Αξιολόγηση και Βελτίωση Απόδοσης Συστήματος Εκτέλεσης Χρηματοοικονομικών Συναλλαγών

Κασσελούρης Απόστολος

Α.Μ: 2994

Διπλωματική Εργασία

Επιβλέπων: Αναστασιάδης Στέργιος

Ιωάννινα, Φεβρουάριος, 2022

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Ευχαριστίες

Πρωτίστως, θα ήθελα να ευχαριστήσω τον καθηγητή μου κύριο Στέργιο Αναστασιάδη για την καθοδήγηση που μου προσέφερε καθ' όλη την διάρκεια της διπλωματικής μου εργασίας, όσο και για το πολύ ενδιαφέρον θέμα που διάλεξε για να δουλέψουμε. Θα ήθελα επίσης να ευχαριστήσω και να εκφράσω την ευγνωμοσύνη μου στους φίλους και την οικογένεια μου για την πολύτιμη βοήθεια και αγάπη που μου έδειξαν καθ' όλη την διάρκεια συγγραφής της εργασίας μου.

Περίληψη

Τα χρηματιστήρια στις μέρες μας ανταγωνίζονται σθεναρά για το ποιο θα καταφέρει να έχει τη ταχύτερη μηχανή ταιριάσματος χρηματιστηριακών εντολών. Σε αυτή την διπλωματική εργασία θα μελετήσουμε το CppTrader, μια συλλογή από ενότητες κώδικα ανοιχτού λογισμικού πολύ υψηλών επιδόσεων για την κατασκευή πλατφόρμας χρηματιστηριακών συναλλαγών, εστιάζοντας στη ταχύτατη μηχανή ταιριάσματος εντολών που διαθέτει. Ο βασικός μας στόχος είναι μέσω διάφορων τεχνικών αξιολόγησης της απόδοσης να εντοπίσουμε πιθανές στενωπούς απόδοσης στις δομές δεδομένων και τους αλγορίθμους που χρησιμοποιούνται από το CppTrader και να τις βελτιώσουμε. Επίσης, για την βελτίωση του συστήματος, βασιζόμαστε σε ένα φορτίο εργασίας, το οποίο περιέχει αληθινά ιστορικά δεδομένα εντολών από το χρηματιστήριο του Nasdaq. Στο τέλος αυτής της διπλωματικής εργασίας, παρουσιάζουμε τα ευρήματα μας και προτείνουμε συγκεκριμένες μελλοντικές βελτιώσεις.

Λέξεις Κλειδιά: μηχανή ταιριάσματος χρηματιστηριακών εντολών, συστήματα υψηλής απόδοσης, αξιολόγηση απόδοσης συστήματος, δομές δεδομένων, αλγόριθμοι, βελτιστοποίηση στενωπού απόδοσης, φορτίο εργασίας Nasdaq

Abstract

Exchanges nowadays aggressively compete on having the fastest order matching engine. In this thesis we will study CppTrader, an open source set of high-performance components for building a trading platform, with focus on its ultra-fast matching engine. Our main goal is to identify potential bottlenecks in the data structures and algorithms of CppTrader, through various performance evaluation techniques, and improve them. Also, to improve the system we use a real workload that consists of historic trading data from Nasdaq stock exchange. At the end of this thesis, we present the findings of our work and propose specific future improvements.

Keywords: order matching engine, trading engine, high performance systems, performance evaluation, data structures, algorithms, bottleneck optimization, Nasdaq workload

Πίνακας Περιεχομένων

Κεφάλαιο 1.

Εισαγωγή

1.1 Κίνητρο

Οι τεχνολογικές καινοτομίες των τελευταίων δεκαετιών έχουν οδηγήσει τα χρηματιστήρια να ανταγωνίζονται για την ταχύτητα με την οποία επεξεργάζονται τις διάφορες εντολές αγοράς ή πώλησης χρηματιστηριακών προϊόντων, με ταχύτητες που φτάνουν ακόμα και τα μερικά δέκατα του μικρο-δευτερολέπτου. Οι λόγοι για τους οποίους συμβαίνει αυτό είναι κυρίως δύο [3, 4]. Πρώτον, τα χρηματιστήρια με τις ταχύτερες μηχανές ταιριάσματος εντολών τείνουν να χρεώνουν υψηλότερα τέλη (fees) για κάθε συναλλαγή (ταίριασμα), καθώς επίσης και να προσελκύουν μεγαλύτερη μερίδα χρηματιστών που τους ενδιαφέρει η ταχύτητα των συναλλαγών τους[1]. Ο δεύτερος λόγος είναι ένας νόμος που υφίσταται στις περισσότερες χώρες (Order Protection Rule στις ΗΠΑ [16]). Σύμφωνα με τον νόμο αυτό, εάν ένα χρηματιστηριακό προϊόν διατίθεται προς διαπραγμάτευση σε περισσότερα από ένα χρηματιστήρια, μια συναλλαγή (σε ένα χρηματιστήριο) δεν μπορεί να πραγματοποιηθεί εάν το ίδιο προϊόν διατίθεται σε καλύτερη τιμή σε ένα άλλο χρηματιστήριο. Έτσι, τα χρηματιστήρια αποκτούν κίνητρο για όλο και ταχύτερες μηχανές ταιριάσματος εντολών, καθώς ανταποκρίνονται γρηγορότερα στις μεταβολές της τιμής ενός προϊόντος, εξασφαλίζοντας την καλύτερη δυνατή τιμή.

Η όλο και ταχύτερη επεξεργασία των εντολών όμως δεν είναι εύκολη υπόθεση, καθώς τα μεγάλα χρηματιστήρια ανά τον κόσμο δέχονται εκατοντάδες εκατομμύρια εντολές την ημέρα. Περίπου οι μισές οφείλονται στα λεγόμενα συστήματα συναλλαγών υψηλής συχνότητας (HFT), τα οποία μπορούν, για ορισμένα χρονικά διαστήματα, να

[1] Κυρίως χρηματιστές συναλλαγών υψηλής συχνότητας (High Frequency Trading [17]).

παράγουν μέχρι και εκατοντάδες χιλιάδες εντολές το δευτερόλεπτο. Είναι υψίστης σημασίας, λοιπόν, τα χρηματιστήρια να έχουν μια μηχανή ταιριάσματος χρηματιστηριακών εντολών (trading ή order matching engine) που να μπορεί να διαχειριστεί όσο το δυνατόν ταχύτερα και αποδοτικότερα όλον αυτόν τον όγκο εντολών.

Σε αυτή την διπλωματική θα μελετήσουμε το CppTrader [1], μια συλλογή από ενότητες κώδικα ανοιχτού λογισμικού για την κατασκευή πλατφόρμας χρηματιστηριακών συναλλαγών, εστιάζοντας στην πολύ υψηλών επιδόσεων μηχανή ταιριάσματος χρηματιστηριακών εντολών που διαθέτει. Ο βασικός μας στόχος είναι να βρούμε πιθανές στενωπούς απόδοσης (bottlenecks) στο σύστημα σε επίπεδο αλγορίθμων και δομών δεδομένων και να τις βελτιώσουμε. Αυτό θα επιτευχθεί και εξακριβωθεί μέσω διαφόρων τεχνικών μέτρησης και αξιολόγησης της απόδοσης ενός συστήματος.

1.2 Απαιτήσεις Αξιολόγησης και Βελτίωσης της Απόδοσης

Η αξιολόγηση αλλά και η βελτίωση της απόδοσης ενός συστήματος είναι μια δύσκολη και απαιτητική διαδικασία [5, 6].

Αρχικά, μια βασική δυσκολία της αξιολόγησης της απόδοσης αποτελεί η **ερμηνεία των μετρήσεων** που θα πραγματοποιήσουμε σε ένα σύστημα για να το αξιολογήσουμε, η οποία πρέπει να βασίζεται σε **αντικειμενικά κριτήρια**. Ας σκεφτούμε το ακόλουθο παράδειγμα:

Η μέση ρυθμαπόδοση από εντολές που επεξεργάζεται ένα σύστημα είναι 1 εκατομμύριο εντολές/δευτερόλεπτο.

Η ερμηνεία αυτής της πληροφορίας από μόνη της δεν είναι ξεκάθαρη. Ενώ η ρυθμαπόδοση (throughput) είναι μια χρήσιμη μετρική, η ερμηνεία της βασίζεται σε υποκειμενικά κριτήρια. Η αντικειμενικότητα μιας μέτρησης μπορεί να επιτευχθεί όταν ορίζουμε ξεκάθαρους στόχους, όπως να έχουμε μια τιμή-στόχο για την μέση ρυθμαπόδοση, ή μέσω της σύγκρισης δύο ή περισσότερων υλοποιήσεων χρησιμοποιώντας την ίδια μετρική (στην περίπτωση μας την ρυθμαπόδοση) και παραμέτρους.

Δεύτερον, η βελτίωση της απόδοσης μπορεί να είναι απαιτητική λόγω της **πολυπλοκότητας** του συστήματος και της **έλλειψης ξεκάθαρου σημείου αφετηρίας**. Στο τελευταίο μπορούν να βοηθήσουν διάφορα εργαλεία ανάλυσης της

απόδοσης (profiling tools) ενός συστήματος (κεφάλαιο 3.1), τα οποία αποκαλύπτουν τις διάφορες στενωπούς της απόδοσής του. Δυστυχώς όμως, και οι στενωποί απόδοσης κρύβουν πολυπλοκότητα, μιας και η εξάλειψη τους δεν αποτελεί συνήθως μια εύκολη διαδικασία. Για παράδειγμα, σε έναν πίνακα που θέλουμε γρήγορη εισαγωγή και αναζήτηση, το να τον ταξινομήσουμε επιταχύνει την αναζήτηση αλλά επιβραδύνει την εισαγωγή και το αντίστροφο. Αυτό σημαίνει ότι χρειάζεται μια **ολιστική προσέγγιση και καλή γνώση ολόκληρου του συστήματος** που προσπαθούμε να βελτιώσουμε.

Τρίτον, μια ακόμα δυσκολία που κρύβει η βελτίωση της απόδοσης ενός συστήματος είναι η επιλογή των προβλημάτων απόδοσης στα οποία αξίζει να αφιερώσουμε χρόνο για να τα βελτιώσουμε. Αυτό είναι σημαντικό γιατί τα περισσότερα πολύπλοκα συστήματα κρύβουν πολλά προβλήματα απόδοσης και ο χρόνος ενός προγραμματιστή ή μιας ομάδας προγραμματιστών είναι πεπερασμένος. Αυτό σημαίνει ότι πρέπει να **ποσοτικοποιήσουμε** το μέγεθος των προβλημάτων, καθώς και των πιθανών βελτιώσεων που θα προκύψουν αν διορθωθούν. Το πρώτο είναι σχετικά απλό μιας και επιτυγχάνεται κυρίως μετρώντας το ποσοστό χρησιμοποίησης των πόρων του συστήματος. Αντίθετα η ποσοτικοποίηση της πιθανής βελτίωσης ενός προβλήματος, που είναι εξίσου σημαντική, επιτυγχάνεται αρκετά πιο δύσκολα αφού δεν υπάρχει συγκεκριμένη μεθοδολογία, και πολλές φορές επαφίεται στην εμπειρία του προγραμματιστή.

Τέταρτον, σημαντική έμφαση πρέπει πάντα να δίνεται στην ανάλυση των χαρακτηριστικών του **φορτίου εργασίας** που θα χρησιμοποιηθεί από ένα σύστημα. Μια εξαιρετικά επιτυχημένη βελτίωση χρησιμοποιώντας ένα συγκεκριμένο φορτίο, μπορεί να είναι καταστροφική σε θέμα απόδοσης για το ίδιο σύστημα με χρήση διαφορετικού φορτίου.

Τέλος, πολλές φορές για να βελτιώσουμε την απόδοση ενός συστήματος πρέπει να **θυσιάσουμε επιπλέον πόρους** (π.χ. μνήμη, πυρήνες CPU). Για παράδειγμα, σε αυτή τη διπλωματική εργασία ορισμένες βελτιώσεις που υλοποιήσαμε ήταν σε βάρος επιπλέον χρήσης μνήμης του συστήματος.

Αυτά και άλλα παρόμοια προβλήματα πιθανώς να αντιμετωπίσει ένας προγραμματιστής που καλείται να αξιολογήσει και/ή να βελτιώσει ένα σύστημα.

1.3 Σύνοψη Συνεισφορών

Αναφορικά με τις συνεισφορές μας, αυτές μπορούν να συνοψιστούν ως εξής:

- **Ανάλυση** της αρχιτεκτονικής, των λειτουργιών, των δομών δεδομένων και των αλγορίθμων που χρησιμοποιεί η μηχανή ταιριάσματος χρηματιστηριακών εντολών του CppTrader.
- **Αντικατάσταση** ορισμένων δομών δεδομένων, που αποτελούν στενωπούς της απόδοσης του συστήματος, με πιο αποδοτικές, λαμβάνοντας υπόψιν τις ιδιαίτερες απαιτήσεις και χαρακτηριστικά του φορτίου εργασίας.
- Ποσοτική **αξιολόγηση** της απόδοσης των νέων αυτών δομών δεδομένων και του CppTrader συνολικά.
- Συγκεκριμένες προτάσεις για πιθανές **μελλοντικές βελτιώσεις** του συστήματος.

1.4 Δομή Εργασίας

Η διπλωματική αυτή αποτελείται από 5 κεφάλαια στο σύνολο. Στο κεφάλαιο 2 παρουσιάζονται κάποιες προ-απαιτούμενες έννοιες για την κατανόηση αυτής της εργασίας, καθώς και οι βασικές λειτουργίες και δομές δεδομένων του CppTrader. Στο κεφάλαιο 3 γίνεται ανάλυση του φορτίου εργασίας και των χαρακτηριστικών του. Επίσης με βάση το profiling εντοπίζονται οι δομές δεδομένων που αποτελούν στενωπούς απόδοσης στο σύστημα και παρουσιάζεται η υλοποίηση των νέων δομών δεδομένων που τις αντικαθιστούν. Στο κεφάλαιο 4 εξετάζεται και αναλύεται πειραματικά η βελτίωση της απόδοσης που πετυχαίνουν οι νέες αυτές δομές δεδομένων. Τέλος, στο κεφάλαιο 5 αναλύονται και προτείνονται πιθανές μελλοντικές βελτιώσεις και παρουσιάζονται τα τελικά συμπεράσματα αυτής της εργασίας.

Κεφάλαιο 2.

Επισκόπηση του Συστήματος CppTrader

Σε αυτό το κεφάλαιο αρχικά θα παρουσιάσουμε ορισμένες βασικές έννοιες για την κατανόηση της λειτουργίας μιας μηχανής ταιριάσματος χρηματιστηριακών εντολών. Έπειτα θα δούμε τις βασικές ενότητες (modules) και δομές δεδομένων του CppTrader και την διασύνδεση μεταξύ τους. Ύστερα θα εμβαθύνουμε στην κάθε ενότητα και θα δείξουμε την λειτουργία της με χρήση ψευδοκώδικα. Τέλος, θα δούμε την τεχνική του memory pooling που χρησιμοποιεί το CppTrader για την διαχείριση αποθήκευσης στην μνήμη των αντικειμένων (objects) του.

2.1 Υπόβαθρο

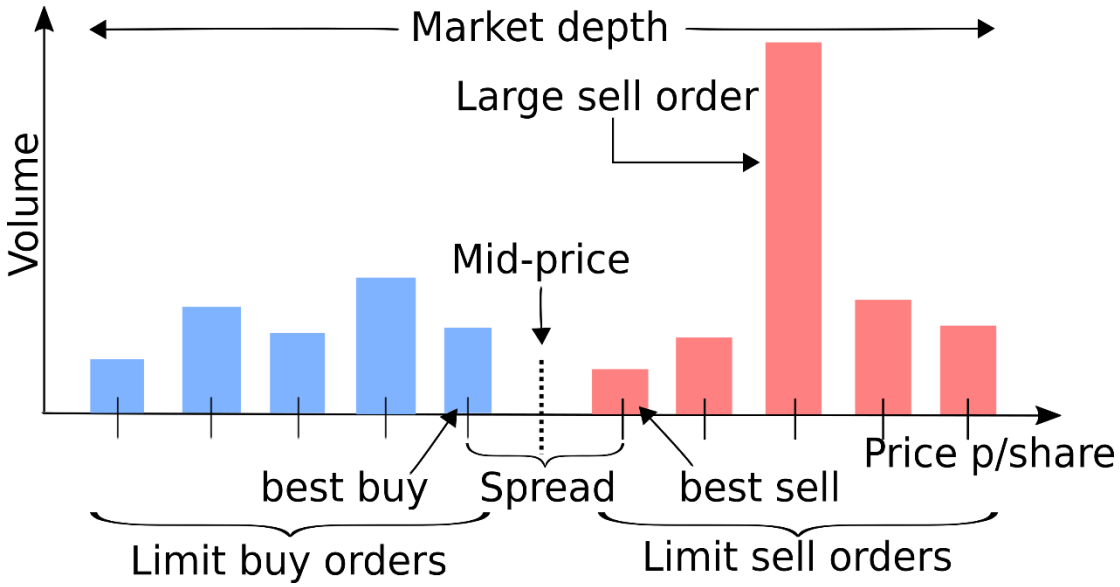
Ως **χρηματιστήριο (stock exchange)** εννοούμε την οργανωμένη αγορά, η οποία συνήθως είναι επίσημα αναγνωρισμένη από το κράτος, όπου συναντώνται οι ενδιαφερόμενοι για την διενέργεια αγοροπωλησιών **κινητών αξιών (securities)**[2]**.** Τις τελευταίες τρεις δεκαετίες σχεδόν όλα τα χρηματιστήρια πραγματοποιούν όλες αυτές οι αγοραπωλησίες ηλεκτρονικά. Ένα σύστημα ενός ηλεκτρονικού χρηματιστηρίου αποτελείται από πολλαπλά άλλα υποσυστήματα. Τα υποσυστήματα αυτά και οι λειτουργίες τους διαφέρουν σε ένα βαθμό από χρηματιστήριο σε χρηματιστήριο. Το μόνο υποσύστημα που είναι κοινό σε όλα τα χρηματιστήρια, είναι και το πιο βασικό, και ονομάζεται **μηχανή ταιριάσματος χρηματιστηριακών εντολών (matching ή trading engine)**. Η κύρια δουλειά ενός τέτοιου συστήματος είναι το ταίριασμα (matching) χρηματιστηριακών εντολών από τους χρήστες ενός χρηματιστηρίου. Με τον

[2] Οι κινητές αξίες μπορεί να είναι μετοχές, ομόλογα, παράγωγα και γενικά οποιαδήποτε αξία ή προϊόν μπορεί να είναι διαπραγματεύσιμο σε μια χρηματαγορά.

όρο χρηματιστηριακές εντολές, ή απλά **εντολές (orders)**, ονομάζουμε το σύνολο των οδηγιών για την αγορά ή πώληση μιας συγκεκριμένης ποσότητας μεριδίων μιας κινητής αξίας**.** Ιδιαίτερα, όταν μια κινητή αξία αναφέρεται στο κεφάλαιο προς διαπραγμάτευση μιας ανώνυμης εταιρείας, τα μερίδια αυτά ονομάζονται **μετοχές (stocks ή shares)**. Κινητή αξία ονομάζεται η αξία (ή προϊόν) που διατίθεται προς διαπραγμάτευση σε μια οργανωμένη χρηματαγορά, και μπορεί να αναφέρεται σε μετοχές, ομόλογα, παράγωγα, κλπ. Στην εργασία αυτή, κάθε αναφορά στον όρο της κινητής αξίας θα εννοείται η κινητή αξία μετοχών. Μετοχή ονομάζεται ένα από τα ίσα μερίδια, στα οποία διαιρείται μια κινητή αξία (μετοχών). Κάθε κινητή αξία που υπόκειται σε διαπραγμάτευση σε ένα χρηματιστήριο έχει μοναδική και ξεχωριστή δική της **συντομογραφία-σύμβολο (symbol ή ticker),** συνήθως τριών ή τεσσάρων γραμμάτων, που την ξεχωρίζει από οποιαδήποτε άλλη κινητή αξία σε ένα συγκεκριμένο χρηματιστήριο. Π.χ. η κινητή αξία της εταιρείας Apple Inc. στο χρηματιστήριο του Nasdaq έχει την συντομογραφία-σύμβολο AAPL

Ο πιο σημαντικός και συνήθης τύπος εντολής είναι η **οριακή εντολή (limit order).** Είναι μια εντολή για την αγορά ή πώληση μιας συγκεκριμένης ποσότητας μετοχών σε μια συγκεκριμένη (οριακή) τιμή ή σε καλύτερη από αυτή. Οριακές εντολές αγοράς μιας ποσότητας μετοχών μιας κινητής αξίας ταιριάζουν με οριακές εντολές πώλησης μετοχών της ίδιας κινητής αξίας και το αντίστροφο. Οι οριακές εντολές που λήφθηκαν από το σύστημα και δεν ταίριαξαν πλήρως ή δεν ταίριαξαν καθόλου αποθηκεύονται σε μια δομή που ονομάζεται βιβλίο εντολών για μελλοντική χρήση.

