a node may contain 6 to 16 keys. For the contention-adaptive treap we set the maximum number of keys a leaf can contain to 64 as indicated in [SW16]. For the k-ary tree we have set the k parameter to 32 as indicated in [BA12].
- We pin each worker thread on a single hardware thread. The first 22 threads occupy the 22 physical cores of a single socket, 44 threads span two sockets and 88 threads use hyper-threads.
- All reported results are the average of 10 independent executions with no significant variance.

6.4.1 Impact of B+-tree node size

In this set of experiments we aim to analyze what is the impact of the order of the B+-tree. The order of the B+-tree defines the minimum and maximum number of keys a node is allowed to have. More specifically, each node in the tree includes $order$ up to $2 * order$ keys. The size of the node impacts our implementation in the following way. An RQ uses a transaction to get a set of pointers to the appropriate leaves, the memory footprint of which is proportional to the number of leaves and not to the number of keys inside the range. This is true because as we traverse the list of sibling pointers, we only read the maximum key from each node and not the whole array of keys. For this reason, RQs in our B+-tree benefit from larger nodes. However, as the size of the node increases, the depth of the tree becomes more dense (i.e., uses less nodes to store the same set of keys), thus increasing contention. This is the tradeoff we seek to analyze and understand in this evaluation section.

Figure 6.3 presents the performance and the abort rate for *RCU-HTM* B+-trees with different node sizes and for varying RQ sizes. The key range is set to 2M and the operation mix to 50% updates and 50% RQs. For a small RQ size of 100 keys, the smaller the tree node the highest the performance. For 44 and 88 threads, the trees with order 64 and 128 have a high abort ratio, thus lacking in performance from the other trees with smaller nodes. This is attributed to the higher contention imposed in a tree with fewer leves. For 10K RQ size larger node sizes are better due to the smaller transaction sizes for the RQs. For 100K RQ size all our trees suffer from high number of aborts. In our future work we aim to explore ways to execute RQs with smaller fine-grained transactions, or ideally without executing any transaction, to overcome this limitation.

