



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

Προανάκτηση σε Blockchain Client με στατική ανάλυση και υποθετική εκτέλεση των Έξυπνων Συμβολαίων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΟΥ

ΠΑΝΑΓΙΩΤΗ ΓΚΟΝΗ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2022



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Προανάκτηση σε Blockchain Client με στατική ανάλυση και υποθετική εκτέλεση των Έξυπνων Συμβολαίων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΠΑΝΑΓΙΩΤΗ ΓΚΟΝΗ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ;;;; Μαρτίου 2022.

(Υπογραφή)

(Υπογραφή)

(Υπογραφή)

.....
Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

.....
Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής

.....
Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2022



(Υπογραφή)

.....

Παναγιώτης Γκόνης

1 Μαρτίου 2022

Copyright © – All rights reserved. Με την επιφύλαξη παντός δικαιώματος.

Παναγιώτης Γκόνης, 2022.

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

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

Περίληψη

TODO

Λέξεις Κλειδιά

TODO

Abstract

TODO

Keywords

TODO

στην οικογένειά μου

Ευχαριστίες

ΤΟΔΟ

Αθήνα, Μάρτιος 2022

Παναγιώτης Γκόνης

Περιεχόμενα

| | |
|--|-----------|
| Περίληψη | 7 |
| Abstract | 9 |
| Ευχαριστίες | 13 |
| 1 Εισαγωγή | 23 |
| 1.1 Κίνητρο και συναφείς προσεγγίσεις | 23 |
| 1.2 Αντικείμενο της διπλωματικής | 24 |
| 1.3 Οργάνωση του κειμένου | 24 |
| 2 Συστήματα αλυσίδων κοινοποιήσεων | 27 |
| 2.1 Σύντομο θεωρητικό υπόβαθρο | 27 |
| 2.2 Έξυπνα συμβόλαια (smart contracts) | 27 |
| 3 Αρχιτεκτονική και υλοποιήσεις λογισμικού για Ethereum | 29 |
| 3.1 Δομή Ethereum | 29 |
| 3.1.1 World state | 29 |
| 3.1.2 Transaction | 30 |
| 3.1.3 Δημιουργία συμβολαίων | 30 |
| 3.1.4 Blockchain | 31 |
| 3.2 Δομές δεδομένων | 31 |
| 3.2.1 Block | 32 |
| 3.2.2 State | 32 |
| 4 Υλοποίηση συστήματος προανάκτησης | 39 |
| 4.1 Σκοπός και περιορισμοί | 39 |
| 4.2 Υψηλού επιπέδου περιγραφή | 40 |
| 5 Αναλυτική περιγραφή του συστήματος | 43 |
| 5.1 Tracing | 43 |
| 5.2 Ανάλυση | 44 |
| 5.3 Προανάκτηση | 52 |
| 5.3.1 Νήμα προανάγνωσης | 52 |
| 5.3.2 Νήματα προανάκτησης | 54 |

| | |
|--|-----------|
| 6 Περιγραφή Benchmarking | 59 |
| 6.1 Εξοπλισμός | 59 |
| 6.1.1 Κύρια κομμάτια του υπολογιστή | 59 |
| 6.2 Μέθοδος δοκιμών | 63 |
| 6.3 Ακρίβεια μετρήσεων | 64 |
| 7 Αποτελέσματα | 65 |
| 7.1 Ποσοστά επιτυχίας predictors | 65 |
| 7.1.1 Επίδραση του Readahead και του πλήθους νημάτων προανάκτησης . . | 65 |
| 7.2 Ταχύτητα εκτέλεσης | 68 |
| 7.2.1 Speedup σε τυπικές συνθήκες | 68 |
| 7.2.2 Επίδραση της κύριας μνήμης | 69 |
| 7.2.3 Επίδραση ταχύτητας CPU (TODO verify) | 70 |
| 7.2.4 Ταχύτητα εκτέλεσης Ποσοστό χρόνου στη βάση δεδομένων | 70 |
| 7.2.5 Δοκιμή σε άλλες χρονικές περιόδους | 71 |
| 8 Επίλογος | 73 |
| Βιβλιογραφία | 77 |

Κατάλογος Σχημάτων

| | | |
|------|--|----|
| 1.1 | Υψηλού επιπέδου όψη του συστήματος, όπως θα αναλυθεί στο κεφάλαιο 4. . . | 24 |
| 3.1 | Αναπαράσταση δύο transaction και των message calls που προκαλούν. . . . | 31 |
| 3.2 | Ενδεικτική διάρκεια των σταδίων του <i>Erigon</i> | 34 |
| 3.3 | Ενδεικτική διάρκεια των σταδίων του <i>Erigon</i> , αγνοώντας τα στάδια μεταφόρτωσης. | 35 |
| 4.1 | Υψηλού επιπέδου όψη του συστήματος. | 40 |
| 5.1 | | 44 |
| 5.2 | | 45 |
| 5.3 | | 45 |
| 5.4 | | 46 |
| 5.5 | Οι επιλεγμένες εντολές έχουν ένα αστεράκι * στα αριστερά των διευθύνσεων, ομοίως και τα επιλεγμένα block αριστερά του ονόματός τους. | 47 |
| 5.6 | | 48 |
| 5.7 | | 48 |
| 5.8 | Με την αρίθμηση των επιλεγμένων εντολών και υπολογισμό των εκφράσεών τους ως προς την αρίθμηση, δεν χρειάζονται ποια οι αρχικές. | 49 |
| 5.9 | Σχηματική αναπαράσταση του υποσυστήματος προανάκτησης. Τα μπλε κουτιά αντιπροσωπεύουν ήδη υπάρχουσες δομές, ενώ τα μαύρα είναι αυτά που προστέθηκαν. | 53 |
| 5.10 | | 56 |
| 5.11 | | 56 |
| 5.12 | | 56 |
| 5.13 | | 56 |
| 6.1 | | 60 |
| 6.2 | | 60 |
| 6.3 | Κατανομή χρόνου απόκρισης των δύο δίσκων. Ο nvme έχει περίπου το μισό από τον sata. | 62 |
| 6.4 | Σε αντίθεση με του nvme, η κατανομή του sata έχει «heavy tail» | 63 |
| 6.5 | Κατανομή της ταχύτητας εκτέλεσης των 10 ίδιων δοκιμών για κάθε δίσκο, με 4 νήματα προανάκτησης. | 64 |
| 7.1 | | 66 |
| 7.2 | | 66 |
| 7.3 | | 67 |
| 7.4 | | 67 |