Το **βιβλίο εντολών (order book)** αποτελεί μια πολύ βασική έννοια. Σε υψηλό επίπεδο ένα βιβλίο εντολών είναι ένα είδος μητρώου, που χρησιμοποιούν τα χρηματιστήρια, αποτελούμενο από οριακές εντολές αγοράς και πώλησης μετοχών μιας συγκεκριμένης κινητής αξίας. Όλα τα κεντρικοποιημένα χρηματιστήρια χρησιμοποιούν ένα ξεχωριστό βιβλίο εντολών για κάθε κινητή αξία που φιλοξενούν. Ένα βιβλίο εντολών είναι δυναμικό, με την έννοια ότι συνεχώς ενημερώνεται σε πραγματικό χρόνο κατά την διάρκεια της ημέρας.



Σχήμα 2.1: Σχήμα από https://nms.kcl.ac.uk/rll/enrique-miranda/index.html. Στο σχήμα παρουσιάζεται η γραφική αναπαράσταση ενός βιβλίου εντολών (order book).

Κάθε ράβδος στο σχήμα 2.1 αναπαριστά ένα συγκεκριμένο **επίπεδο τιμής (price level),** το οποίο αποτελείται από μια ή περισσότερες εντολές. Οι εντολές προτεραιοποιούνται για ταίριασμα πρώτα με βάση την τιμή και έπειτα με βάση τον χρόνο που ελήφθησαν **(price/time priority).** Ο χρόνος αξιολογείται με την μέθοδο **FIFO (First In First Out),** δηλαδή, δεδομένου ότι έχουμε τις εντολές με την καλύτερη τιμή, από αυτές θα δρομολογηθεί πρώτη προς ταίριασμα αυτή που λήφθηκε και πρώτη. Οι εντολές **ταιριάζουν (matched)** όταν τα συμφέροντα των αγοραστών και των πωλητών ικανοποιούνται. Για παράδειγμα, όταν έρχεται μια εντολή αγοράς όπου η τιμή της είναι πιο υψηλή από τις εντολές που βρίσκονται στο χαμηλότερο επίπεδο τιμής πώλησης (best sell), τότε η εντολή αγοράς θα προσπαθήσει να ταιριάξει πλήρως με όσες περισσότερες εντολές από αυτό το επίπεδο πώλησης. Ισχύει, όμως, και το αντίστροφο, δηλαδή όταν έρχεται μια εντολή πώλησης όπου η τιμή της είναι πιο χαμηλή από τις εντολές που βρίσκονται στο υψηλότερο επίπεδο τιμής αγοράς (best buy), τότε η εντολή πώλησης θα προσπαθήσει να ταιριάξει πλήρως με όσες περισσότερες εντολές από αυτό το επίπεδο αγοράς. Το υψηλότερο επίπεδο τιμής αγοράς (best buy) και το χαμηλότερο πώλησης (best sell) αναφέρονται ως **η κορυφή του βιβλίου (top of the book)** και η διαφορά τους ονομάζεται **απόκλιση προσφοράς-ζήτησης (bid-ask spread).** Το **βάθος του βιβλίου (market ή book depth)** αναφέρεται στον αριθμό από διαφορετικά επίπεδα τιμής που υπάρχουν σε ένα συγκεκριμένο βιβλίο εντολών.

Επιπλέον των οριακών εντολών, η μηχανή ταιριάσματος χρηματιστηριακών εντολών του CppTrader υποστηρίζει εντολές διαγραφής, μείωσης και αντικατάστασης.

Οι **εντολές διαγραφής (delete orders)**, όπως υποδηλώνει και το όνομά τους, είναι υπεύθυνες για την διαγραφή μιας συγκεκριμένης οριακής εντολής από το βιβλίο εντολών που βρίσκεται. Οι **εντολές μείωσης (reduce orders)** είναι υπεύθυνες για την μείωση της ποσότητας μετοχών που μια οριακή εντολή θέτει προς αγορά ή πώληση. Τέλος, η λειτουργία των **εντολών αντικατάστασης (replace orders)** είναι η αντικατάσταση μιας οριακής εντολής σε ένα βιβλίο εντολών με μια άλλη στο ίδιο βιβλίο. Υπάρχει μια βασική διαφορά αυτών των εντολών με τις εντολές μείωσης. Η εντολή που επρόκειτο να αντικατασταθεί δεν ενημερώνεται, όπως στις εντολές μείωσης, αλλά ουσιαστικά διαγράφεται και επανεισάγεται στο βιβλίο εντολών, χάνοντας έτσι την χρονική της προτεραιότητα. Από αυτό το σημείο και μετά αν δεν αναφέρεται ξεκάθαρα ο τύπος της εντολής (π.χ. οριακή, διαγραφής, κλπ.) θα εννοείται ο τύπος των οριακών εντολών.

Το CppTrader υποστηρίζει επίσης τις **εντολές συμβόλου (symbol orders).** Αυτές είναι ειδικές εντολές υπεύθυνες για την δημιουργία βιβλίων εντολών για κάθε κινητή αξία με το συγκεκριμένο σύμβολο-συντομογραφία. Εμφανίζονται συνήθως στην αρχή της ημέρας, πριν ανοίξει το χρηματιστήριο, και αρχικοποιούν όλα τα βιβλία εντολών για όλες τις κινητές αξίες που φιλοξενεί το χρηματιστήριο. Δεν θα μας απασχολήσουν ιδιαίτερα σε αυτή την εργασία.

Υπάρχουν και άλλα είδη εντολών που μπορεί να υποστηρίξει το CppTrader, όπως **ελεύθερες εντολές (market orders)**, οι οποίες είναι εντολές για την αγορά ή πώληση μιας ποσότητας μετοχών ανεξαρτήτως τιμής. Δεν θα μελετηθούν βέβαια σε αυτή την εργασία καθώς το φορτίο εργασίας που θα χρησιμοποιήσουμε δεν τις περιλαμβάνει. Επίσης δε θα ληφθούν υπόψιν τα ιδιαίτερα χαρακτηριστικά αυτών των εντολών στις βελτιώσεις που θα πραγματοποιήσουμε.

Αναφορικά με την **έξοδο (output)** που παράγει το CppTrader αυτή μπορεί να είναι είτε διάφορα μηνύματα ενημερώσεων και/ή στατιστικές. Οι **ενημερώσεις (updates)** αναφέρονται σε μηνύματα όπως την προσθήκη ενός βιβλίου εντολών, την διαγραφή ενός επιπέδου τιμής, το ταίριασμα μιας εντολής, κλπ. Τα μηνύματα ενημερώσεων παράγονται σε πραγματικό χρόνο. Ένα παράδειγμα τέτοιου μηνύματος παρουσιάζεται στο σχήμα 2.2.

```
Add order: Order(Id=9538; SymbolId=65; Type=Limit; Side=BUY; Price=15;
    Quantity=30;)
```

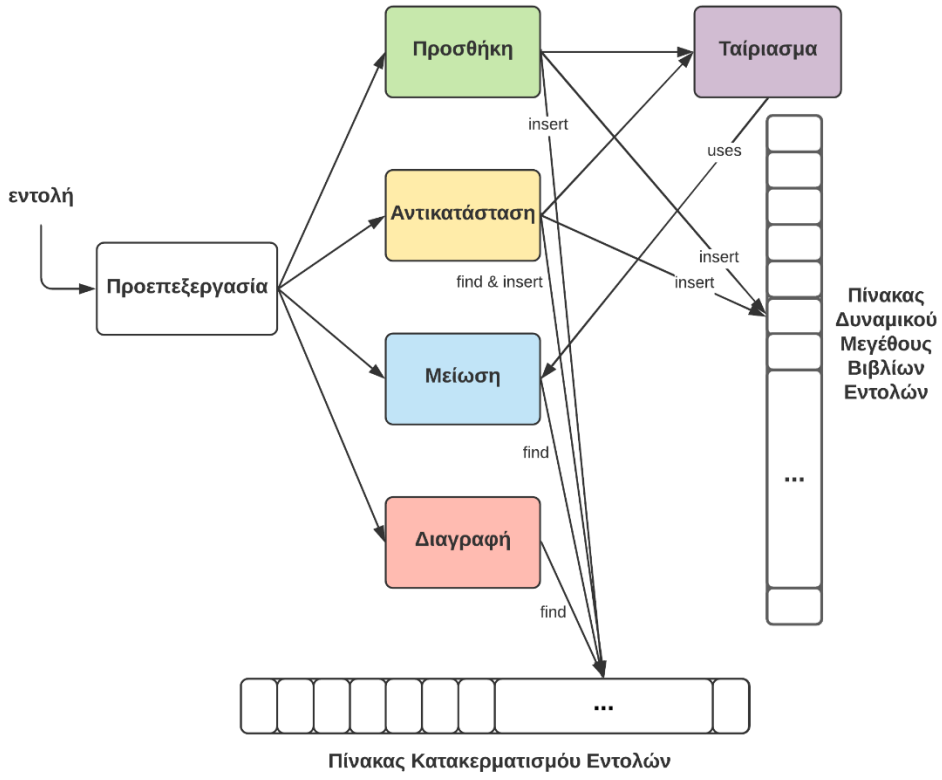
Σχήμα 2.2: Παράδειγμα μηνύματος ενημέρωσης, που παράγεται ως έξοδος από το CppTrader, και αναφέρεται στην προσθήκη μιας οριακής εντολής αγοράς με αναγνωριστικό 9538, στο βιβλίο εντολών για το αντίστοιχο σύμβολο με αναγνωριστικό 65, τιμή 15 και ποσότητα μετοχών 30.

Οι στατιστικές από την άλλη μεριά μπορεί να είναι ένας μετρητής για τον αριθμό των εντολών που βρίσκονται σε ένα βιβλίο εντολών, η μέση ρυθμαπόδοση (throughput) επεξεργασίας των εντολών από το σύστημα, κλπ. Οι στατιστικές μπορούν να εξαχθούν είτε κατά την διάρκεια είτε στο τέλος της εκτέλεσης του συστήματος.

2.2 Βασικές Ενότητες και Δομές Δεδομένων

Σε αυτήν την ενότητα θα παρουσιάσουμε τις βασικές ενότητες (modules) του CppTrader, πως αυτές αλληλοεπιδρούν μεταξύ τους αλλά και ποιες είναι οι λειτουργίες τους. Επίσης θα παραθέσουμε τις βασικές δομές δεδομένων (data structures), τον ρόλο τους, και από ποιες ενότητες χρησιμοποιούνται.

Στο σχήμα 2.3 παρουσιάζονται οι ενότητες του CppTrader. Αρχικά, όταν το σύστημα λαμβάνει μια εντολή, αυτή περνάει από την **ενότητα προ-επεξεργασίας** όπου μετατρέπεται σε κατανοητή μορφή από το σύστημα. Έπειτα, ανάλογα με το είδος της (π.χ. οριακή, διαγραφής, κλπ.) ανακατευθύνεται στην αντίστοιχη ενότητα.

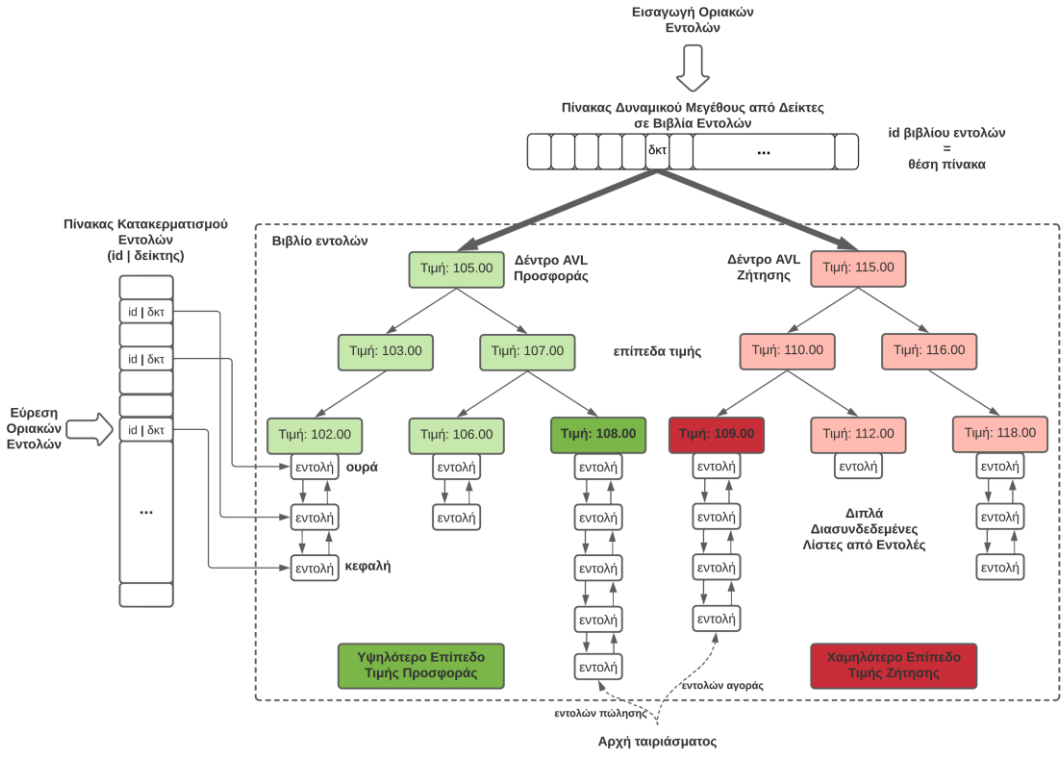


Σχήμα 2.3: Διάγραμμα τύπου μπλοκ που περιγράφει πως αλληλοεπιδρούν οι βασικές ενότητες του CppTrader μεταξύ τους αλλά και με δύο βασικές δομές δεδομένων του συστήματος.

Συνοπτικά, στην **ενότητα προσθήκης (add)** κατευθύνονται οι οριακές εντολές. Από εκεί πηγαίνουν στην **ενότητα ταιριάσματος (match)** και αν δεν ταιριάξουν

πλήρως επιστρέφουν στην ενότητα προσθήκης όπου εισάγονται στον **πίνακα κατακερματισμού εντολών (hashmap of orders)** και στην συνέχεια στο κατάλληλο βιβλίο εντολών που επιλέγεται από τον **πίνακα δυναμικού μεγέθους που περιέχει όλα τα βιβλία εντολών (dynamic array of order books).** Στην **ενότητα αντικατάστασης (replace)** μέσω του πίνακα κατακερματισμού εντοπίζεται η κατάλληλη εντολή και αφού εντοπιστεί, διαγράφεται από τον πίνακα κατακερματισμού και από το βιβλίο εντολών και εισάγεται η νέα εντολή ακολουθώντας τα ίδια βήματα με την ενότητα προσθήκης. Στην **ενότητα μείωσης (reduce)** μέσω του πίνακα κατακερματισμού εντοπίζεται η κατάλληλη εντολή και μειώνεται η ποσότητα μετοχών της καταλλήλως. Τέλος, στην **ενότητα διαγραφής (delete)** πάλι μέσω του πίνακα κατακερματισμού εντοπίζεται η κατάλληλη εντολή η οποία και διαγράφεται και από τον πίνακα κατακερματισμού και από το βιβλίο εντολών.

Για να γίνει πιο κατανοητή η λειτουργία των παραπάνω ενοτήτων πρέπει πρώτα να παρουσιάσουμε τις δομές δεδομένων του συστήματος, την λειτουργία τους, καθώς και την οργάνωση των εντολών μέσα σε αυτές (σχήμα 2.4).



Σχήμα 2.4: Οι βασικές δομές δεδομένων του συστήματος, η λειτουργία τους και η αλληλεπίδραση μεταξύ τους.

Αρχικά, ένας **δυναμικά μεταβαλλόμενου μεγέθους πίνακας από δείκτες (dynamic array of pointers)** χρησιμοποιείται για την αποθήκευση δεικτών σε αντικείμενα (objects) **βιβλίων εντολών**. Λόγω του τρόπου με τον οποίο δημιουργείται

ο πίνακας αυτός έχει επίσης την ιδιότητα κάθε θέση του (index) να έχει δείκτη σε βιβλίο εντολών με αντίστοιχο αναγνωριστικό (θέση πίνακα = id βιβλίου εντολών). Κάθε βιβλίο εντολών με την σειρά του περιέχει δύο **δέντρα AVL**, ένα που αντιπροσωπεύει την **προσφορά (bid)** μιας κινητής αξίας και ένα που αντιπροσωπεύει την **ζήτηση (ask)**. Τα δέντρα αυτά ανήκουν στην κατηγορία των ισορροπημένων δυαδικών δέντρων αναζήτησης (balanced binary trees) και οι κόμβοι τους αντιπροσωπεύουν τα **επίπεδα τιμής (price levels)** προσφοράς ή ζήτησης μιας κινητής αξίας. Τα επίπεδα τιμής είναι ταξινομημένα μέσα στο δέντρο με βάση την τιμή που αντιπροσωπεύουν. Τα πιο σημαντικά επίπεδα τιμής σε κάθε βιβλίο εντολών είναι:

- Από το δέντρο AVL της **προσφοράς,** το **υψηλότερο** επίπεδο τιμής που αποθηκεύεται στην μεταβλητή **best_bid** του βιβλίου εντολών.
- Από το δέντρο AVL της **ζήτησης,** το **χαμηλότερο** επίπεδο τιμής που αποθηκεύεται στην μεταβλητή **best_ask** του βιβλίου εντολών.

Τα επίπεδα αυτά είναι σημαντικά γιατί από το επίπεδο του best_bid ξεκινάει το ταίριασμα των εντολών πώλησης και το αντίθετο. Επιπλέον, τα επίπεδα τιμής με την σειρά τους αποτελούν και αυτά αντικείμενα (objects) όπου το καθένα περιέχει μια **διπλά διασυνδεδεμένη λίστα (doubly linked list)** από εντολές που βρίσκονται στο επίπεδο αυτό. Όλες οι εντολές εισέρχονται στην ουρά (tail) αυτής της λίστας και εξέρχονται για ταίριασμα από την κεφαλή (head). Οι λίστες αυτές δηλαδή λειτουργούν σαν δομές ουράς (queues). Τέλος, υπάρχει και ένας **πίνακας κατακερματισμού (hashmap)** για την άμεση πρόσβαση εντολών από τις ενότητες διαγραφής, μείωσης και αντικατάστασης. Τα στοιχεία σε αυτόν τον πίνακα είναι της μορφής ζεύγος κλειδιού-τιμής (key-value pair), όπου σαν κλειδί περιέχεται το αναγνωριστικό μιας εντολής και σαν τιμή ο δείκτης της εντολής αυτής στην μνήμη.

2.3 Λειτουργία Ενοτήτων

Στην ενότητα αυτή θα δούμε σε μεγαλύτερο βάθος την λειτουργία της κάθε ενότητας (module) του σχήματος 2.3 σε ψευδοκώδικα.

Αρχικά στον πίνακα 2.1 παρουσιάζονται κάποιες προαπαιτούμενες δομές και μεταβλητές για την κατανόηση των παρακάτω κομματιών ψευδοκώδικα.