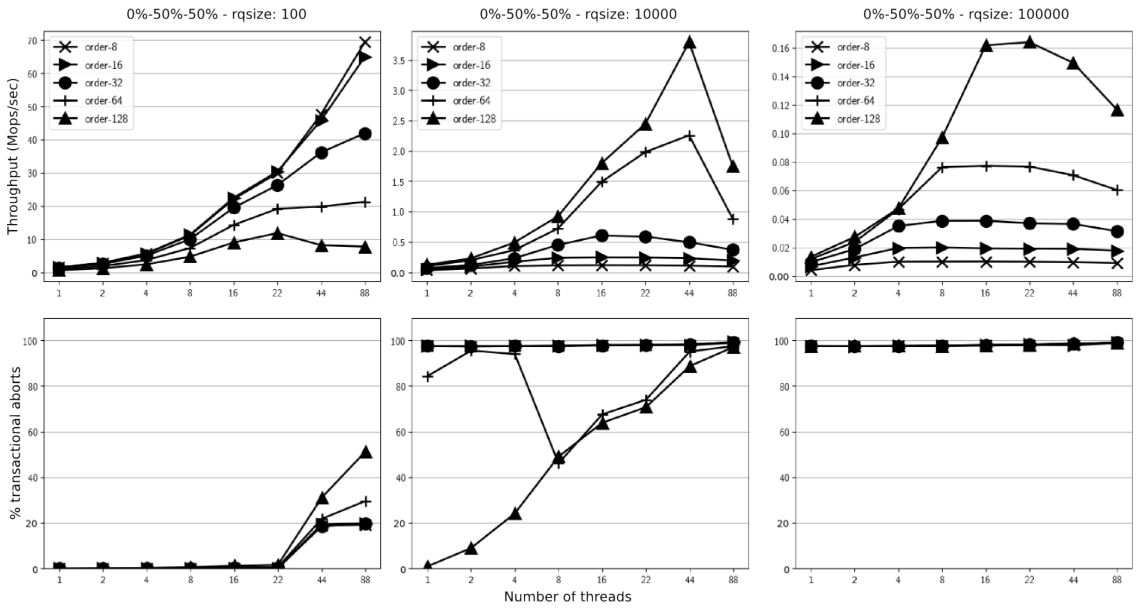


Figure 6.3: Performance of *RCU-HTM* B+-trees with varying B+-tree node sizes (i.e., order of the tree). The top row shows the throughput and the bottom row shows the associated percentage of aborted transactions.

6.4.2 Overall scalability

In this set of experiments we present the performance of our *RCU-HTM* B+-tree and compare it with the state-of-the-art concurrent maps with RQ support. Figures 6.4 and 6.5 present the throughput of all concurrent maps for RQ sizes of 100 and 100K keys. For the small RQ size, our *RCU-HTM* B+-tree outperforms its competitors on almost all cases. Its high performance is attributed to the low abort ratio for the HTM transactions that RQs execute. Only in the 20M key range with 100% RQs, the k-ary tree outperforms it. In this case the k-ary tree benefits from the absence of contention. Our implementation also suffers a performance drop for 44 and 88 threads on the 20K keys case with 40% and 50% updates. This is attributed NUMA effects due to *RCU-HTM*'s node allocating and copying mechanism which stresses the memory subsystem more than the other implementations.

For the large RQ size with 100K keys in the requested range our implementation is still the best for the 20K and 2M cases. In the 20K case, the RQ size is actually 10K since this is the size of the whole data structure. For this range size, our *RCU-HTM* B+-tree provides very high performance and scalability since it has very low transactional abort rates. On the larger trees, where the majority of RQ transactions abort, its performance drops. Although, in the 2M case, it still outperforms its competitors.