Orderbooks	Πίνακας δυναμικού μεγέθους που περιέχει όλα τα βιβλία εντολών
OrderHashMap	Πίνακας κατακερματισμού εντολών
Bids	Δέντρο AVL επιπέδων τιμής προσφοράς ενός βιβλίου εντολών
Asks	Δέντρο AVL επιπέδων τιμής ζήτησης ενός βιβλίου εντολών
priceLevel	Επίπεδο τιμής ενός δέντρου AVL
OrderList	Διπλά διασυνδεδεμένη λίστα από εντολές ενός επιπέδου τιμής
best_bid	Μεταβλητή ενός βιβλίου εντολών που περιέχει το υψηλότερο επίπεδο τιμής ενός δέντρου προσφοράς
best_ask	Μεταβλητή ενός βιβλίου εντολών που περιέχει το χαμηλότερο επίπεδο τιμής ενός δέντρου ζήτησης

Πίνακας 2.1: Αριστερά η ονομασία στα κομμάτια ψευδοκώδικα και δεξιά ο ρόλος των δομών και των μεταβλητών του σχήματος 2.4.

Στο σχήμα 2.5 παρουσιάζεται ο ψευδοκώδικας της ενότητας προσθήκης. Ουσιαστικά προσπαθούμε πρώτα να ταιριάξουμε την οριακή εντολή που μόλις λάβαμε καλώντας την συνάρτηση MATCH της ενότητας ταιριάσματος. Αν δεν τα καταφέρουμε να την ταιριάξουμε πλήρως προσπαθούμε να την εισάγουμε στο τέλος της διπλά διασυνδεδεμένης λίστας του κατάλληλου επιπέδου τιμής, δέντρου AVL και βιβλίου εντολών. Αν δεν υπάρχει κατάλληλο επίπεδο τιμής, το δημιουργούμε εμείς και το εισάγουμε στο δέντρο.

```
function ADD(id, orderbook_id, price, quantity, side)
    order = new Order(id, orderbook_id, price, quantity, side)
    orderbook = Orderbooks[orderbook_id]
    MATCH(order, orderbook)
    if order.quantity > 0 then  //order NOT fully matched
        OrderHashMap.insert(id, order)
        if is_bid(side) then
            //if priceLevel not found, we create it
            orderbook.Bids.priceLevel.OrderList.push_back(order)
        else
            //if priceLevel not found, we create it
            orderbook.Asks.priceLevel.OrderList.push_back(order)
```

Σχήμα 2.5: Η λειτουργία της ενότητας προσθήκης (add module) σε ψευδοκώδικα. Η μεταβλητή quantity αναφέρετε στο αριθμό (ποσότητα) των μετοχών προς αγορά ή πώληση μιας οριακής εντολής,

Στο σχήμα 2.6 παρουσιάζεται ο ψευδοκώδικας της ενότητας μείωσης, όπου και προσπαθούμε να μειώσουμε την ποσότητα μετοχών μιας οριακής εντολής.

```
function REDUCE(id, reduce_quantity)
    order = OrderHashMap.find(id)
    order.quantity -= reduce_quantity
    if order.quantity <= 0 then
        order.priceLevel.OrderList.remove(order) //orders cache in a variable their
                                                 //price level for fast access
        OrderHashMap.erase(id)
        delete order
```

Σχήμα 2.6: Η λειτουργία της ενότητας μείωσης (reduce module) σε ψευδοκώδικα. Κάθε οριακή εντολή κρατάει σε μια μεταβλητή της το επίπεδο τιμής που βρίσκεται για γρήγορη πρόσβαση σε αυτό.

Στο σχήμα 2.7 παρουσιάζεται ο ψευδοκώδικας της ενότητας αντικατάστασης. Αρχικά αφαιρούμε μια οριακή εντολή από το βιβλίο εντολών της και τον πίνακα κατακερματισμού. Έπειτα τροποποιούμε τα πεδία της εντολής με νέα. Τέλος, για την νέα πλέον εντολή ακολουθούμε την ίδια διαδικασία όπως στην ενότητα προσθήκης.

```
function REPLACE(id, orderbook_id, new_id, new_price, new_quantity)
    //find and delete order from orderbook and OrderHashMap
    order = OrderHashMap.find(id)
    orderbook = Orderbooks[orderbook_id]
    order.priceLevel.OrderList.remove(order)
    OrderHashMap.erase(id)

    //replace order fields
    order.id = new_id
    order.price = new_price
    order.quantity = new_quantity

    //from here on same process as ADD function
    MATCH(order, orderbook)
    if order.quantity > 0 then  //order NOT fully matched
        OrderHashMap.insert(id, order)
        if is_bid(side) then
            //if priceLevel not found, we create it
            orderbook.Bids.priceLevel.OrderList.push_back(order)
        else
            //if priceLevel not found, we create it
            orderbook.Asks.priceLevel.OrderList.push_back(order)
```

Σχήμα 2.7: Η λειτουργία της ενότητας αντικατάστασης (replace module) σε ψευδοκώδικα.

Στο σχήμα 2.8 παρουσιάζεται ο ψευδοκώδικας της ενότητας διαγραφής, όπου ουσιαστικά διαγράφουμε μια οριακή εντολή, από όλες τις δομές που ανήκει και από την μνήμη.

```
function DELETE(id)
    order = OrderHashMap.find(id)
    order.priceLevel.OrderList.remove(order)
    OrderHashMap.erase(id)
    delete order
```

Σχήμα 2.8: Η λειτουργία της ενότητας διαγραφής (delete module) σε ψευδοκώδικα.

Τέλος, στο σχήμα 2.9 παρουσιάζεται ο ψευδοκώδικας της ενότητας ταιριάσματος. Ουσιαστικά είναι η υλοποίηση του αλγορίθμου ταιριάσματος προτεραιότητας τιμής/χρόνου (price/time priority) που περιγράφηκε στο κεφάλαιο 2.1.

```
function MATCH(order)
    //select best price level based on the order side (buy or sell)
    if is_bid(order.side) then
        price_level = orderbook.best_ask
    else
        price_level = orderbook.best_bid

    while contains_orders(price_level)  //while price_level is not empty
        if is_bid(order.side) then
            if order.price < price_level.price then
                return  //NO match possible
        else
            if order.price > price_level.price then
                return  //NO match possible

       exec_order = price_level.OrderList.front()  //get first order from best price level
       while exec_order.quantity > 0
            exec_quantity = min(order.quantity, exec_order.quantity)
            exec_price = exec_order.price
            REDUCE(exec_order.id, exec_quantity)
            order.quantity -= exec_quantity
            if order.quantity == 0 then
                return  //order FULLY matched
            exec_order = exec_order.next()  //get next order from price level

            //select the now new best price level
            if is_bid(order.side) then
                price_level = orderbook.best_ask
            else
                price_level = orderbook.best_bid
```

Σχήμα 2.9: Η λειτουργία της ενότητας ταιριάσματος (match module) σε ψευδοκώδικα. Ουσιαστικά είναι η υλοποίηση του αλγορίθμου ταιριάσματος προτεραιότητας τιμής/χρόνου.

2.4 Διαχείριση Μνήμης

Πριν αναφερθούμε στις τεχνικές διαχείρισης της μνήμης από το CppTrader πρέπει πρώτα να παρουσιάσουμε τον λόγο που οι τεχνικές αυτές είναι απαραίτητες σε ένα σύστημα υψηλών αποδόσεων όπως αυτό. Ο λόγος έγκειται στο φαινόμενο του **κατακερματισμού (fragmentation),** που πολλές φορές εμφανίζεται σε συστήματα με μη βέλτιστη διαχείριση της μνήμης, και οδηγεί σε μείωση της απόδοσης και της αποθηκευτικής ικανότητας ενός συστήματος. Αυτό, με απλά λόγια, συμβαίνει γιατί ο χώρος της μνήμης χρησιμοποιείται μη αποδοτικά, και ουσιαστικά σπαταλάται. Πιο συγκεκριμένα υπάρχουν δύο είδη κατακερματισμού, ο εσωτερικός και ο εξωτερικός. **Εσωτερικός** ονομάζεται ο κατακερματισμός όπου για ένα αντικείμενο (object) δεσμεύεται περισσότερος χώρος στην μνήμη από ότι χρειάζεται. **Εξωτερικός** κατακερματισμός ονομάζεται το φαινόμενο όπου ενώ έχουμε συνολικά στην μνήμη τον απαιτούμενο χώρο για την αποθήκευση ενός αντικειμένου, εντούτοις ο χώρος αυτός δεν βρίσκεται σε συνεχείς θέσεις μνήμης και δεν μπορούμε να βρούμε ένα αρκετά μεγάλο κομμάτι χώρου ώστε να δεσμεύσουμε για το αντικείμενο.

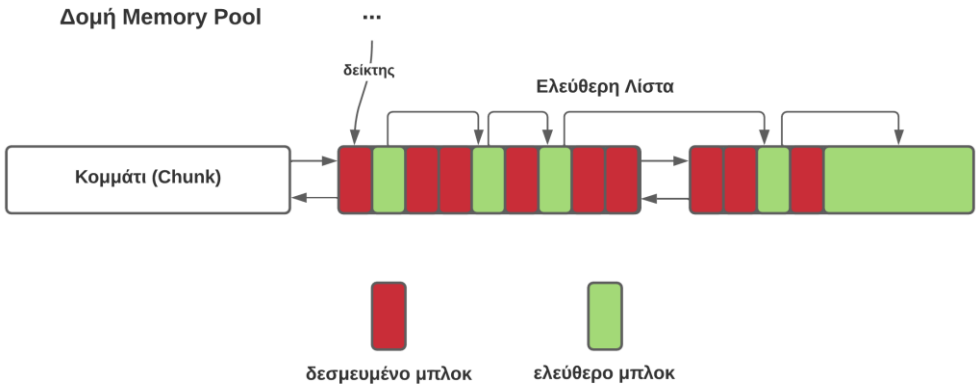
Για να αντιμετωπίσει αυτό το φαινόμενο, το CppTrader αποθηκεύει τα διάφορα αντικείμενά του (objects) χρησιμοποιώντας μια δομή που ονομάζεται **memory pool** [13]. Το CppTrader περιέχει τρεις τύπους αντικειμένων, και ειδικότερα ένα αντικείμενο βιβλίου εντολών, ένα εντολής, και ένα επιπέδου τιμής, το καθένα με το δικό του memory pool. Η λειτουργία αυτής της δομής είναι παρόμοια με την συνάρτηση malloc της γλώσσας C ή του operator new της γλώσσας C++. Η διαφορά έγκειται στη διαχείριση του σωρού (heap) της μνήμης ώστε να αποφεύγεται ο κατακερματισμός της και οι όποιες επιπτώσεις του σε θέματα απόδοσης και μνήμης. Υπάρχουν διάφορες τεχνικές για την υλοποίηση μιας δομής memory pool. Αυτή που χρησιμοποιεί το CppTrader είναι κάθε memory pool να ξεκινάει με ένα **κομμάτι (chunk)** ελεύθερης μνήμης 64KB το οποίο δεσμεύεται κανονικά με malloc. Μόλις γεμίσει ένα τέτοιο κομμάτι δημιουργούμε ένα καινούργιο με τον ίδιο τρόπο. Η διαδικασία αυτή συνεχίζεται για όσο χρειάζεται ή μέχρι να ξεμείνουμε από μνήμη. Κάθε κομμάτι έχει δύο πεδία δείκτη, ένα που δείχνει στο επόμενο και ένα στο προηγούμενο κομμάτι. Αποτελούν δηλαδή συνολικά μια διπλά διασυνδεδεμένη λίστα (double linked list) μεταξύ τους. Τέλος, αν ένα αντικείμενο που θέλουμε να αποθηκεύσουμε στην μνήμη είναι πολύ μεγάλο (μεγαλύτερο από το μέγεθος ενός κομματιού), δεσμεύουμε μνήμη με χρήση malloc.

Κάθε κομμάτι (chunk) αποτελείται από **μπλοκ (blocks)** ιδίου μεγέθους (υπάρχει δυνατότητα και διαφορετικού). Κάθε μπλοκ μπορεί να είναι είτε δεσμευμένο

(allocated) από κάποιο αντικείμενο, είτε ελεύθερο (free). Το κάθε ελεύθερο μπλοκ έχει ένα πεδίο δείκτη που δείχνει στο επόμενο, είναι δηλαδή συνολικά τα μπλοκ διασυνδεδεμένα ως λίστα (linked list). Η δομή αυτή της λίστας ονομάζεται και **ελεύθερη λίστα (free list)** [14] γιατί διασυνδέει μόνο ελεύθερα μπλοκ. Κάθε φορά που θέλουμε, λοιπόν, να αποθηκεύσουμε ένα αντικείμενο στην μνήμη εκτελούμε τα εξής βήματα:

1. Ανάλογα με τον τύπο του αντικειμένου επιλέγουμε το αντίστοιχο memory pool.
2. Ξεκινάμε να διασχίζουμε την ελεύθερη λίστα και στο πρώτο ελεύθερο μπλοκ που θα βρούμε και είναι αρκετά μεγάλο ώστε να χωρέσει το αντικείμενο μας το αποθηκεύουμε. Αυτή η μέθοδος ανάθεσης ονομάζεται **First Fit**.

Η δομή και η λειτουργία ενός memory pool στο CppTrader συνοψίζεται στο σχήμα 2.10.



Σχήμα 2.10: Η δομή memory pool του CppTrader είναι υπεύθυνη για την διαχείριση της μνήμης ενός τύπου αντικειμένων (objects). Κάθε κομμάτι (chunk) αρχικοποιείται με ένα ελεύθερο μπλοκ ιδίου μεγέθους, στο οποίο σταδιακά δεσμεύουμε μέρη του σειριακά ως μπλοκ. Κάθε δεσμευμένο μπλοκ (allocated block) είναι υπεύθυνο για την αποθήκευση ενός αντικειμένου. Τα ελεύθερα μπλοκ (free blocks) είναι διασυνδεδεμένα με τέτοιο τρόπο ώστε να δημιουργούν μια ελεύθερη λίστα (free list). Κάθε δεσμευμένο μπλοκ που αποδεσμεύεται από κάποιο αντικείμενο, εισάγεται ως ελεύθερο μπλοκ πλέον στην ελεύθερη λίστα.

Τέλος, η δομή memory pool, εκτός του περιορισμού του φαινομένου του κατακερματισμού, φαίνεται να πετυχαίνει και μια πιο ντετερμινιστική συμπεριφορά στην διαχείριση της μνήμης, αποφεύγοντας έτσι τα όποια σφάλματα όπως π.χ. την διαρροή μνήμης (memory leak).

Κεφάλαιο 3.

Βελτίωση Απόδοσης

Σε αυτό το κεφάλαιο αρχικά θα μελετήσουμε την έννοια του profiling, καθώς και το εργαλείο που θα χρησιμοποιήσουμε για να κάνουμε profiling. Επίσης θα αναλύσουμε ενδελεχώς το φορτίο εργασίας και τα ιδιαίτερα χαρακτηριστικά του. Τέλος με βάση τα χαρακτηριστικά του φορτίου και τα αποτελέσματα του profiling θα προτείνουμε βελτιώσεις σε διάφορες δομές του συστήματος που αποτελούν στενωπούς απόδοσης, αφού πρώτα τις αναλύσουμε.

3.1 Profiling

Profiling ονομάζουμε μια μορφή δυναμικής ανάλυσης του χώρου (μνήμη) και του χρόνου (χρήση CPU) ενός προγράμματος. Σε αυτή τη διπλωματική θα χρησιμοποιηθεί και θα μελετηθεί μόνο η χρονική ανάλυση. Αυτού του είδους η ανάλυση πραγματοποιείται από ειδικά προγράμματα που ονομάζονται profilers. Η βασική λειτουργία αυτών των προγραμμάτων δεν είναι η μέτρηση του χρόνου εκτέλεσης ενός προγράμματος, αλλά η **ποσοστιαία ανάλυση του χρόνου** των συναρτήσεων ενός προγράμματος και των συναρτήσεων που αυτές οι συναρτήσεις καλούν. Για παράδειγμα, μια συνάρτηση που τρέχει για 5 δευτερόλεπτα σε ένα πρόγραμμα που τρέχει για συνολικά 10 δευτερόλεπτα απασχολεί το 50% του χρόνου του. Αντίστοιχα, μια συνάρτηση που τρέχει για 1 λεπτό σε ένα πρόγραμμα που τρέχει για συνολικά 10 λεπτά απασχολεί το 10% του χρόνου του. Άρα με βάση το profiling και στα δύο αυτά προγράμματα, η πρώτη συνάρτηση επηρεάζει σε πολύ μεγαλύτερο βαθμό το πρόγραμμα που την καλεί από ότι η δεύτερη, παρόλο που η δεύτερη τρέχει για μεγαλύτερο χρονικό διάστημα.

Υπάρχουν δύο βασικές τεχνικές που οι profilers χρησιμοποιούν για την συλλογή δεδομένων τους. Η πρώτη ονομάζεται **ένθεση κώδικα (code injection).** Οι profilers που χρησιμοποιούν αυτή τη τεχνική ουσιαστικά προσθέτουν επιπλέον κώδικα σε ένα πρόγραμμα με σκοπό την συλλογή των απαιτούμενων δεδομένων. Το βασικό μειονέκτημα αυτών των profilers είναι ότι επιβαρύνουν σημαντικά το πρόγραμμα που αναλύουν, αυξάνοντας τον χρόνο εκτέλεσής του (μέχρι και 50 φορές παραπάνω), κάτι που τους καθιστά μη πρακτικούς σε μεγαλύτερα και πιο χρονοβόρα συστήματα όπως το CppTrader. Επίσης, ενώ θεωρητικά παράγουν πιο ακριβή αποτελέσματα σε σχέση με άλλου τύπου profilers, αυτή η σημαντική μείωση της ταχύτητας εκτέλεσης συχνά αποκρύπτει τις στενωπούς απόδοσης που ένα πρόγραμμα πιθανώς να έχει.

Η δεύτερη τεχνική profiling, και αυτή που θα χρησιμοποιήσουμε, ονομάζεται **δειγματοληψία (sampling).** Ένας profiler δειγματοληψίας ελέγχει και καταγράφει περιοδικά, μέσω διακοπών λειτουργικού συστήματος, ποια συνάρτηση/μέθοδος (ελέγχοντας την στοίβα μνήμης) τρέχει την δεδομένη χρονική στιγμή. Αυτού του είδους οι profilers είναι λιγότερο ακριβείς στις μετρήσεις τους, αλλά επιτρέπουν στο πρόγραμμα που αναλύουν να τρέχει σχεδόν χωρίς καμία επιβάρυνση. Επιπλέον, τα δεδομένα που παράγουν αυτοί οι profilers αποτελούν μια στατιστική προσέγγιση. Στην πράξη βέβαια, οι profilers δειγματοληψίας παρέχουν πιο ακριβή εικόνα για τους χρόνους εκτέλεσης ενός προγράμματος, μιας και είναι λιγότερο επεμβατικοί στο πρόγραμμα που αναλύουν.

Ένας πολύ γνωστός profiler δειγματοληψίας, και αυτός που θα χρησιμοποιήσουμε, είναι ο **gperftools** [7] (αρχικά Google Performance Tools). Μπορεί να κάνει profiling στη μνήμη, στον επεξεργαστή ή και στα δύο. Εμείς θα κάνουμε profiling μόνο στον επεξεργαστή. Η συχνότητα δειγματοληψίας που το έχουμε θέσει είναι η προκαθορισμένη στα **100 δείγματα το δευτερόλεπτο**. Ένα τυπικό αποτέλεσμα κειμένου του συγκεκριμένου profiler μπορούμε να δούμε στο σχήμα 3.1 και την ερμηνεία του στον πίνακα 3.1:

[1]
Total samples: 1362

[2]	**[3]**	[4]	[5]	[6]	**[7]**
14	**2.1%**	17.2%	58	8.7%	**std::_Rb_tree::find**

Σχήμα 3.1: Ένα παράδειγμα αποτελέσματος κειμένου του gperftools profiler. Τα έντονα γραμμένα στοιχεία είναι τα ποιο σημαντικά.