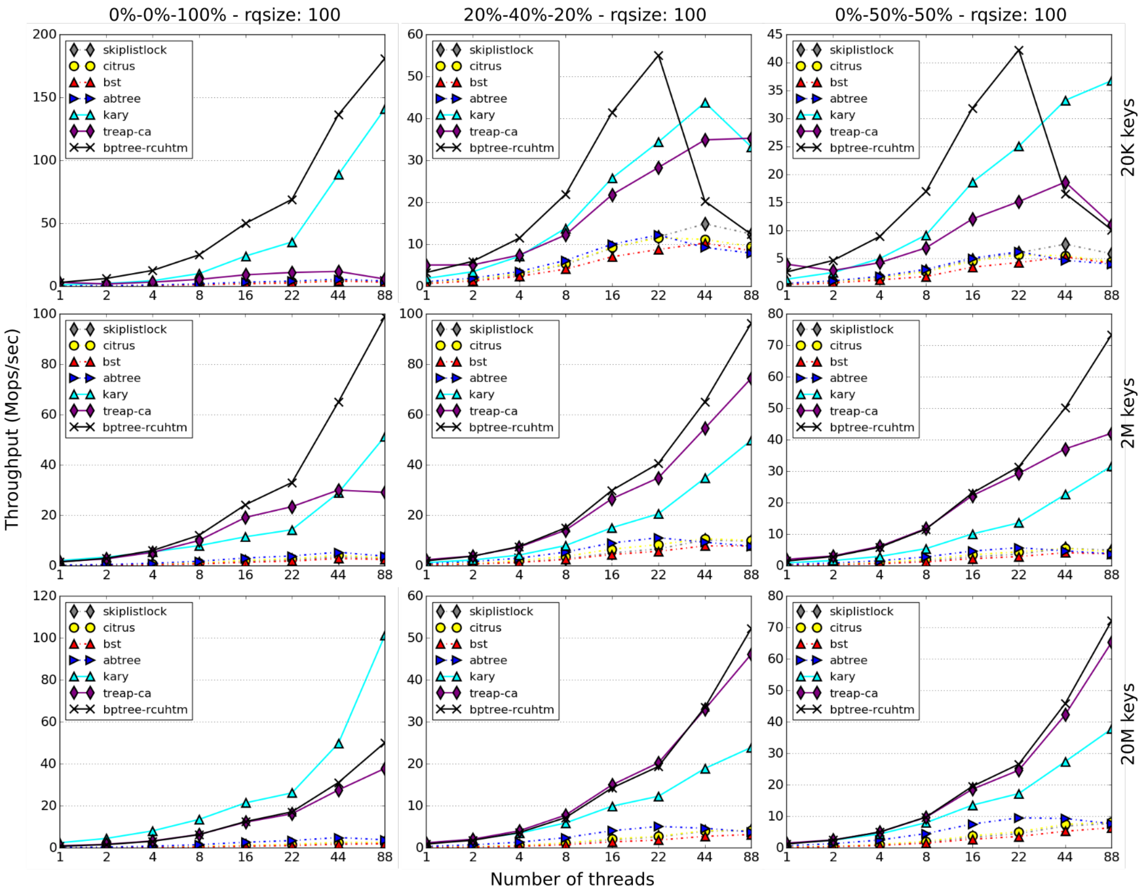


Figure 6.4: Performance of concurrent maps with RQ support for RQ size 100.