[1]	**Συνολικός αριθμός δειγμάτων**
[2]	Αριθμός από δείγματα σε αυτή την συνάρτηση
[3]	**Ποσοστό από τα συνολικά δείγματα σε αυτή τη συνάρτηση**
[4]	Ποσοστό από τα συνολικά δείγματα των συναρτήσεων που έχει εμφανιστεί έως τώρα
[5]	Αριθμός από δείγματα σε αυτή την συνάρτηση καθώς και στις συναρτήσεις που κάλεσε.
[6]	Ποσοστό από τα συνολικά δείγματα σε αυτή τη συνάρτηση καθώς και στις συναρτήσεις που κάλεσε
[7]	**Το όνομα της συνάρτησης**

Πίνακας 3.1: Η ερμηνεία του αποτελέσματος του σχήματος 3.1. Οι έντονα γραμμένες σειρές είναι οι ποιο σημαντικές.

3.2 Ανάλυση Φορτίου Εργασίας

Σε αυτή την ενότητα θα αναλύσουμε το φορτίο εργασίας, και τα ιδιαίτερα χαρακτηριστικά του, που θα χρησιμοποιήσουμε τόσο για τις μετρήσεις των πειραμάτων στο επόμενο κεφάλαιο όσο και για τις μετρήσεις profiling που θα κάνουμε σε αυτό το κεφάλαιο. Ο λόγος είναι ότι παίζει καθοριστικό ρόλο σε πολλές από τις αλλαγές σε θέματα δομών δεδομένων αυτού του κεφαλαίου.

Το είδος των φορτίων που θα χρησιμοποιήσουμε είναι αρχεία δεδομένων τύπου NASDAQ ITCH [8], τα οποία βρίσκονται σε δυαδική μορφή και περιέχουν αληθινά ιστορικά δεδομένα ημερών του χρηματιστήριου του Nasdaq [2]. ITCH ονομάζεται το πρωτοκόλλο που χρησιμοποιεί το χρηματιστήριο του Nasdaq για να περιγράψει την δομή και τα χαρακτηριστικά των χρηματιστηριακών δεδομένων που παράγει. Ένα παράδειγμα της μορφής που έχουν οι εντολές ενός τέτοιο φορτίου μετά την προ-επεξεργασία μπορούμε να δούμε στο σχήμα 3.2. Το πεδίο Type αναφέρεται στο είδος της εντολής (A για οριακή εντολή, X για εντολή μείωσης, D, για εντολή διαγραφής και U για εντολή αντικατάστασης). Το πεδίο StockLocate αναφέρεται στο βιβλίο εντολών για το οποίο προορίζεται αυτή η εντολή. Το πεδίο OrderReferenceNumber είναι το αναγνωριστικό (id) της κάθε εντολής. Το BuySellIndicator μας δείχνει αν μια οριακή εντολή προορίζεται για αγορά (B) ή πώληση (S) μετοχών. Το πεδίο Shares μας δείχνει την ποσότητα των μετοχών προς αγορά, πώληση ή μείωση, ενώ το πεδίο Stock περιέχει το σύμβολο-συντομογραφία της κινητής αξίας στην οποία αναφέρεται αυτή η εντολή. Το πεδίο Price περιέχει την τιμή μιας μετοχής πολλαπλασιασμένη x10.000. Τέλος,

υπάρχουν και άλλα πεδία, ανάλογα και το είδος της εντολής, αλλά δεν είναι απαραίτητα για την κατανόηση των εντολών του σχήματος 3.2.

```
AddOrderMessage(Type='A'; StockLocate=496; TrackingNumber=0;
    Timestamp=864714; OrderReferenceNumber=292701; BuySellIndicator='B';
    Shares=700; Stock="ARGX"; Price=1624000)

AddOrderMessage(Type='A'; StockLocate=8162; TrackingNumber=0;
    Timestamp=864714; OrderReferenceNumber=84412; BuySellIndicator='S';
    Shares=700; Stock="UN"; Price=579500)

OrderDeleteMessage(Type='D'; StockLocate=8162; TrackingNumber=0;
    Timestamp=864714; OrderReferenceNumber=84204)

OrderReplaceMessage(Type='U'; StockLocate=6643; TrackingNumber=0;
    Timestamp=864710; OriginalOrderReferenceNumber=122427;
    NewOrderReferenceNumber=122431; Shares=200; Price=591000)
```

Σχήμα 3.2: Παράδειγμα της μορφής που έχουν οι εντολές του φορτίου εργασίας αφού έχουν περάσει το στάδιο της προ-επεξεργασίας.

Το φορτίο που θα χρησιμοποιήσουμε για τις μετρήσεις μας περιλαμβάνει όλες τις εντολές που έλαβε το χρηματιστήριο του Nasdaq στις 30-12-2019. Περιέχει συνολικά **268,744,780** μηνύματα εντολών από τα οποία χρησιμοποιούμε τα **257.428.102**. Στον πίνακα 3.2 μπορούμε να δούμε τις εντολές από τις οποίες αποτελούνται οι χρήσιμες αυτές εντολές για εμάς.

Στατιστικά Εντολών Φορτίου Εργασίας

Είδος Εντολών	Πλήθος
Εντολές Συμβόλου	**8.906**
Οριακές Εντολές	**118.631.456**
Εντολές Διαγραφής	**114.360.997**
Εντολές Αντικατάστασης	**2.787.676**
Εντολές Μείωσης[3]	**21.639.067**

Πίνακας 3.2: Οι εντολές από τις οποίες αποτελούνται οι 257.428.102 χρήσιμες εντολές του φορτίου εργασίας. Υπενθύμιση ότι μια εντολή συμβόλου είναι υπεύθυνη για την δημιουργία μοναδικού βιβλίου εντολών για μια συγκεκριμένη κινητή αξία (μετοχών).

Στον πίνακα 3.3 μπορούμε να δούμε στατιστικά για το μέγιστο πλήθος στοιχείων που δέχονται οι δομές δεδομένων του συστήματος μια ορισμένη χρονική στιγμή. Ο πίνακας αυτός είναι ιδιαίτερα χρήσιμος γιατί μας δίνει μια ιδέα για την κλίμακα μεγέθους στοιχείων που οι δομές αυτές καλούνται να διαχειριστούν.

[3] Στο πρωτόκολλο ITCH του Nasdaq ονομάζονται εντολές ακύρωσης (cancel orders).

Στατιστικά Μέγιστου Πλήθους Στοιχείων Δομών Δεδομένων

Δομή δεδομένων (στοιχεία που αποτελείται)	Μέγιστο Πλήθος Στοιχείων
Πίνακας Δυναμικού Μεγέθους (βιβλία εντολών)	**8.906**
Πίνακας Κατακερματισμού (αποθηκευμένες εντολές σε όλα τα βιβλία εντολών)	**2.089. 674**
Δέντρο AVL (επίπεδα τιμής είτε προσφοράς είτε ζήτησης)	**4.272**
Διπλά Διασυνδεδεμένη Λίστα (εντολές σε ένα επίπεδο τιμής)	**3.415**

Πίνακας 3.3: Στατιστικά μέγιστου πλήθους στοιχείων που δέχονται σε μια δεδομένη χρονική στιγμή οι δομές δεδομένων του συστήματος (σχήμα 2.4). Ο μέγιστος αριθμός βιβλίων εντολών οφείλεται στις 8.906 εντολές συμβόλου του φορτίου (πίνακας 3.1).

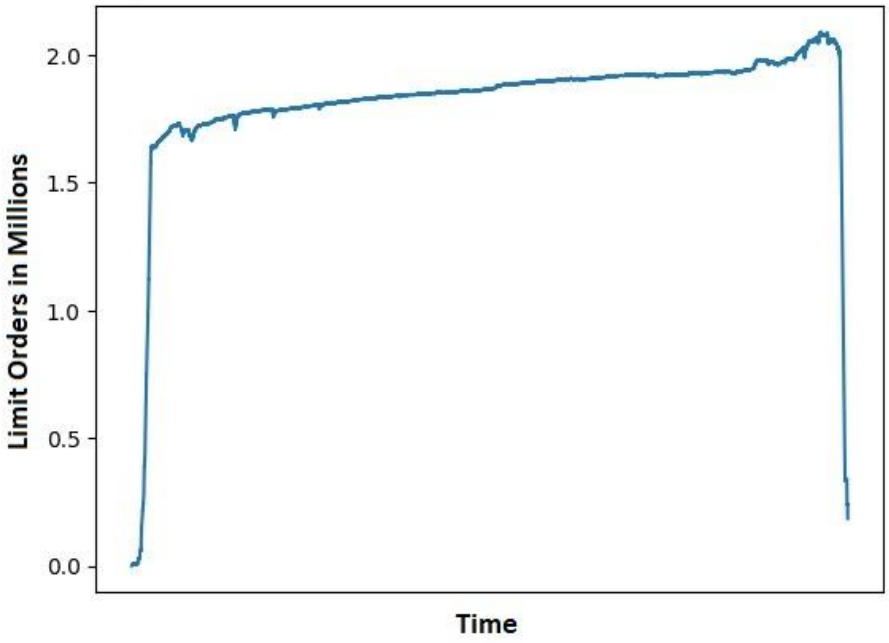
Τέλος στον πίνακα 3.4 παρουσιάζονται τα βασικότερα στατιστικά εξόδου (output) του CppTrader.

Στατιστικά Εξόδου

	Πλήθος
Μηνύματα ενημερώσεων που παράχθηκαν	**574.983.404**
Ταιριάσματα εντολών που πραγματοποιήθηκαν	**11.828.382**

Πίνακας 3.4: Δύο βασικά στατιστικά εξόδου που παράγονται από το CppTrader χρησιμοποιώντας το τρέχον φορτίο. Σημείωση ότι μια εντολή μπορεί να ταιριάξει περισσότερες από μια φορές.

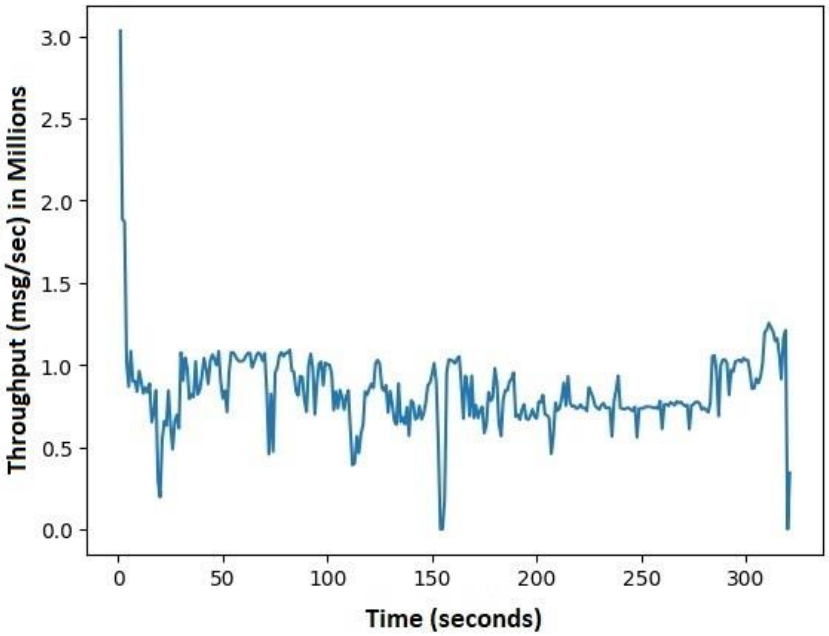
Ένα ενδιαφέρον χαρακτηριστικό αυτού του φορτίου που πρέπει να δώσουμε σημασία είναι το πως οι συνολικές οριακές εντολές που είναι αποθηκευμένες σε όλα τα βιβλία εντολών κυμαίνονται κατά την διάρκεια επεξεργασίας του φορτίου από το CppTrader (σχήμα 3.3). Στο σχήμα 3.3 παρατηρούμε ότι αρχικά οι εντολές που είναι αποθηκευμένες αυξάνονται απότομα γύρω στις 1.7 εκατομμύρια εντολές και από εκεί και έπειτα αυξάνονται πολύ πιο αργά αλλά σταθερά μέχρι περίπου τις 2.1 εκατομμύρια εντολές, για να διαγραφούν σχεδόν όλες στο τέλος της ημέρας. Σημαντικό να επισημάνουμε ότι αυτό το μοτίβο ακολουθούν όλα τα φορτία από το χρηματιστήριο του Nasdaq, με μικρές διακυμάνσεις.



Σχήμα 3.3: Στον κάθετο άξονα παρουσιάζονται οι συνολικές οριακές εντολές (limit orders) που είναι αποθηκευμένες σε όλα τα βιβλία εντολών. Στον οριζόντιο άξονα παρουσιάζεται κάθε χρονική στιγμή (time) επεξεργασίας του φορτίου από το CppTrader. Ουσιαστικά βλέπουμε την διακύμανση των συνολικών οριακών εντολών στα βιβλία εντολών καθ' όλη την διάρκεια εκτέλεσης του CppTrader.

Η απότομη αυτή αύξηση των αποθηκευμένων οριακών εντολών μειώνει απότομα και την ρυθμαπόδοση (throughput) των εντολών που μπορεί να επεξεργαστεί το CppTrader. Η ρυθμαπόδοση έπειτα παραμένει σε σχετικά σταθερά επίπεδα (περίπου 1 εκατομμύριο εντολές το δευτερόλεπτο) καθ' όλη την υπόλοιπη διάρκεια εκτέλεσης (σχήμα 3.4), παρά την συνεχή αύξηση των αποθηκευμένων εντολών. Επίσης το γεγονός ότι το ποσοστό χρησιμοποίησης (utilization) του επεξεργαστή φτάνει το 100% (γίνεται χρήση μόνο ενός πυρήνα) μετά την αποθήκευση περίπου 100.000 οριακών εντολών σημαίνει δύο πράγματα:

1. Ο πόρος στενωπός της απόδοσης είναι ο επεξεργαστής, καθώς η χρήση της μνήμης φτάνει το πολύ το 50% και ο δίσκος δεν χρησιμοποιείται καθόλου.
2. Το ποσοστό χρησιμοποίησης του επεξεργαστή στο 100% από τις 100.000 αποθηκευμένες εντολές και μετά (περίπου 1 δευτερόλεπτο από την εκκίνηση του συστήματος και μέχρι το τέλος) σε σχέση με το μέγεθος του φορτίου που χρησιμοποιούμε σημαίνει ότι και ένα φορτίο με το μισό ή το διπλάσιο μέγεθος να χρησιμοποιούσαμε δεν θα έκανε καμία διαφορά στην μέση ρυθμαπόδοση επεξεργασίας εντολών από το CppTrader. Αυτό δεν είναι απαραίτητα αρνητικό για εμάς καθώς μπορούμε να είμαστε βέβαιοι ότι **το φορτίο επιβαρύνει (stresses) σε μέγιστο βαθμό και καθ' όλη την διάρκεια (σταθερά) το σύστημα.**



Σχήμα 3.4: Η διακύμανση της ρυθμαπόδοσης (throughput) επεξεργασίας μηνυμάτων εντολών του CppTrader καθ' όλη τη διάρκεια εκτέλεσης του φορτίου.

Τέλος, είναι σημαντικό να αναφέρουμε ότι το να χρησιμοποιούσαμε διαφορετικά φορτία τέτοιου τύπου δεν θα παρήγαγε πολύ διαφορετικά αποτελέσματα, καθώς όλα αυτά τα φορτία (τύπου Nasdaq ITCH) έχουν παρόμοια δομή και κλίμακες (π.χ. ο μέγιστος αριθμός αποθηκευμένων οριακών εντολών σε όλα τα βιβλία εντολών κυμαίνεται στα 2+ εκατομμύρια σε όλα τα φορτία). Επομένως, μπορούμε να εξάγουμε το συμπέρασμα ότι το φορτίο που θα χρησιμοποιήσουμε για το profiling και τα πειράματά μας είναι αρκετά αντιπροσωπευτικό μιας ημέρας του χρηματιστήριου του Nasdaq.

3.3 Αρχικό Profiling

Αφού είδαμε τι είναι το profiling και πιο εργαλείο θα χρησιμοποιήσουμε μπορούμε να ξεκινήσουμε με το profiling του CppTrader. Αρχικές μετρήσεις profiling έδειξαν ότι η συνάρτηση __read() που χρησιμοποιείται για την ανάγνωση των εντολών από το αρχείο που περιέχει το φορτίο ήταν η κύρια στενωπός απόδοσης του συστήματος, καθώς χρησιμοποιούσε περισσότερο από το 25% του συνολικού χρόνου επεξεργασίας. Για να περιορίσουμε αυτήν την στενωπό απόδοσης πράξαμε τις ακόλουθες δυο ενέργειες:

1. Πρώτα συμπεριλάβαμε την εντολή "ios_base::sync_with_stdio(false);" στην αρχή του προγράμματός μας, απενεργοποιώντας έτσι τον συγχρονισμό μεταξύ των βασικών ρευμάτων εισόδου και εξόδου (I/O streams) της C και της C++

και επιτρέποντάς τους έτσι να έχουν τους δικούς τους ανεξάρτητους buffers [9]. Επομένως, ο buffer εισόδου του ρεύματος της C που μας ενδιαφέρει δεν επιβαρύνεται με τον συγχρονισμό εισόδου του ρεύματος της C++. Από την στιγμή, κιόλας, που χρησιμοποιήσαμε μόνο την συνάρτηση __read() της C ως είσοδο δεν αντιμετωπίσαμε κάποιο θέμα με την έλλειψη συγχρονισμού.

2. Επιπλέον, η συνάρτηση __read() παίρνει ως όρισμα το μέγεθος του buffer εισόδου της, δηλαδή τον αριθμό σε bytes που θα διαβάζει κάθε φορά. Αυξήσαμε, λοιπόν, τον αριθμό αυτό από τα 8192 bytes στα 32768 bytes (x4 αύξηση), ώστε να περιορίσουμε τις χρονοβόρες κλήσης στην συνάρτηση __read(), αυξάνοντας έτσι την συνολική ρυθμαπόδοση, χωρίς να αυξήσουμε σημαντικά τον χρόνο καθυστέρησης (latency) επεξεργασίας μιας εντολής.

Με αυτούς τους δύο τρόπους καταφέραμε να κάνουμε την συνάρτηση __read() να χρησιμοποιεί περίπου το 8% του συνολικού χρόνου του συστήματος. Περισσότερες λεπτομέρειες για την συγκεκριμένη βελτίωση θεωρούμε πως είναι εκτός του εύρους μελέτης αυτής της διπλωματικής.

```
Total: 19742 samples
    5000  25.3%  25.3%     5000  25.3% CppCommon::BinTreeAVL::InternalFind (inline)
    1692   8.6%  33.9%     1743   8.8% CppCommon::HashMap::find (inline)
    1688   8.6%  42.4%     1796   9.1% CppCommon::HashMap::emplace_internal (inline)
    1652   8.4%  50.8%     1652   8.4% __read
    1576   8.0%  58.8%     1576   8.0% std::vector::size (inline)
    1526   7.7%  66.5%     2920  14.8% CppTrader::Matching::OrderBook::DeleteOrder
    1363   6.9%  73.4%     1534   7.8% CppTrader::Matching::MarketManager::MatchOrder
     801   4.1%  77.5%      801   4.1% CppCommon::List::push_back (inline)
     575   2.9%  80.4%     1092   5.5% CppCommon::BinTreeAVL::erase
```

Σχήμα 3.5: Αρχικά αποτελέσματα profiling CppTrader. Στα αποτελέσματα παρουσιάζονται οι 9 πιο χρονοβόρες συναρτήσεις οι οποίες συνολικά ξοδεύουν περίπου το 80% του συνολικού χρόνου του συστήματος.