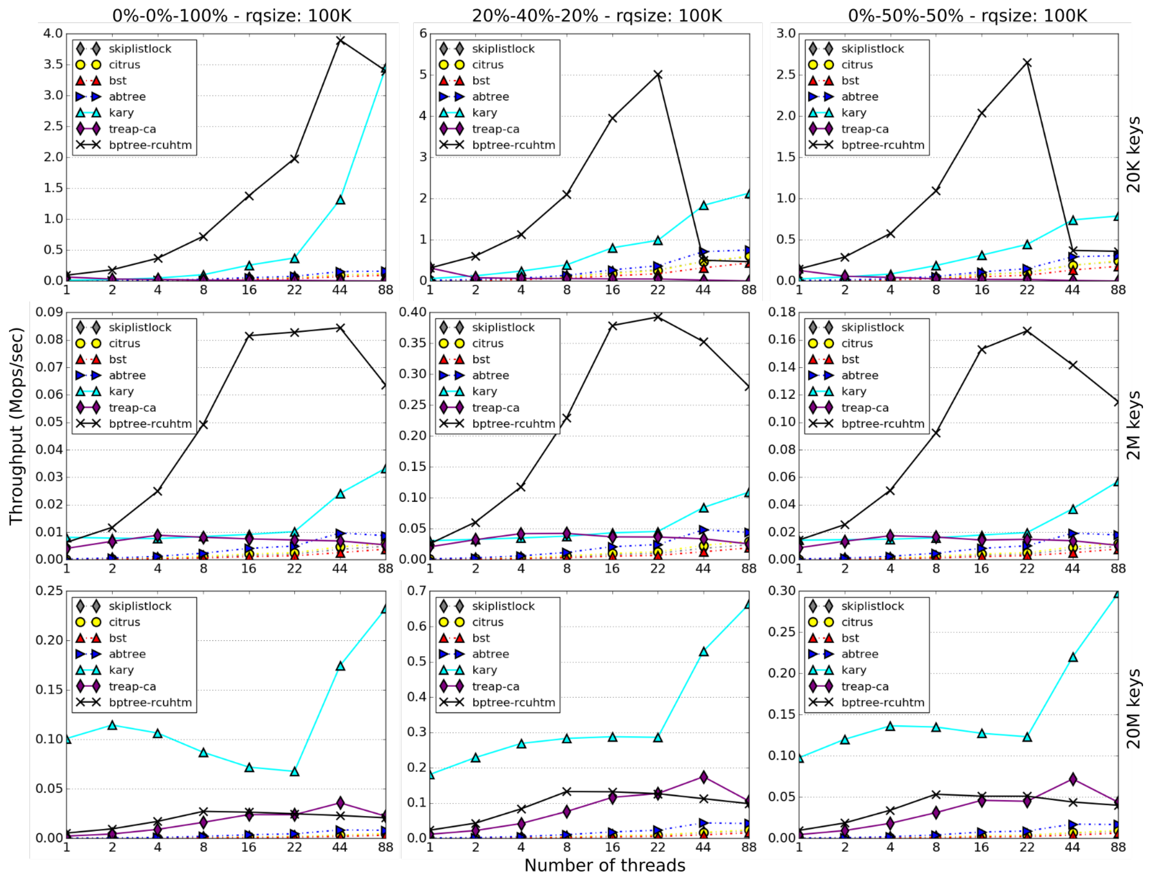


Figure 6.5: Performance of concurrent maps with RQ support for RQ size 100K.

CHAPTER 7

Conclusions & Future Work

In this work we proposed, implemented and evaluated *RCU-HTM*, a generic synchronization mechanism for implementing efficient concurrent search trees of any type (i.e., binary/B+-tree, internal/external, balanced/unbalanced). By leveraging two well known synchronization methods, RCU and HTM, *RCU-HTM* manages to combine their benefits and avoid their limitations. To the best of our knowledge, *RCU-HTM* is the first technique that is widely applicable to any search tree and provides high performance under any execution scenario. Further, we make it practical to use in large long-running applications by applying and evaluating a low-overhead epoch-based memory reclamation scheme. Finally, we added support for range query operations which are of critical importance for applications that use concurrent dictionaries, such as database management systems.

In our future work we plan to focus mainly on three directions. The first one is to automate the process of parallelizing a serial search tree using *RCU-HTM*. As shown during the presentation of *RCU-HTM* implementations, most parts of *RCU-HTM* are re-used. We believe therefore that it is possible to have a library and/or a compiler do all this repetitive work removing this burden from the programmer. Our second direction is to apply *RCU-HTM* on transactional data structures, that is data structures that can perform multiple operations in an atomic step. Finally, we are also interested in exploring whether *RCU-HTM* can also support range updates, that is operations that modify multiple key-value pairs in the data structure in an atomic way.

CHAPTER 8

Δημοσιεύσεις

Περιοδικά

- **D. Siakavaras**, K. Nikas, G. Goumas and N. Koziris: RCU-HTM: A Generic Synchronization Technique for Highly Efficient Concurrent Search Trees. In Journal of Concurrency and Computation: Practice and Experience (CCPE), Accepted 15 December 2020, Published 25 April 2021.

Συνέδρια

- **D. Siakavaras**, P. Billis, K. Nikas, G. Goumas and N. Koziris: Brief Announcement: Efficient Concurrent Range Queries in B+-trees using RCU-HTM. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, July 14-17, 2020, Philadelphia, PA, USA.

- F. Strati, C. Giannoula, **D. Siakavaras**, G. Goumas and N. Koziris: An Adaptive Concurrent Priority Queue for NUMA Architectures. In Proceedings of the ACM International Conference on Computing Frontiers (CF 2019), April 30 - May 2, 2019, Alghero, Sardinia, Italy.

- M. Kardaras, **D. Siakavaras**, K. Nikas, G. Goumas, N. Koziris: Fast Concurrent Skip Lists with HTM. In Proceedings of the International Symposia on High-Level Parallel Programming and Applications (HLPP 2018), July 12-13, 2018, Orléans, France.

- **D. Siakavaras**, K. Nikas, G. Goumas, N. Koziris: RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees. In Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT 2017), September 9-13, 2017, Portland, Oregon, USA.