Αφού, λοιπόν, κάναμε αυτήν την πρώτη βελτίωση στο σύστημα, με ένα επόμενο profiling πήραμε τα αποτελέσματα που φαίνονται στο σχήμα 3.5. Το πρώτο πράγμα που παρατηρούμε σε αυτά τα αποτελέσματα είναι ότι η συνάρτηση InternalFind() του δέντρου AVL, που χρησιμοποιείται για την αναζήτηση ενός επιπέδου τιμής, αποτελεί τη μεγαλύτερη στενωπό της απόδοσης χρησιμοποιώντας το 25.3% του συνολικού χρόνου του συστήματος. Οι συναρτήσεις find() και emplace_internal() του πίνακα κατακερματισμού (hashmap) που χρησιμοποιούνται για την αναζήτηση και την εισαγωγή εντολών αντίστοιχα, χρησιμοποιούν και οι δυο το 8.6% του χρόνου του συστήματος ή συνολικά το 17.2% του χρόνου. Αυτό τις κατατάσσει μαζί ως τη δεύτερη μεγαλύτερη στενωπό απόδοσης του συστήματος. Μια παράξενη μέτρηση που παρατηρούμε είναι ότι η συνάρτηση vector::size() που είναι υπεύθυνη για την επιστροφή του μεγέθους ενός vector (δυναμικός πίνακας της C++) χρησιμοποιεί το 8% του χρόνου. Από διάφορα τεστ που πραγματοποιήσαμε παρατηρήσαμε ότι αυτή η

συνάρτηση στην πραγματικότητα χρησιμοποιεί πολύ λιγότερο συνολικό χρόνο (<1%). Καταλήξαμε έτσι στο συμπέρασμα ότι αυτή η μέτρηση πιθανώς οφείλεται σε βαριές βελτιστοποιήσεις τύπου -O3 του μεταγλωττιστή (compiler) που δεν είμαστε απόλυτα εξοικειωμένοι. Τέλος, παρατηρούμε ότι η συνάρτηση MatchOrder() που είναι υπεύθυνη για το ταίριασμα των εντολών ξοδεύει μόνο το 6.9% του συνολικού χρόνου.

Στις επόμενες ενότητες θα μελετήσουμε περισσότερο την δομή δέντρου AVL και του πίνακα κατακερματισμού (hashmap), την χρήση τους, ποια προβλήματα αναφορικά με την απόδοση αντιμετωπίζουν και θα προτείνουμε άλλες γρηγορότερες δομές για να τις αντικαταστήσουμε.

3.4 Ταξινομημένος Πίνακας Δυναμικού Μεγέθους από Δείκτες

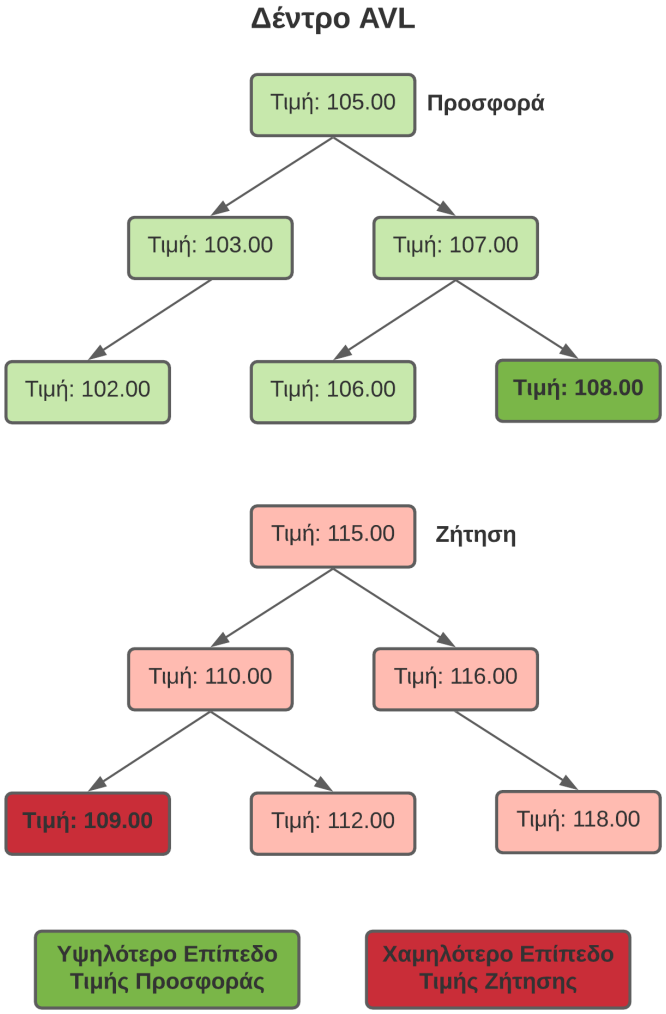
Αρχικά, μιας και η δομή του δέντρο AVL είναι η μεγαλύτερή μας στενωπός απόδοσης αυτή τη στιγμή, θα ήταν χρήσιμο να εξηγήσουμε την λειτουργία και τον ρόλο της στο σύστημα. Ένα δέντρο AVL είναι ένα ισορροπημένο δυαδικό δέντρο αναζήτησης (balanced binary search tree). Σε ένα δέντρο AVL τα ύψη των δύο θυγατρικών υποδέντρων οποιουδήποτε κόμβου διαφέρουν το πολύ κατά ένα, επομένως ονομάζεται και δέντρο ισορροπημένου ύψους (heigh-balanced tree). Η αναζήτηση, η εισαγωγή και η διαγραφή έχουν όλες O(log N) χρονική πολυπλοκότητα τόσο στην μέση όσο και στην χειρότερη περίπτωση. Οι εισαγωγές και οι διαγραφές κόμβων μπορεί να απαιτήσουν την εξισορρόπηση κατά μία ή περισσότερες περιστροφές του δέντρου. Ο παράγοντας ισορροπίας (balance factor) ενός κόμβου είναι το ύψος του δεξιού υποδέντρου του μείον το ύψος του αριστερού υποδέντρου του. Κάθε κόμβος με παράγοντα ισορροπίας 1, 0 ή -1 θεωρείται ότι είναι ισορροπημένος, ενώ με κάθε άλλο παράγοντα ισορροπίας θεωρείται μη ισορροπημένος και χρειάζεται εξισορρόπηση του δέντρου.

Στην περίπτωσή μας, όπως φαίνεται και στο σχήμα 2.4, ένα δέντρο AVL χρησιμοποιείται για την αποθήκευση των επιπέδων τιμής της προσφοράς και της ζήτησης ενός βιβλίου εντολών. Υπάρχουν 4 περιπτώσεις χρήσης ενός τέτοιου δέντρου στο σύστημα του CppTrader:

1. Μόλις λάβαμε μια οριακή εντολή και θέλουμε να **βρούμε (find)** σε πιο επίπεδο τιμής να την εισάγουμε.
2. Ψάξαμε για το κατάλληλο επίπεδο τιμής αλλά δεν το βρήκαμε. Έτσι πρέπει να το δημιουργήσουμε εμείς και να το **εισάγουμε (insert)** στο κατάλληλο δέντρο AVL. Και η εισαγωγή και η αναζήτηση έχουν O(log N) χρονική πολυπλοκότητα.

3. Θέλουμε να διαγράψουμε την τελευταία εντολή που έχει ένα επίπεδο τιμής, επομένως πρέπει να **διαγράψουμε (delete)** και το ίδιο το επίπεδο τιμής. Η χρονική πολυπλοκότητα αυτής της ενέργειας είναι κατά μέσο όρο O(1) μιας και κάθε εντολή αποθηκεύει σε μια μεταβλητή "Level" το επίπεδο τιμής στο οποίο ανήκει για άμεση πρόσβαση.
4. Τέλος, η συνάρτηση MatchOrder() που είναι υπεύθυνη για το ταίριασμα των εντολών χρειάζεται να έχει πάντα **πρόσβαση (access) στο υψηλότερο επίπεδο τιμής της προσφοράς και το χαμηλότερο επίπεδο τιμής της ζήτησης.** Η χρονική πολυπλοκότητα αυτής της ενέργειας είναι πάντα O(1) μιας και κάθε βιβλίο εντολών έχει δυο μεταβλητές "best_bid" και "best_ask" που κρατάνε τα αντίστοιχα επίπεδα τιμής. Μετά από κάθε εισαγωγή ή διαγραφή ενός επιπέδου τιμής οι μεταβλητές αυτές ενημερώνονται καταλλήλως, επίσης με χρονική πολυπλοκότητα O(1).

Αφού εξετάσαμε την λειτουργία του δέντρου AVL στο σύστημα, η αρχική μας σκέψη για την εξάλειψη της στενωπού απόδοσης που προκαλεί η συνάρτηση αναζήτησης (find) του δέντρου ήταν να ελέγξουμε τον αλγόριθμο με τον οποίο υλοποιείται. Ήταν όμως ένας απλός αλγόριθμος δυαδικής αναζήτησης, οπότε δεν μπορούσαν να γίνουν και πολλές αλλαγές. Η δεύτερη σκέψη μας ήταν να αλλάξουμε την δομή αυτή με μια παρόμοια (άλλου είδους δυαδικό δέντρο), όπως το κόκκινο-μαύρο δέντρο (red-black tree). Όμως και αυτή η δομή αντιμετώπιζε τα ίδια ακριβώς προβλήματα με το δέντρο AVL και μάλιστα απέδιδε ελαφρώς χειρότερα λόγω της μη αυστηρής ισορροπίας ύψους της.



Σχήμα 3.6: Η δομή δέντρου AVL που χρησιμοποιούνταν αρχικά από το CppTrader για την αποθήκευση των επιπέδων τιμής.

Η λύση στο παραπάνω πρόβλημα ήρθε από μια απάντηση [10] σε ένα φόρουμ με θέμα «Κατάλληλες δομές δεδομένων για την υλοποίηση ενός βιβλίου εντολών». Η απάντηση ουσιαστικά συνιστούσε την χρήση μιας δομής ταξινομημένου πίνακα δυναμικού μεγέθους αντί της χρήσης δυαδικών δέντρων, μιας και τα επιθυμητά επίπεδα τιμής συνήθως βρίσκονται σε πολύ μικρό βάθος από την κορυφή του βιβλίου εντολών τους (δείτε σχήμα 2.1 για υπενθύμιση των εννοιών «βάθος βιβλίου» και «κορυφή του βιβλίου»). Έτσι παρ' όλο που μια τέτοια δομή έχει γραμμική αναζήτηση O(N), εντούτοις παρουσιάζει τα εξής **πλεονεκτήματα:**

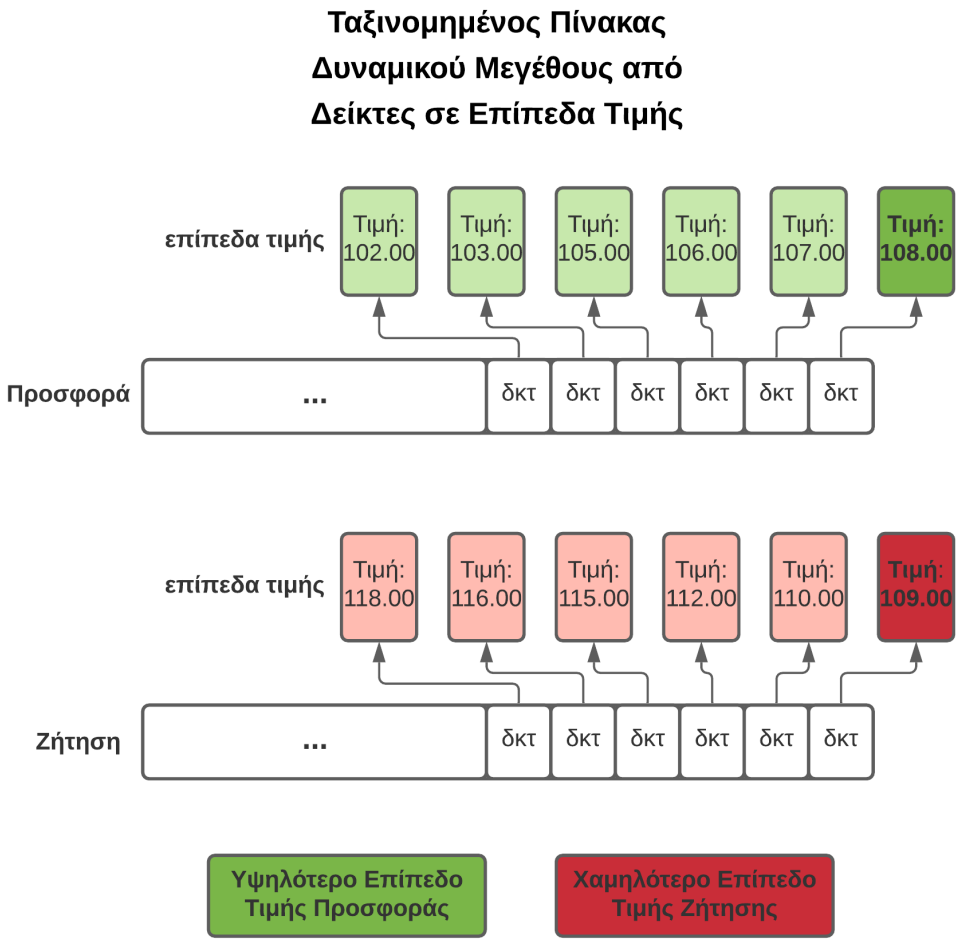
1. Είναι πιο **βολική στο σύστημα πρόβλεψης διακλαδώσεων (branch predictor)** που έχουν όλοι οι σύγχρονοι επεξεργαστές. Ο λόγος είναι ότι η πρόβλεψη των διακλαδώσεων σε μια γραμμική αναζήτηση αποτελεί μια τετριμμένη διαδικασία. Κάθε σύγκριση αποτυγχάνει (επιτυχημένη πρόβλεψη),

εκτός από την τελευταία (αποτυχημένη πρόβλεψη). Αντίθετα οι διακλαδώσεις σε ένα δυαδικό δέντρο προβλέπονται αρκετά πιο δύσκολα και πολλές φορές αποτυγχάνουν, επιφέροντας σημαντικές χρονικές επιβαρύνσεις στον επεξεργαστή.

2. Πετυχαίνει πολύ **καλύτερη χωρική τοπικότητα (spatial locality) στην μνήμη cache** του υπολογιστή, η οποία είναι δεκάδες φορές πιο γρήγορη από την μνήμη (RAM). Αυτό συμβαίνει διότι τα στοιχεία ενός πίνακα αποθηκεύονται σε συνεχείς θέσεις μνήμης και επομένως φορτώνονται μαζικά στην μνήμη cache.
3. Επιπλέον, μια δομή πίνακα σε σχέση με μια δομή δυαδικού δέντρου μπορεί να **βελτιστοποιηθεί πολύ καλύτερα από τον βελτιστοποιητή του μεταγλωττιστή (compiler optimizer)[4].**
4. Τέλος, έπειτα από μετρήσεις στο φορτίο εργασίας βρήκαμε ότι το **μέσο βάθος αναζήτησης, εισαγωγής και διαγραφής ενός επιπέδου τιμής ήταν μόλις 15.** Αυτό σημαίνει ότι στην πραγματικότητα η χρονική πολυπλοκότητα ενός ταξινομημένου πίνακα με αυτό το φορτίο εργασίας είναι πιο κοντά στο O(log N) ή ακόμα και στο O(1) παρά στο O(N). Υπήρχαν βέβαια και σπάνιες περιπτώσεις όπου χρειάστηκε να φτάσουμε μέχρι και σε βάθος 4265 επιπέδων (ουσιαστικά διάσχιση όλου του πίνακα), στις οποίες η δομή πίνακα υποαπέδιδε σε σχέση με την δομή δέντρου.

Έτσι αντικαταστήσαμε την δομή του δέντρου AVL (σχήμα 3.6) με έναν **ταξινομημένο πίνακα δυναμικού μεγέθους από δείκτες σε επίπεδα τιμής (sorted dynamic sized array of pointers to price levels)**, όπως φαίνεται και στο σχήμα 3.7.

[4] Οι βελτιστοποιήσεις που πραγματοποιεί ένας σύγχρονος μεταγλωττιστής είναι πολλές και σύνθετες και βρίσκονται εκτός του εύρους έρευνας αυτής της εργασίας.



Σχήμα 3.7: Αντικατάσταση της δομής δέντρου AVL με ταξινομημένο πίνακα δυναμικού μεγέθους από δείκτες. Η αναζήτηση σε έναν τέτοιο πίνακα ξεκινάει από το τέλος του προς την αρχή. Ο πίνακας προσφοράς (bids) είναι ταξινομημένος σε φθίνουσα σειρά (τέλος προς αρχή), ενώ ο πίνακας ζήτησης (asks) είναι ταξινομημένος σε αύξουσα σειρά (τέλος προς αρχή). Τα «καλύτερα» επίπεδα τιμής βρίσκονται πάντα στο τέλος του εκάστοτε πίνακα.

Επιπλέον, μια αλγοριθμική βελτίωση που πετύχαμε ήταν οι ενέργειες της αναζήτησης και της εισαγωγής σε έναν πίνακα να γίνονται με μια επανάληψη (iteration) του πίνακα, μιας και είναι ταξινομημένος. Αντίθετα, στην υλοποίηση του δέντρου AVL έπρεπε να γίνουν δύο επαναλήψεις σε περίπτωση εισαγωγής επιπέδου τιμής: μία για την αναζήτηση του επιπέδου, και αφού δεν βρέθηκε, μια για την εισαγωγή του νέου επιπέδου. Βέβαια, πρέπει να έχουμε υπ' όψη μας, ότι η εισαγωγή ή η διαγραφή ενός στοιχείου του πίνακα, εκτός του τέλους και της αρχής, έχει το επιπλέον κόστος της μετανάθεσης των στοιχείων που βρίσκονται δεξιά του εισαγόμενου ή προς διαγραφή στοιχείου κατά μια θέση.

3.5 Ταξινομημένος Πίνακας Δυναμικού Μεγέθους από Δομές Τύπου Struct

Μετά και την αντικατάσταση του δέντρου AVL με ένα ταξινομημένο πίνακα δυναμικού μεγέθους, κάναμε εκ νέου profiling, όπως φαίνεται και στο σχήμα 3.8. Εκεί παρατηρήσαμε ότι η αναζήτηση (+ εισαγωγή πλέον), καταλάμβανε από το περίπου 25% του χρόνου πλέον το 15.6% το οποίο φαίνεται να αποτελεί σημαντική βελτίωση. Η συνάρτηση της διαγραφής ενός επιπέδου τιμής όμως, από εκεί που καταλάμβανε μόλις το 2.9% του συνολικού χρόνου (συνάρτηση erase, σχήμα 3.5), έφτασε να καταλαμβάνει το 13.9% του συνολικού χρόνου (συνάρτηση DeleteLevel, σχήμα 3.8), καθώς η χρονική πολυπλοκότητα από O(1) μετατράπηκε σε O(N). Έπρεπε, λοιπόν, να βελτιώσουμε εκ νέου τη δομή αυτή.

```
Total: 19382 samples
    3018  15.6%  15.6%     3159  16.3% CppTrader::Matching::OrderBook::FindInsertLevel
    2698  13.9%  29.5%     2935  15.1% CppTrader::Matching::OrderBook::DeleteLevel
    1890   9.8%  39.2%     1931  10.0% CppCommon::HashMap::find (inline)
    1886   9.7%  49.0%     2003  10.3% CppCommon::HashMap::emplace_internal (inline)
    1604   8.3%  57.2%     1604   8.3% std::vector::size (inline)
    1433   7.4%  64.6%     1433   7.4% __read
    1429   7.4%  72.0%     4645  24.0% CppTrader::Matching::OrderBook::DeleteOrder
    1339   6.9%  78.9%     1518   7.8% CppTrader::Matching::MarketManager::MatchOrder
     829   4.3%  83.2%      829   4.3% CppCommon::List::push_back (inline)
```

Σχήμα 3.8: Οι συναρτήσεις FindInsertLevel και DeleteLevel που είναι υπεύθυνες για την αναζήτηση/εισαγωγή και διαγραφή ενός επιπέδου τιμής καταλαμβάνουν τις δύο πρώτες θέσεις στις μετρήσεις του profiling και αποτελούν τις μεγαλύτερες στενωπούς απόδοσης.