- **D. Siakavaras**, K. Nikas, G. Goumas, N. Koziris: Combining HTM and RCU to Implement Highly Efficient Balanced Binary Search Trees. In Proceedings of the 12th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2017), February 5, 2017, Austin, Texas, USA.

- **D. Siakavaras**, K. Nikas, G. Goumas, N. Koziris: Massively Concurrent Red-Black Trees with Hardware Transactional Memory. In Proceedings of the The 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2016), Feb 17-19, 2016, Heraklion, Crete, Greece.

- **D. Siakavaras**, K. Nikas, G. Goumas, N. Koziris: Performance Analysis of Concurrent Red-Black Trees on HTM Platforms. In Proceedings of the 10th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2015), June 15-16, 2015, Portland, Oregon, USA.

Bibliography

[AA14] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[AAS11] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards consistency oblivious programming. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2011.

[AK14] Hillel Avni and Bradley C Kuszmaul. Improving HTM scaling with consistency-oblivious programming. *TRANSACT*, 2014.

[AM15] Maya Arbel and Adam Morrison. Predicate rcu: An rcu for scalable concurrent updates. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 21–30, New York, NY, USA, 2015. ACM.

[ARBM18] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, Boston, MA, 2018. USENIX Association.

[AS89] Cecilia Aragon and Raimund Seidel. Randomized search trees. pages 540–545, 11 1989.

[ASS13] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing tm-supported range queries. pages 299–308, 07 2013.

[BA12] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 31–45, 2012.

[BBB$^+$17] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *PPoPP '17*, 2017.

[BCCO10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[BER13] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, page 13–22, New York, NY, USA, 2013. Association for Computing Machinery.

[BER14] Trevor Brown, Faith Ellen, and Eric Ruppert. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM.

[BGMS98] Luc Bougé, Joaquim Gabarró, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries, 1998.

[BH11] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, page 207–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[BPA20] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 276–291, New York, NY, USA, 2020. Association for Computing Machinery.

[Bro15] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM.

[Bro17] Trevor Brown. A template for implementing fast lock-free trees using htm. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 293–302, New York, NY, USA, 2017. ACM.

[CBM$^+$08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, nov 2008.

[CDT14] Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas. Efficient lock-free binary search trees. *CoRR*, abs/1404.3272, 2014.

[CGR13] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag.

[Cha17] Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, New York, NY, USA, 2017. Association for Computing Machinery.

[DGT15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 631–644, New York, NY, USA, 2015. ACM.

[DVY14] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 343–356, New York, NY, USA, 2014. ACM.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[Fra04] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[FSBA11] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for gpu architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 296–307, 2011.

[HFP03] Timothy Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. volume 2508, 05 2003.

[HJ12] Shane V. Howley and Jeremy Jones. A Non-blocking Internal Binary Search Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, oct 2006.

[HLMM05] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[HW14] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 26(16):2684–2712, 2014.

[Mic02] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM.

[MS98] Paul E. Mckenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[MSFM15] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[PT13] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, page 224–238, Berlin, Heidelberg, 2013. Springer-Verlag.

[RM15] Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 37:1–37:10, 2015.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.

[ST97] Nir Shavit and Dan Touitou. *Software Transactional Memory.* 1997.

[SW15] K. Sagonas and K. Winblad. Contention adapting search trees. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 215–224, June 2015.

[SW16] Konstantinos Sagonas and Kjell Winblad. Efficient support for range queries and range updates using contention adapting search trees. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing*, pages 37–53, Cham, 2016. Springer International Publishing.

[TPK$^+$09] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 145–155, New York, NY, USA, 2009. Association for Computing Machinery.

[WIC$^+$18] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13, New York, NY, USA, 2018. ACM.

[Win17] Kjell Winblad. Faster concurrent range queries with contention adapting search trees using immutable data, 2017.

[WPL$^+$18] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 473–488, New York, NY, USA, 2018. Association for Computing Machinery.

[YBP$^+$14] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.