Το πρόβλημα με την τρέχουσα υλοποίηση είναι ότι δεν αξιοποιεί την καλή χωρική τοπικότητα που πετυχαίνουν οι πίνακες στην μνήμη cache, μιας και σε κάθε επανάληψη (iteration) του πίνακα για να ελέγξουμε την τιμή ενός επιπέδου τιμής πρέπει να αναφερθούμε (reference) σε αυτό μέσου του δείκτη του (pointer) στην μνήμη. Το κάθε επίπεδο τιμής βρίσκεται σε τυχαίες θέσεις στην μνήμη, οπότε χάνουμε ουσιαστικά την χωρική τοπικότητα. Για να λύσουμε αυτό το πρόβλημα, αντί για δείκτες σε επίπεδα τιμής, σκεφτήκαμε να αποθηκεύουμε δομές τύπου struct της μορφής:

Struct{

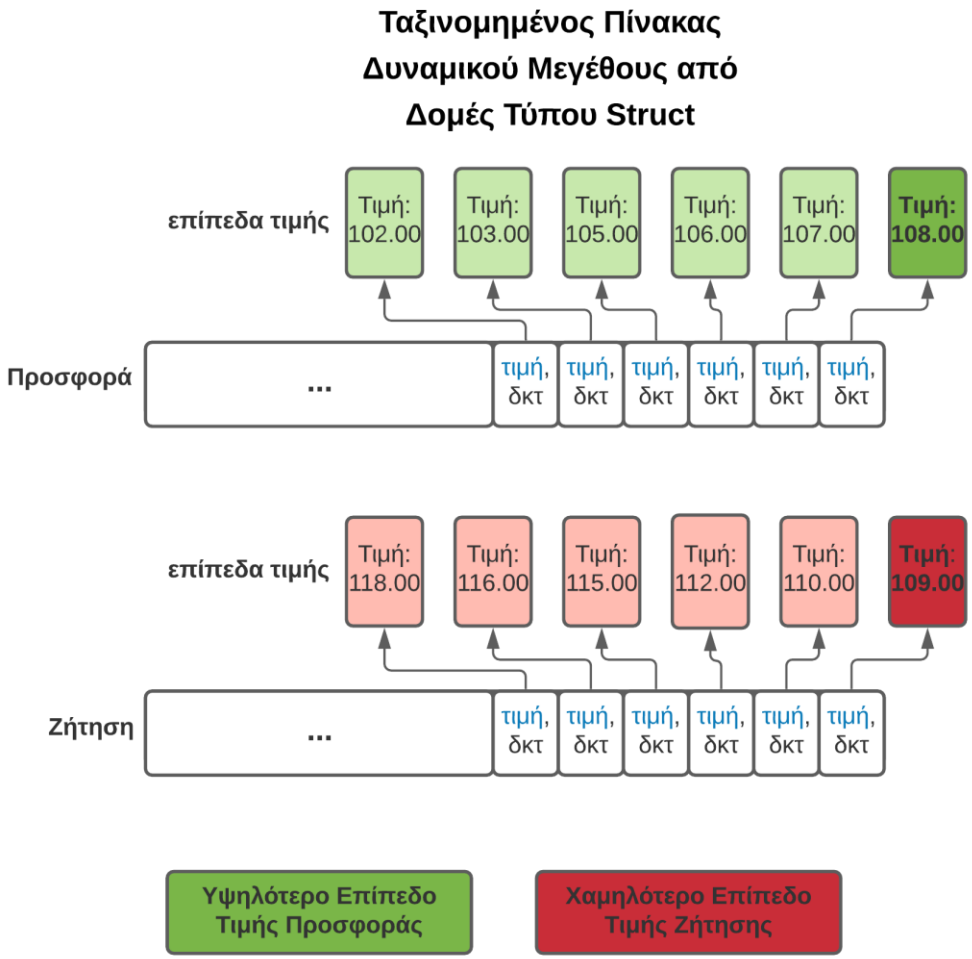
Τιμή (price) του επιπέδου τιμής

Δείκτης (pointer) στο επίπεδο τιμής

}

Η νέα αυτή δομή ονομάζεται **ταξινομημένος πίνακας δυναμικού μεγέθους από δομές τύπου struct (sorted dynamic sized array of structs),** και η διαφορά της με την αρχική υλοποίηση του πίνακα φαίνεται στο σχήμα 3.9. Έτσι, η τιμή κάθε επιπέδου αποθηκεύεται διπλά, μία στο ίδιο το επίπεδο και μία στον πίνακα. Αυτό σημαίνει ότι

χρειάζεται να δεσμεύσουμε ελάχιστα παραπάνω χώρο μνήμης (μερικά MB συνολικά). Ο επιπλέον αυτός χώρος, όμως, σε καμία περίπτωση δεν μας επηρεάζει και ούτε συγκρίνεται με τα πολλαπλά οφέλη σε θέμα απόδοσης που πετυχαίνουμε.



Σχήμα 3.9: Χρήση δομών τύπου struct αντί για δείκτες ως στοιχεία του πίνακα, όπου το struct περιέχει τον δείκτη στο κατάλληλο επίπεδο τιμής, αλλά και την τιμή αυτού του επιπέδου για ταχύτερη πρόσβαση σε αυτήν. Στο σχήμα με μπλε γράμματα φαίνονται οι αλλαγές που έγιναν.

Πραγματοποιώντας ένα ακόμα profiling, μετά και την βελτίωση της δομής πίνακα ώστε να αποτελείται από δομές τύπου struct, παρατηρούμε ότι οι συναρτήσεις τις αναζήτησης/εισαγωγής (find/insert) και της διαγραφής (delete) σε έναν τέτοιο πίνακα ξοδεύουν τώρα το 3.7% και το 2.6% του συνολικού χρόνου αντίστοιχα και δεν αποτελούν πλέον σοβαρές στενωπούς απόδοσης (σχήμα 3.10).

```
Total: 16066 samples
    1861  11.6%  11.6%     1945  12.1% CppCommon::HashMap::emplace_internal (inline)
    1798  11.2%  22.8%     2909  18.1% CppTrader::Matching::OrderBook::DeleteOrder
    1735  10.8%  33.6%     1780  11.1% CppCommon::HashMap::find (inline)
    1642  10.2%  43.8%     1642  10.2% std::vector::size (inline)
    1507   9.4%  53.2%     1664  10.4% CppTrader::Matching::MarketManager::MatchOrder
    1246   7.8%  60.9%     1246   7.8% __read
     823   5.1%  66.1%     2696  16.8% CppTrader::Matching::OrderBook::AddOrder
     808   5.0%  71.1%      808   5.0% CppCommon::List::push_back (inline)
     590   3.7%  74.8%     1019   6.3% CppTrader::Matching::OrderBook::FindInsertLevel
     415   2.6%  77.3%      867   5.4% CppTrader::Matching::OrderBook::DeleteLevel
```

Σχήμα 3.10: Οι συναρτήσεις FindInsertLevel και DeleteLevel δεν αποτελούν πλέον στενωπούς απόδοσης του συστήματος. Αντίθετα, οι συναρτήσεις emplace_internal (insert) και find του πίνακα κατακερματισμού (hashmap) συνολικά αποτελούν πλέον τη μεγαλύτερη στενωπό απόδοσης του CppTrader.

3.6 Πίνακας από Δείκτες

Οι εντολές μείωσης, αντικατάστασης και διαγραφής για να αποκτήσουν γρήγορα πρόσβαση στην οριακή εντολή που θέλουν να τροποποιήσουν/διαγράψουν χρησιμοποιούν την δομή του πίνακα κατακερματισμού (hashmap). Η κατασκευή αυτή συνοπτικά βασίζεται στη δομή πίνακα και στον ορισμό μίας συνάρτησης κατακερματισμού και αποτελεί μια πολύ αποδοτική δομή για την αποθήκευση δεδομένων τύπου ζεύγος κλειδιού-τιμής (key-value pair) και την πρόσβασή τους χωρίς να είναι ταξινομημένα. Το ζεύγος κλειδιού-τιμής που αποθηκεύουμε στην υλοποίησή μας περιέχει ως κλειδί το αναγνωριστικό μιας εντολής, και ως τιμή τον δείκτη σε αυτήν (id/pointer). Η συνάρτηση κατακερματισμού, από την μεριά της, χρησιμοποιεί το κλειδί για να υπολογίσει την θέση του πίνακα που θα αποθηκευτεί το ζεύγος κλειδιού-τιμής. Όταν θέλουμε να αναζητήσουμε ένα τέτοιο ζεύγος το μόνο που πρέπει να ξέρουμε είναι το κλειδί και με χρονική πολυπλοκότητα κατά μέσο όρο O(1) μπορούμε να αποκτήσουμε πρόσβαση στο ζεύγος. Η πολυπλοκότητα είναι ίδια και για την εισαγωγή αλλά και για την διαγραφή. Βέβαια πολλές φορές προκύπτουν συγκρούσεις κατά την εισαγωγή γιατί η θέση ενός ζεύγους κλειδιού-τιμής στον πίνακα μπορεί να είναι ίδια και για διαφορετικά κλειδιά σε ζεύγη (αυτός είναι και ο λόγος που η χρονική πολυπλοκότητα είναι *περίπου* O(1)). Για να αντιμετωπίσουμε αυτό το πρόβλημα αναθέτουμε το ζεύγος κλειδιού-τιμής στην πρώτη ελεύθερη θέση του πίνακα δεξιά από την θέση που προοριζόταν αρχικά, αν αυτή η θέση είναι κατειλημμένη. Για να περιορίσουμε όσο μπορούμε αυτές τις συγκρούσεις προσπαθούμε το μέγεθος του πίνακα κατακερματισμού να είναι αρκετά μεγαλύτερο απ' όσα ζεύγη έχουμε αποθηκευμένα σε αυτόν. Στην περίπτωση μας φροντίζουμε ο πίνακας να έχει πληρότητα το πολύ 50%.

Η δομή αυτή για την άμεση πρόσβαση των εντολών θεωρητικά φαίνεται να είναι ιδανική (O(1) χρονική πολυπλοκότητα). Όμως αντιμετωπίζει ορισμένα προβλήματα, το οποίο φαίνεται και από τα αποτελέσματα του profiling (σχήμα 3.10), καθώς οι

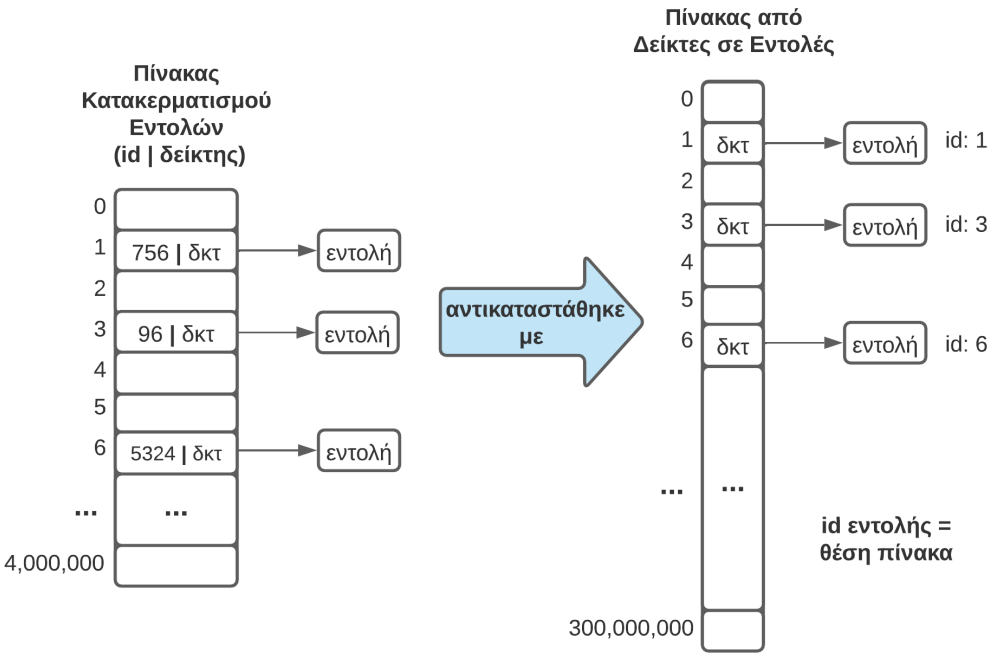
συναρτήσεις αναζήτησης και εισαγωγής αποτελούν σημαντικές στενωπούς της απόδοσης του συστήματος. Αυτά τα **προβλήματα** οφείλονται σε τρεις λόγους:

1. Αρχικά στις **πιθανές συγκρούσεις** που συμβαίνουν κατά την εισαγωγή ενός στοιχείου.
2. Επίσης, μιας και η δομή αυτή χρησιμοποιείται εκατομμύρια φορές από όλους τους τύπους εντολών (και από τις οριακές). ακόμα και η μικρή **επιβάρυνση του κατακερματισμού του κλειδιού** για αναζήτηση, εισαγωγή ή διαγραφή ενός ζεύγους κλειδιού-τιμής αθροιστικά επιφέρει σημαντική επιβάρυνση στο σύστημα.
3. Τέλος, ένα, ακόμα, βασικό μειονέκτημα αυτής της δομής είναι ότι **δεν εκμεταλλεύεται την μνήμη cache**. Αυτό συμβαίνει διότι τα αναγνωριστικά (id) των εντολών ενός αρχείου Nasdaq ITCH παράγονται κατά βάση σειριακά. Επίσης πρόσφατα ληφθείσες εντολές συνήθως αλληλοεπιδρούν με άλλες πρόσφατα ληφθείσες εντολές. Στον πίνακα κατακερματισμού όμως, εντολές που έχουν κοντινούς αριθμούς αναγνωριστικών (id), αποθηκεύονται με τυχαίο τρόπο. Αντίθετα, σε έναν απλό πίνακα, εντολές με κοντινούς αριθμούς αναγνωριστικών (id), αποθηκεύονται στις αντίστοιχες κοντινές θέσεις στον πίνακα, πετυχαίνοντας έτσι πολύ καλύτερη **χωρική τοπικότητα (spatial locality) στην μνήμη cache.**

Η ιδέα, λοιπόν, από την στιγμή που κάθε εντολή ενός αρχείου τύπου Nasdaq ITCH έχει μοναδικό αναγνωριστικό (id) είναι, αντί για πίνακα κατακερματισμού να χρησιμοποιήσουμε έναν **πίνακα από δείκτες (array of pointers)** αρκετά μεγάλου μεγέθους ώστε να μην υποστεί υπερχείλιση, όσες εντολές και να λάβει το CppTrader (σχήμα 3.11). Ουσιαστικά κάθε θέση του πίνακα μπορεί να χρησιμοποιηθεί από μια και μοναδική εντολή και ισχύει:

θέση πίνακα (array index) = αναγνωριστικό εντολής (order id)

Αυτή η λύση θα μπορούσε να εξαλείψει τις επιβαρύνσεις της συνάρτησης κατακερματισμού και να εκμεταλλευτεί πολύ καλύτερα την μνήμη cache, σε βάρος βέβαια της χρήσης επιπλέον μνήμης. Για παράδειγμα, για το συγκεκριμένο φορτίο εργασίας έπρεπε να ορίσουμε στον πίνακα περίπου 300 εκατομμύρια θέσεις σε αντίθεση με τις περίπου 8 εκατομμύρια θέσεις που έφτασε στην αιχμή του ο πίνακας κατακερματισμού με το ίδιο φορτίο. Βλέπουμε δηλαδή μια αύξηση της τάξεως του 3750% στην χρήση μνήμης από την μνήμη που χρησιμοποιεί ο πίνακας κατακερματισμού. Στο παράδειγμά μας και πάλι, ένας τέτοιος πίνακας 300 εκατομμυρίων θέσεων περίπου δεσμεύει περίπου 2.4 GB μνήμης.



Σχήμα 3.11: Αντικατάσταση του πίνακα κατακερματισμού με πίνακα από δείκτες σε εντολές, αρκετά μεγάλου μεγέθους ώστε να μην υποστεί υπερχείλιση από τον τεράστιο όγκο των εντολών.

Κάνοντας ένα τελευταίο profiling (σχήμα 3.12), μετά την αντικατάσταση του πίνακα κατακερματισμού με απλό πίνακα, παρατηρούμε ότι οι συναρτήσεις αναζήτησης και εισαγωγής μιας εντολής στον πίνακα αυτό ξοδεύουν τόσο λίγο συνολικό χρόνο συστήματος που δεν είναι μέσα στα κορυφαία 10 αποτελέσματα του profiling. Πλέον η σημαντικότερη στενωπός απόδοσης του συστήματος είναι η συνάρτηση OrderBook::DeleteOrder, που είναι υπεύθυνη για την διαγραφή μιας εντολής από το βιβλίο εντολών της. Δεν θα την μελετήσουμε σε αυτήν την εργασία.

```
Total: 13874 samples
    1831  13.2%  13.2%     2810  20.3% CppTrader::Matching::OrderBook::DeleteOrder
    1519  10.9%  24.1%     1519  10.9% std::vector::size (inline)
    1506  10.9%  35.0%     1506  10.9% __read
    1496  10.8%  45.8%     1673  12.1% CppTrader::Matching::MarketManager::MatchOrder
    1353   9.8%  55.5%     5145  37.1% CppTrader::Matching::MarketManager::DeleteOrder (inline)
     933   6.7%  62.3%     2931  21.1% CppTrader::Matching::OrderBook::AddOrder
     862   6.2%  68.5%      862   6.2% CppCommon::List::push_back (inline)
     603   4.3%  72.8%     1050   7.6% CppTrader::Matching::OrderBook::FindInsertLevel
     509   3.7%  76.5%     1669  12.0% CppTrader::Matching::MarketManager::ReplaceOrder [clone .part.0]
     387   2.8%  79.3%      761   5.5% CppTrader::Matching::OrderBook::DeleteLevel
```

Σχήμα 3.12: Οι συναρτήσεις αναζήτησης και εισαγωγής μιας εντολής σε πίνακα δεν αποτελούν πλέον στενωπούς απόδοσης.

Κεφάλαιο 4.

Πειραματική Αξιολόγηση Απόδοσης

Σε αυτό το κεφάλαιο θα ελέγξουμε πειραματικά τις βελτιώσεις που έγιναν στο προηγούμενο κεφάλαιο, καθώς και αν αυξήθηκε και κατά πόσο η απόδοση του CppTrader.

4.1 Πειραματικό Περιβάλλον

Αρχικά, όλα τα πειράματα της εργασίας έγιναν στο υπολογιστικό σύστημα με τα χαρακτηριστικά του πίνακα 4.1 και με βάση το φορτίο του κεφαλαίου 3.2.

Αρχιτεκτονική CPU	**Intel(R) Core(TM) i5-6400 CPU @ 2.7GHz**
Λογικοί πυρήνες CPU	**3**
Φυσικοί πυρήνες CPU	**3**
Συνολική μνήμη RAM	**11.8 GB**
Ελεύθερη μνήμη RAM	**9.8 GB**
Μνήμη swap	**1.4 GB**
Μηχανή	**Vmware 15.6**
Έκδοση λειτουργικού συστήματος	**Linux**
Πυρήνας (kernel)	**5.8.0-48-generic x86_64**
Διανομή (Distro)	**Ubuntu 20.04.2 LTS (Focal Fossa)**
Bits λειτουργικού συστήματος	**64-bit**

Πίνακας 4.1: Χαρακτηριστικά hardware και λειτουργικού συστήματος που πραγματοποιήθηκαν τα πειράματα.

Όσο αναφορά την έκδοση της γλώσσας προγραμματισμού, χρησιμοποιήσαμε την **C++17** με έκδοση μεταγλωττιστή (compiler) **gcc 9.3.0 (for x86_64-linux-gnu).** Όλα τα πειράματα εκτελέστηκαν με τις ακόλουθες παραμέτρους μεταγλωττιστή:

-O3 -ggdb3 -DNDEBUG -Wl,--no-as-needed,-lprofiler,--as-needed

Αξιοσημείωτη είναι η χρήση της παραμέτρου -O3, καθώς συνοψίζει πολλές ακόμα βασικές παραμέτρους [15], που είναι πολύ σημαντικές για την βελτίωση της απόδοσης του CppTrader τουλάχιστον κατά 300%. Η αύξηση αυτή γίνεται μέσω βελτιστοποιήσεων του μεταγλωττιστή. Επίσης, ορισμένες φορές η χρήση -O3 βελτιστοποιήσεων κρύβει κινδύνους (π.χ. διάφορα bugs), σε σχέση με την χρήση -O2. Παρ' όλα αυτά profiling και μετρήσεις που κάναμε με -O2 έδιναν σχεδόν παρόμοια αποτελέσματα. Δεν τίθεται, λοιπόν, θέμα σύγκρισης των δύο σε αυτή την εργασία.

Τέλος, πριν από κάθε πείραμα **ζεστάναμε την μνήμη cache (cache warming),** τρέχοντας δύο διαφορετικά φορτία από αυτό που θα χρησιμοποιούσαμε, για τουλάχιστον 5 λεπτά. Αυτό το κάναμε γιατί το σύστημα επιταχύνονταν μέχρι και 20% με «ζεστή» μνήμη cache. Επιπλέον, διασφαλίζαμε ότι δεν έτρεχαν ενεργά άλλα προγράμματα κατά την διάρκεια των πειραμάτων, ώστε να εξασφαλίσουμε όσο δυνατόν παρόμοιο περιβάλλον εκτέλεσης των πειραμάτων.

4.2 Μετρικές και Τρόποι Χρονομέτρησης

Η μετρική που χρησιμοποιήσαμε για να μετρήσουμε και να συγκρίνουμε την απόδοση των διαφορετικών υλοποιήσεων, δομών, συναρτήσεων, κλπ. είναι η **ρυθμαπόδοση (throughput).** Ουσιαστικά η μετρική αυτή υπολογίζει τις συνολικές ενέργειες που έγιναν σε ένα συγκεκριμένο χρονικό διάστημα προς το χρονικό διάστημα αυτό. Η μετρική αυτή συνήθως μετράτε σε ενέργειες/δευτερόλεπτο.

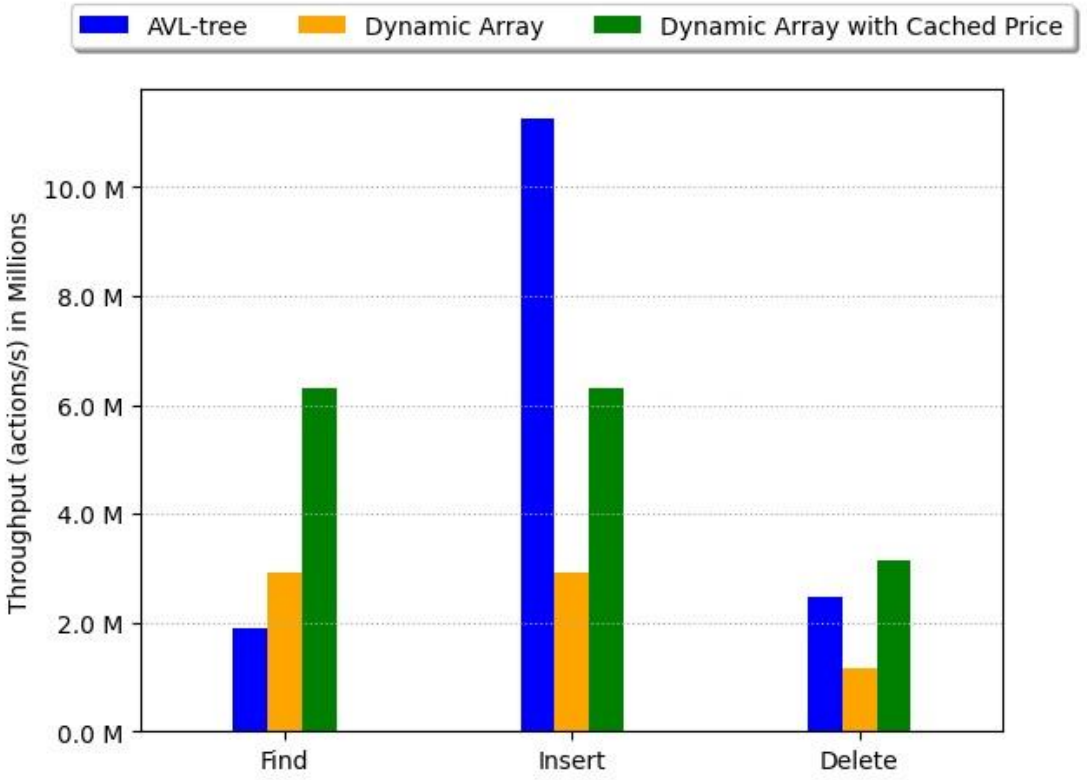
Για την μέτρηση της ρυθμαπόδοσης χρησιμοποιήσαμε χρονοσφραγίδες (timestamps). Για να πάρουμε με υψηλή χρονική ακρίβεια μια χρονοσφραγίδα χρησιμοποιήσαμε την συνάρτηση clock_gettime() και το ρολόι CLOCK_MONOTONIC της βιβλιοθήκης time.h της C. Η συνάρτηση και το ρολόι αυτό έχουν ακρίβεια χρονοσφραγίδας σε επίπεδο νανο-δευτερολέπτων. Ο αλγόριθμος χρονομέτρησης της ρυθμαπόδοσης μιας περιοχής κώδικα με την συνάρτηση clock_gettime() έχει ως εξής:

1. Κλήση της συνάρτησης ακριβώς στο σημείο που ξεκινά η περιοχή του κώδικα που επιθυμούμε να χρονομετρήσουμε.
2. Κλήση της συνάρτησης ακριβώς μετά το σημείο που ολοκληρώνεται η περιοχή του κώδικα που επιθυμούμε να χρονομετρήσουμε.

3. Αφαίρεση της αρχικής τιμής από την δεύτερη ώστε να υπολογιστεί ο χρόνος που χρειάστηκε για την εκτέλεση του κώδικα που μας ενδιαφέρει να χρονομετρήσουμε.
4. Επανάληψη των βημάτων 1-3 κάθε φορά που χρησιμοποιούνταν η συγκεκριμένη περιοχή κώδικα σε μια εκτέλεση του CppTrader.
5. Διαίρεση του πλήθους των επαναλήψεων που χρησιμοποιήθηκε αυτή η περιοχή κώδικα με τον άθροισμα όλων των χρόνων εκτέλεσής του.

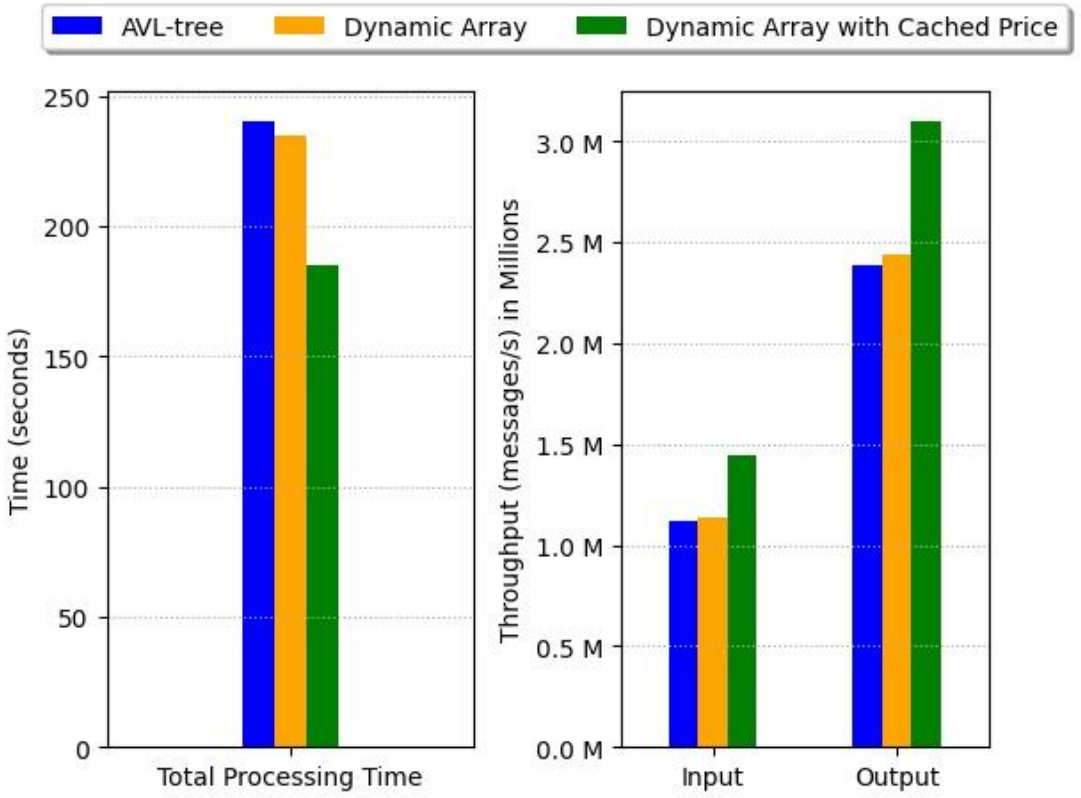
4.3 Αξιολόγηση Αντικατάστασης Δομής Δέντρου AVL με Ταξινομημένο Πίνακα Δυναμικού Μεγέθους

Αρχικά στο σχήμα 4.1 παρουσιάζονται και συγκρίνονται οι αποδόσεις σε ρυθμαπόδοση που πετυχαίνουν οι συναρτήσεις αναζήτησης (find), εισαγωγής (insert) και διαγραφής (delete) ενός επιπέδου τιμής στην εκάστοτε δομή. Παρατηρούμε, λοιπόν, ότι η ρυθμαπόδοση της αναζήτησης, που ήταν και η μεγαλύτερή μας στενωπός απόδοσης, υπερ-τριπλασιάζεται με την χρήση δυναμικού πίνακα από δομές τύπου struct (dynamic array with cached price στο σχήμα 4.1 και 4.2) σε σχέση με την δομή δέντρου AVL. Παρατηρούμε, επίσης ότι η ρυθμαπόδοση της εισαγωγής είναι πολύ καλύτερη στην δομή δέντρου AVL, από ότι σε οποιαδήποτε δομή πίνακα. Τέλος, η ρυθμαπόδοση της διαγραφής χειροτερεύει, αισθητά, με την χρήση πίνακα δυναμικού μεγέθους από δείκτες, αλλά βελτιώνεται συνολικά με την χρήση πίνακα από δομές τύπου struct.



Σχήμα 4.1: Η απόδοση των συναρτήσεων με βάση την δομή από την οποία υλοποιούνται. Σημείωση ότι στις υλοποιήσεις με δομή πίνακα οι συναρτήσεις αναζήτησης (find) και εισαγωγής (insert) γίνονται ταυτόχρονα από μια συνάρτηση (δηλαδή η αναζήτηση περιέχει και την εισαγωγή και το αντίστροφο), για αυτό και έχουν την ίδια απόδοση και στις δύο περιπτώσεις.

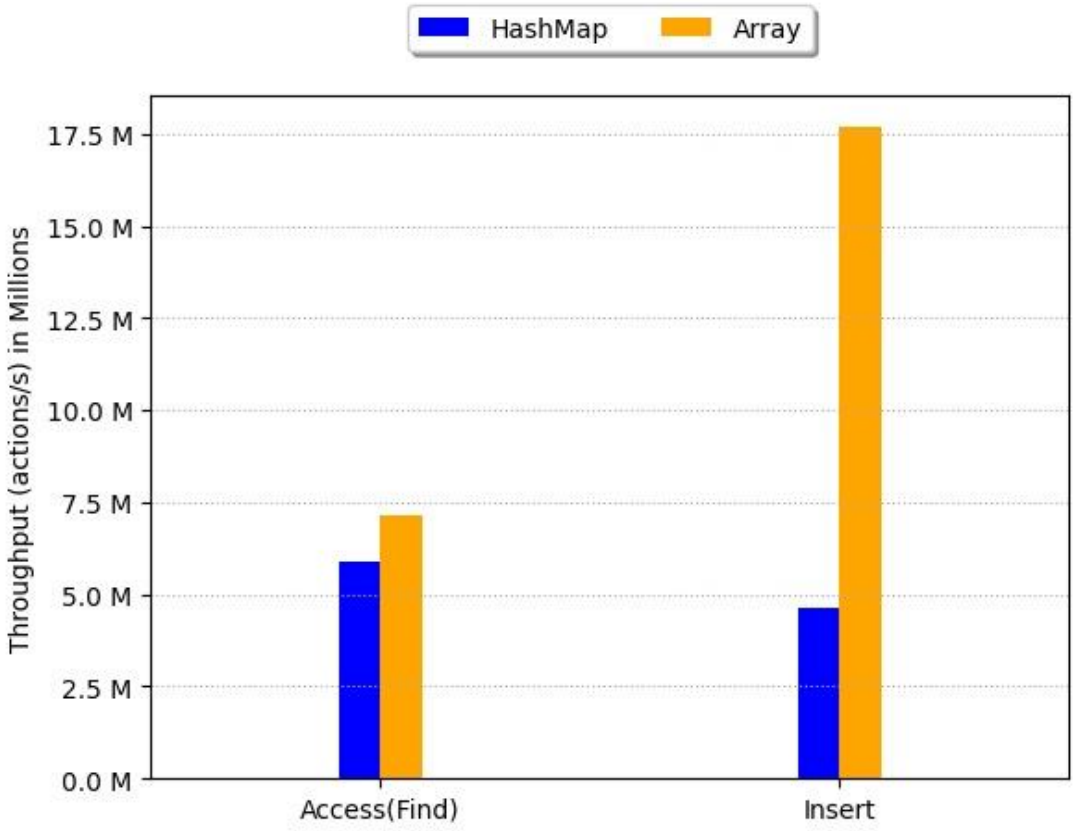
Για να καταλάβουμε την επίδραση στην συνολική απόδοση του συστήματος που πετυχαίνει η κάθε δομή μπορούμε να δούμε το σχήμα 4.2, όπου περιέχει στα αριστερά τον συνολικό χρόνο εκτέλεσης του συστήματος και στα δεξιά την ρυθμαπόδοση εισόδου και εξόδου του συστήματος. Παρατηρούμε, λοιπόν, ότι και στους δυο αυτούς πίνακες ελάχιστη βελτίωση στην απόδοση προσφέρει η αντικατάσταση του δέντρου AVL με έναν ταξινομημένο πίνακα δυναμικού μεγέθους από δείκτες. Αντίθετα, η χρήση ταξινομημένου πίνακα δυναμικού μεγέθους από δομές τύπου struct φαίνεται να βελτιώνει σημαντικά την συνολική απόδοση.



Σχήμα 4.2: Αριστερά βλέπουμε τον συνολικό χρόνος εκτέλεσης, ενώ δεξιά βλέπουμε την ρυθμαπόδοση της εισόδου (input) και της εξόδου (output) του CppTrader με βάση την δομή που χρησιμοποιήθηκε για την αποθήκευση των επιπέδων τιμής.

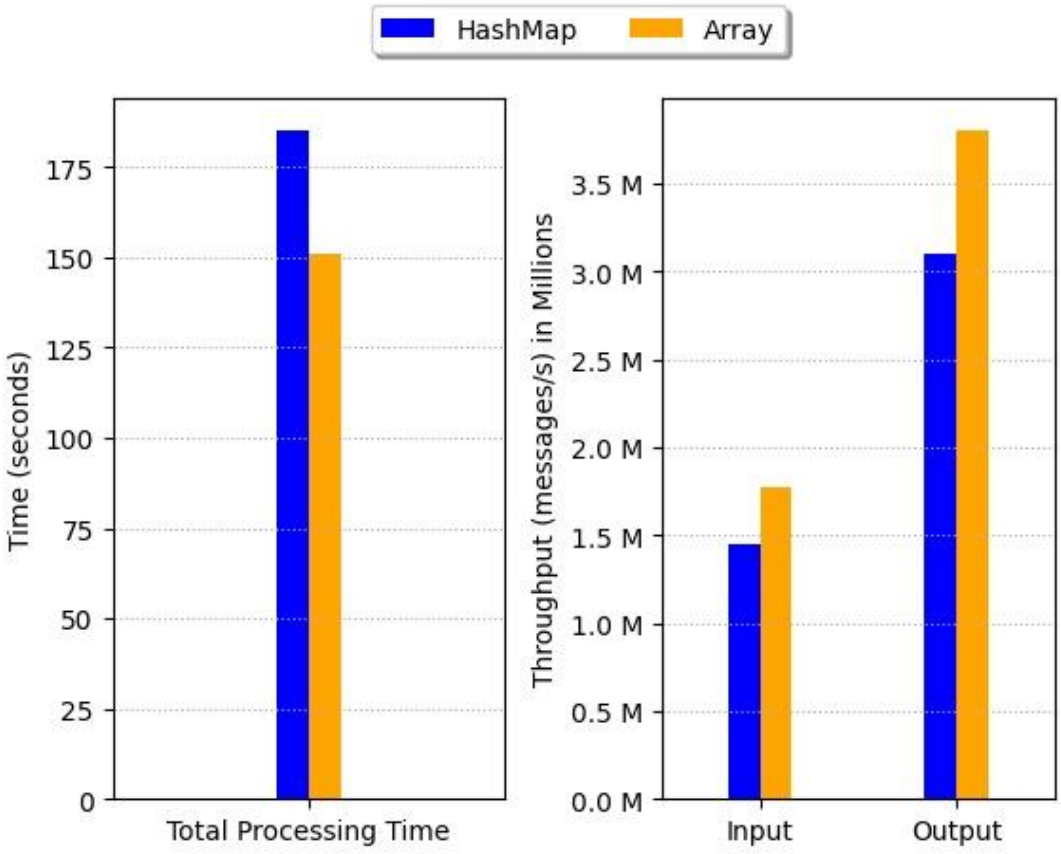
4.4 Αξιολόγηση Αντικατάστασης Δομής Πίνακα Κατακερματισμού με Πίνακα

Αναφορικά με τις συναρτήσεις αναζήτησης (find) και εισαγωγής (insert) του πίνακα κατακερματισμού, που αποτελούσαν συνολικά τη δεύτερη μεγαλύτερη στενωπό απόδοσης του συστήματος, στο σχήμα 4.3 παρουσιάζεται η ρυθμαπόδοση τους με βάση τις δύο δομές υλοποίησης τους. Παρατηρούμε ότι με την χρήση απλού πίνακα αντί του πίνακα κατακερματισμού η ρυθμαπόδοση της εισαγωγής αυξήθηκε πάνω από 3.5 φορές, ενώ αρκετά μικρότερη, αν και διόλου ευκαταφρόνητη, ήταν και η αύξηση της αναζήτησης.



Σχήμα 4.3: Η απόδοση των συναρτήσεων με βάση την δομή από την οποία υλοποιούνται.

Φαίνεται λοιπόν ότι η δομή του πίνακα αύξησε την συνολική απόδοση του συστήματος. Το κατά πόσο αυξήθηκε παρουσιάζεται στο σχήμα 4.4, όπου περιέχει στα αριστερά τον συνολικό χρόνο εκτέλεσης του συστήματος και στα δεξιά την ρυθμαπόδοση εισόδου και εξόδου του συστήματος. Παρατηρούμε λοιπόν, ότι η δομή του απλού πίνακα βελτιώνει σημαντικά την συνολική απόδοση του CppTrader.



Σχήμα 4.4: Αριστερά βλέπουμε τον συνολικό χρόνος εκτέλεσης, ενώ δεξιά βλέπουμε την ρυθμαπόδοση της εισόδου (input) και της εξόδου (output) του CppTrader, με βάση την δομή που χρησιμοποιήθηκε για την άμεση πρόσβαση των εντολών.

4.5 Συνολική Αξιολόγηση των Βελτιώσεων του CppTrader

Στον πίνακα 4.2 συνοψίζονται οι δείκτες απόδοσης του συστήματος πριν και μετά τις αλλαγές στην δομή δέντρου AVL και πίνακα κατακερματισμού με ταξινομημένο πίνακα δυναμικού μεγέθους από δομές τύπου struct και με απλό πίνακα αντίστοιχα. Επίσης στον πίνακα 4.2 παρουσιάζεται και η συνολική ποσοτική βελτίωση που απέφεραν οι παραπάνω αλλαγές στο CppTrader.

	Πριν	Μετά	Βελτίωση
Συνολικός χρόνος εκτέλεσης	4 λεπτά	2:31 λεπτά	**+37%** (γρηγορότερα)
Ρυθμαπόδοση εισόδου	1,115,466 μνμ/δευτ	1,775,058 μνμ/δευτ	**+59%**
Ρυθμαπόδοση εξόδου	2,386,557 ενημ/δευτ	3,797,764 ενημ/δευτ	**+59%**

Πίνακας 4.2: Δείκτες απόδοσης συστήματος πριν και μετά τις αλλαγές στις αντίστοιχες δομές. Ενώ η βελτίωση στην ρυθμαπόδοση εισόδου και εξόδου αγγίζει το 60%, ο συνολικός χρόνος εκτέλεσης βελτιώθηκε μόνο κατά 37%. Δεν έχουν λοιπόν ένα προς ένα συσχέτιση.

Κεφάλαιο 5.

Μελλοντικές Βελτιώσεις και Συμπεράσματα

Στο παρόν κεφάλαιο, θα παρουσιάσουμε τα βασικά συμπεράσματα της ερευνάς μας και θα προτείνουμε συγκεκριμένες μελλοντικές βελτιώσεις για το CppTrader.

5.1 Μελλοντικές Βελτιώσεις

Όσο αναφορά, πρώτα, τις μελλοντικές βελτιώσεις, δεν θα προτείναμε ιδιαίτερα την ενασχόληση με τις κορυφαίες συναρτήσεις που υπέδειξαν τα αποτελέσματα του τελευταίου profiling (σχήμα 3.12), μιας και αυτές είναι ήδη αρκετά βελτιστοποιημένες. Επίσης, στην περίπτωση παραδείγματος χάριν της συνάρτησης vector::size(), ενώ φαίνεται ότι αποτελεί στενωπό απόδοσης του συστήματος, στην πραγματικότητα είναι πλασματικό αποτέλεσμα που προκλήθηκε από -O3 βελτιστοποιήσεις του μεταγλωττιστή. Αντίθετα, ιδιαίτερη έμφαση πιστεύουμε χρειάζεται να δοθεί στις επόμενες δύο υλοποιήσεις που θα σας παρουσιάσουμε.

5.1.1 Πίνακας από Δομές Τύπου Struct σε Σταθερά Επίπεδα Τιμής

Με βάση προηγούμενες μελέτες [11, 12], για την αποθήκευση επιπέδων τιμής, μια πιθανή βελτίωση θα μπορούσε να είναι η χρήση **πινάκων από δομές τύπου struct σε σταθερά επίπεδα τιμής (array of structs with fixed price levels),** όπου struct:

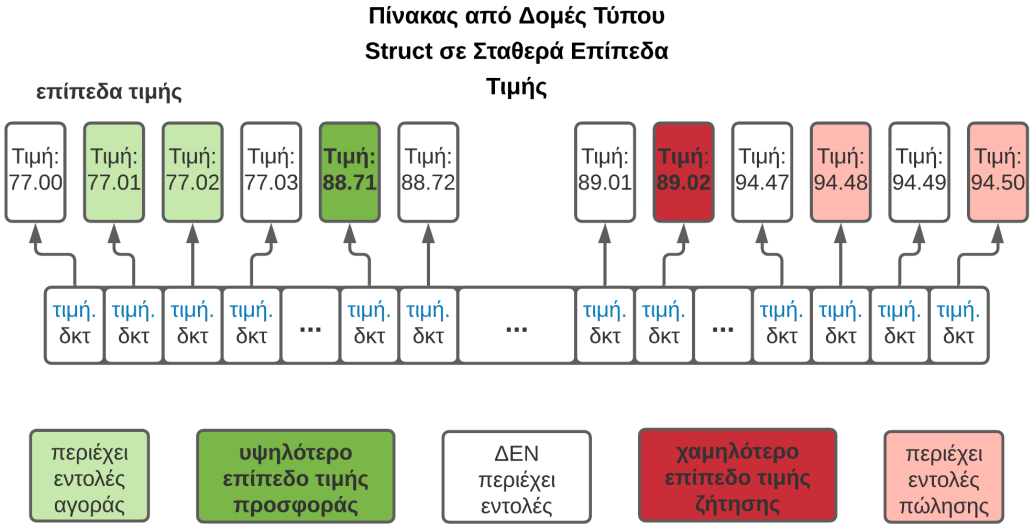
Struct{

Τιμή (price) του επιπέδου τιμής

Δείκτης (pointer) στο επίπεδο τιμής

}

Η διαφορά αυτής της υλοποίησης με του ταξινομημένου πίνακα δυναμικού μεγέθους από δομές τύπου struct του κεφαλαίου 3.5 είναι ότι σε αυτήν την περίπτωση τα επίπεδα τιμής είναι ήδη καθορισμένα για κάθε βιβλίο εντολών μέσα σε ένα εύλογο εύρος [κάτω όριο, πάνω όριο] που μπορεί να πάρει η τιμή μιας μετοχής σε μια μέρα. Επιπλέον, οι τιμές σε αυτό το όριο είναι ομοιόμορφα κατανεμημένες και η μικρότερη διαφορά μεταξύ δύο τέτοιων επιπέδων ονομάζεται price tick. Όσο πιο μικρή είναι αυτή η τιμή τόσο μεγαλύτερος πρέπει να είναι ο πίνακας, αλλά τόσο πιο πολλά δεκαδικά ψηφία μπορεί να περιλαμβάνει η τιμή ενός επιπέδου τιμής και αντίστροφα. Η θεωρητική υλοποίηση της δομής αυτής φαίνεται στο σχήμα 5.1.



Σχήμα 5.1: Σε αυτό το παράδειγμα το price tick είναι 0.01. Το κάτω όριο τιμής που μπορεί να πάρει ένα επίπεδο τιμής είναι 77.00 και το άνω όριο είναι 94.50. Και αυτή η δομή αποθηκεύει την τιμή ενός επιπέδου τιμής σε δομή struct για ταχύτερη πρόσβαση.

Σε αυτή τη δομή, η συνάρτηση της αναζήτησης (find) έχει χρονική πολυπλοκότητα O(1). Για παράδειγμα, ας πούμε ότι θέλουμε να αναζητήσουμε το επίπεδο τιμής με τιμή 77.02 με βάση το σχήμα 5.1. Η θέση (index) του επιπέδου αυτού στον πίνακα υπολογίζεται ως εξής:

θέση πίνακα *= (τιμή επιπέδου τιμής * (1/ price tick)) – (κάτω όριο * (1/ price tick)) = (77.02 * (1/0.01) – (77.00 * (1/0.01)) =* ***2***

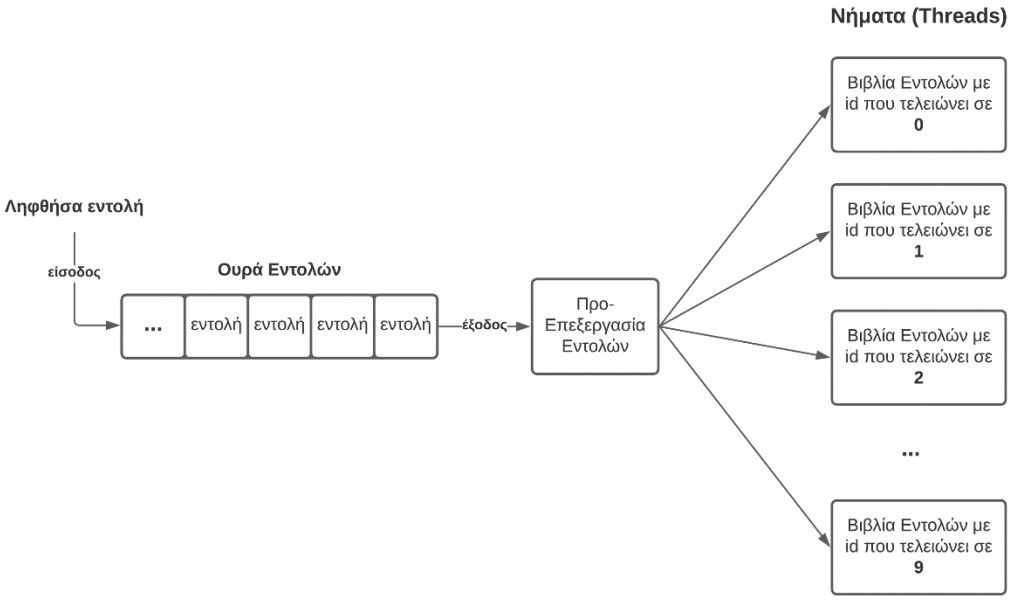
Οι συναρτήσεις της εισαγωγής ή της διαγραφής δεν υφίστανται, μιας και τα επίπεδα τιμής είναι σταθερά (fixed). Όσο αναφορά την πρόσβαση (access) στα καλύτερα

επίπεδα τιμής ζήτησης και προσφοράς από την συνάρτηση ταιριάσματος εντολών, η χρονική πολυπλοκότητα είναι O(1) μόνο για την πρόσβαση ενός επιπέδου. Αν οι εντολές αυτού του επιπέδου ταιριάξουν όλες και πρέπει να αναζητήσουμε καινούργιο καλύτερο επίπεδο τιμής, εδώ η χρονική πολυπλοκότητα στην χειρότερη περίπτωση είναι O(N), όπου N το μέγεθος του πίνακα. Βέβαια η περίπτωση αυτή είναι σπάνια και το νέο επίπεδο τιμής βρίσκεται συνήθως μετά από λίγες προσπελάσεις επιπέδων τιμής του πίνακα. Ακόμα και αυτή η περίπτωση όμως, θα μπορούσε να βελτιωθεί περισσότερο αν αντί για ένα επίπεδο κρατούσαμε παραδείγματος χάριν τα 5 καλύτερα επίπεδα τιμής προσφοράς και τα 5 καλύτερα ζήτησης. Βέβαια το κατά πόσο τα 5 ή κάποιος άλλος αριθμός θα βελτίωναν την απόδοση θέλει και αυτό αρκετά πειράματα και μελέτη.

Οι λόγοι που δεν υλοποιήσαμε αυτή την δομή, αλλά την αφήσαμε για μελλοντική βελτίωση, αποτελούν και τα βασικά μειονεκτήματά της. Πρώτον, θα χρειαζόμασταν πολύ περισσότερη μνήμη για να αναθέσουμε έναν τέτοιον πίνακα σε κάθε βιβλίο εντολών, πολύ περισσότερη από τα 12 GB που διαθέτουμε. Δεύτερον, και πιο σημαντικό, αυτή η λύση χρειάζεται αρκετή έρευνα και είναι αρκετά χρονοβόρα, καθώς χρειάζεται να υπολογιστεί σωστά το κάτω και το πάνω όριο τιμής για κάθε διαφορετική κινητή αξία (το φορτίο που χρησιμοποιήσαμε έχει περίπου 9,000 διαφορετικές κινητές αξίες).

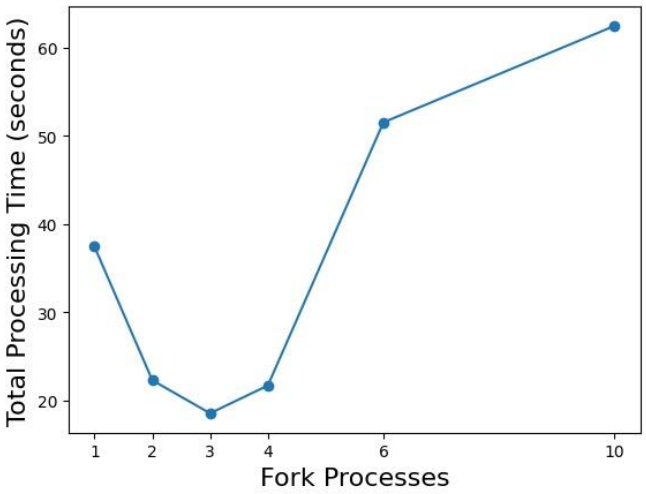
5.1.2 Πολυνηματισμός

Μια ακόμα σημαντική βελτίωση που θα μπορούσαμε να υλοποιήσουμε μελλοντικά είναι η χρήση πολυνηματισμού στο CppTrader. Μιας και τα βιβλία εντολών είναι ανεξάρτητα μεταξύ τους, αυτό σημαίνει ότι θα μπορούσαμε να χρησιμοποιήσουμε πολλαπλά νήματα για τον διαμοιρασμό του βάρους της επεξεργασίας των εντολών, αν εξασφαλίζαμε ότι οι εντολές με προορισμό ένα συγκεκριμένο βιβλίο εντολών υφίστανται επεξεργασία από το ίδιο κάθε φορά νήμα. Αυτό είναι απαραίτητη προϋπόθεση γιατί η επεξεργασία των εντολών ενός βιβλίου εντολών πρέπει υποχρεωτικά να γίνεται σειριακά. Αυτό βέβαια δεν μας απασχολεί ιδιαίτερα στις συνολικές εντολές όλων των βιβλίων εντολών. Στο σχήμα 5.2 παρουσιάζεται ένα προσχέδιο για το πως θα μπορούσε να λειτουργεί μια πολυνηματική υλοποίηση του CppTrader. Προσέξτε ότι οι εντολές με αναγνωριστικό βιβλίου εντολών με ψηφίο που τελειώνει σε 0 υφίστανται επεξεργασία μόνο από το 1ο νήμα, που τελειώνει σε 1 μόνο από το 2ο, κλπ.



Σχήμα 5.2: Ένα απλό προσχέδιο για το πως θα μπορούσε να δουλέψει ο πολυνηματισμός στο σύστημα του CppTrader. Σε αυτό το παράδειγμα γίνεται χρήση 10 νημάτων (threads). Οι εντολές που λαμβάνονται τοποθετούνται σε μια ουρά (queue). Έπειτα, οι εντολές υφίστανται προ-επεξεργασία με FIFO σειρά προτεραιότητας και με βάση το τελευταίο ψηφίο του αναγνωριστικού του βιβλίου εντολών για το οποίο προορίζονται, κατευθύνονται στο κατάλληλο νήμα για επιπλέον επεξεργασία.

Πραγματοποιήσαμε μια πολύ απλοϊκή υλοποίηση της παραπάνω ιδέας χωρίς την χρήση ουράς και χρησιμοποιώντας διεργασίες τύπου fork αντί για νήματα (threads). Επίσης χρησιμοποιήσαμε ένα διαφορετικό φορτίο, από αυτό της ενότητας 3.2, μικρότερου μεγέθους (82,841,542 συνολικές εντολές), και πήραμε τα αποτελέσματα που φαίνονται στο σχήμα 5.3. Από αυτή την εικόνα συμπεραίνουμε ότι μια ιδανική πολυνηματική υλοποίηση πιθανώς θα πρέπει να χρησιμοποιεί τόσα νήματα όσοι και διαθέσιμοι πυρήνες επεξεργαστή, αν και χρειάζεται να γίνουν πιο εκτενείς μετρήσεις πάνω στο συγκεκριμένο ζήτημα.



Σχήμα 5.3: Στον κάθετο άξονα παρουσιάζεται ο συνολικός χρόνος εκτέλεσης της πιο αργής διεργασίας (fork), ενώ στον οριζόντιο ο αριθμός των διεργασιών (fork) που χρησιμοποιήθηκαν.

5.2 Συμπεράσματα

Το πρώτο συμπέρασμα που μπορούμε να εξάγουμε από αυτή την εργασία είναι ότι η **δομή πίνακα (array) αποδίδει καλύτερα** σε σχέση με άλλες πιο σύνθετες δομές, όπως το δέντρο AVL και τον πίνακα κατακερματισμού, στο σύστημα του CppTrader. Αυτό οφείλεται κυρίως στην **καλή χωρική τοπικότητα** που πετυχαίνει στην μνήμη cache, καθώς και στα **ιδιαίτερα χαρακτηριστικά του φορτίου εργασίας** που ευνοούν την χρήση πίνακα. Η χρήση του, όμως, ορισμένες φορές, όπως στην περίπτωση αντικατάστασης του πίνακα κατακερματισμού με πίνακα, δεν έρχεται χωρίς θυσίες, όπως την **επιπλέον δέσμευση μνήμης.**

Το δεύτερο συμπέρασμα είναι ότι, ενώ η αντικατάσταση του πίνακα κατακερματισμού με πίνακα ήταν μια σχετικά απλή στην σύλληψη ιδέα, η αντικατάσταση του δέντρου AVL με έναν ταξινομημένο πίνακα δυναμικού μεγέθους από δομές τύπου struct δεν ήταν καθόλου προφανής. Αυτό οφείλεται στο γεγονός ότι **η επιλογή της δομής για την αποθήκευση των επιπέδων τιμής μιας κινητής αξίας εξαρτάται σημαντικά από τα ιδιαίτερα χαρακτηριστικά του φορτίου εργασίας.** Έτσι, για παράδειγμα, εάν σε μια χρηματιστηριακή μέρα πολλές οριακές εντολές εισάγονται μακριά από την κορυφή του βιβλίου, η χρήση ενός δυαδικού δέντρου πιθανώς να αποδίδει καλύτερα από μια αντίστοιχη λύση πίνακα. Για την αντιμετώπιση αυτού του προβλήματος, τα μεγάλα χρηματιστήρια συνήθως χρησιμοποιούν μια αρκετά βελτιστοποιημένη και πιο σύνθετη δομή πίνακα με σταθερά επίπεδα τιμής (κεφάλαιο 5.1.1).

Τέλος, από αυτή την εργασία μπορούμε να αντλήσουμε το συμπέρασμα ότι για την αξιολόγηση της απόδοσης ενός συστήματος σε καμία περίπτωση δεν μπορούμε να βασιστούμε μόνο στην θεωρητική ανάλυση του (π.χ. χρονική πολυπλοκότητα), αλλά πρέπει να μελετήσουμε και πειραματικά την συμπεριφορά του συστήματός του, π.χ. με διάφορα εργαλεία **profiling ή/και μετρήσεις**, λαμβάνοντας πάντα υπόψιν τα ιδιαίτερα χαρακτηριστικά του φορτίου που χρησιμοποιούμε.

Βιβλιογραφία

[1] Shynkarenka Ivan (chronoxor). *CppTrader*. GitHub, 2021. https://github.com/chronoxor/CppTrader.

[2] Nasdaq, Inc. https://www.nasdaq.com/

[3] Emiliano S. Pagnotta† and Thomas Philippon‡. *Competing on Speed.* †Imperial College Business School, ‡New York University Stern School of Business, National Bureau of Economic Research, and the Centre for Economic Policy Research, 2016.

[4] Xin Wang. *Why Do Stock Exchanges Compete on Speed, and How?,* Nanyang Technological University (NTU) - Division of Banking & Finance, 2021. https://papers.ssrn.com/sol3/papers.cfm? abstract_id=3069529.

[5] Brendan Gregg. *Systems Performance Enterprise and the Cloud*, Prentice Hall, 2014. Chapter 1, Section 1.5 - Performance Is Challenging, pages 4 – 6.

[6] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communications Systems*. Chapter 1 - Methodology., pages 1 – 22. EPFL Press, 2010.

[7] *gperftools (Google Performance Tools)*. GitHub, 2020. https://github.com/gperftools/gperftools.

[8] Sample of historic Nasdaq ITCH files. Nasdaq, 2021. ftp://emi.nasdaq.com/ITCH/.

[9] *std::ios_base::sync_with_stdio*. https://en.cppreference.com/w/cpp/io/ios_base/sync_with_stdio

[10] Charles Cooper. StackExchange, 2017. market microstructure - What is an efficient data structure to model order book? - Quantitative Finance Stack Exchange.

[11] voyager. WaybackMachine, 2014. https://web.archive.org/web/20141222151051/https://dl.dropboxusercontent.com/u/3001534/engine.c.

[12] Conghui He*, Haohuan Fu*, Wayne Luk+, Weijia Li*, and Guangen Yang*. *Exploring the Potential of Reconfigurable Platforms for Order Book Update*. *Tsinghua University, +Imperial College London.

https://www.doc.ic.ac.uk/~wl/papers/17/fpl17ch.pdf.

[13] *Memory Pool*. Wikipedia, 2021. https://en.wikipedia.org/wiki/Memory_pool.

[14] *Explicit Free List*. Wellesley's College Computer Science Department. https://cs.wellesley.edu/~cs240/f18/slides/malloc.pdf.

[15] gcc compiler optimizations. *https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html*

[16] Order Protection Rule. Investopedia, 2021. https://www.investopedia.com/terms/o/order-protection-rule.asp

[17] High Frequency Trading. Investopedia, 2021. https://www.investopedia.com/terms/h/high-frequency-trading.asp