| | | |
|-----|---|----|
| 7.5 | . | 68 |
| 7.6 | . | 69 |
| 7.7 | . | 70 |
| 7.8 | . | 71 |
| 7.9 | . | 72 |

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Από την εισαγωγή του Bitcoin το 2008 με την εμβληματική δημοσίευση "Bitcoin: A Peer-to-Peer Electronic Cash System" [1], το ενδιαφέρον για συστήματα κρυπτονομισμάτων έχει αυξηθεί με ταχύτατο ρυθμό. Αναμφισβήτητα από το πιο δημοφιλή τέτοια έργα είναι το Ethereum blockchain [2] με βασικό χαρακτηριστικό του τη δυνατότητα εκτέλεσης συντομων προγραμμάτων γνωστών ως Έξυπνα Συμβόλαια (Smart Contracts), προσφέροντας τις ίδιες εγγυήσεις ασφαλείας με της απλές συναλλαγές μεταφοράς αξίας (value-transfer transactions). Αυτό ακριβώς το χαρακτηριστικό ήταν ο κινητήριος παράγοντας πίσω από την τεράστια αύξηση της δημοτικότητάς του, όπως και του όγκου των συναλλαγών [3]. Ωστόσο, όπως αποδεικνύεται στα χρόνια που προηγήθηκαν της παρούσας εργασίας, εξακολουθεί να αγωνίζεται για να καλύψει τη ζήτηση για υψηλότερη απόδοση (throughput), με αποτέλεσμα οι προμήθειες συναλλαγών να είναι πολλή υψηλές [4].

Η απόδοση του συστήματος είναι περιορισμένη ώστε να επιτρέπεται σε όλους τους συμμετέχοντες να παραμένουν συγχρονισμένοι με το δίκτυο. Ενώ οι υπάρχοντες κόμβοι δεν έχουν κανένα πρόβλημα με τις τρέχουσες απαιτήσεις, νέοι (πλήρεις) κόμβοι που εισάγονται στο δίκτυο πρέπει πρώτα να συγχρονιστούν μέχρι το πιο πρόσφατο block, εκτελώντας τις συναλλαγές ολόκληρου του blockchain από το πρώτο block, μια χρονοβόρα διαδικασία που επιβραδύνει την έρευνα και ανάπτυξη πάνω στο blockchain του.

1.1 Κίνητρο και συναφείς προσεγγίσεις

Το επίσημο λογισμικό (client) για το δίκτυο του Ethereum, το **Go-ethereum** ή **geth** [5], ακολουθεί πιστά την επίσημη περιγραφή του [2] δίνοντας έμφαση σε ορθότητα και feature richness [?] με αποτέλεσμα να έχει μεγάλες απαιτήσεις σε χωρητικότητα και ταχύτητα αποθηκευτικού μέσου (storage capacity και I/O). Για παράδειγμα, είναι πια αδύνατη η εκτέλεσή του με σκληρό δίσκο [6] λόγω της χαμηλής ταχύτητάς τους σε τυχαίες προσβάσεις (random I/O). Διάφορες εργασίες [7, ?] έχουν προσπαθήσει και επιτύχει σε μεγάλο βαθμό να βελτιώσουν την απόδοσή του, χρησιμοποιώντας διαφορετικές δομές δεδομένων για πιο αποδοτική αποθήκευση των δεδομένων του. Επίσης, έχει εξεταστεί η δυνατότητα παραλληλοποίησης της εκτέλεσής του τόσο με μεθόδους αισιόδοξης ταυτόχρονης εκτέλεσης (optimistic concurrent execution) [8, 9, 10] όσο και με ντετερμινιστικό τρόπο, υποθέτοντας επιπλέον γνώση των συγκρούσεων (conflicts) μεταξύ των transaction [11, 9, 10]. Στην τελευταία περίπτωση προτείνονται κάποιες επεκτάσεις και αλλαγές στο πρωτόκολλο του Ethereum για να υπ-

οστηρίζονται αυτές οι μέθοδοι. Άλλες προσπάθειες επικεντρώνονται στην επιτάχυνση από πλευράς ? χρόνου επεξεργαστή ? (cpu-time) με τη συγγραφή client σε άλλη γλώσσα από την Go, όπως σε Rust [?, ? ΑΛΛΕΣ ?] ή C++ [?, ? ΑΛΛΕΣ ?]. Τέλος, έχει δοθεί και λίγη έμφαση στην εφαρμογή προανάκτησης (prefetching) με προ-εκτέλεση (pre-execution) των Smart Contracts [7, 12], χωρίς όμως ξεχωριστή μελέτη της συγκεκριμένης μεθόδου.

Αξίζει να αναφερθεί και η σταδιακή εισαγωγή ενός συνόλου βελτισεων στο Ethereum [13], προηγουμένως γνωστό και ως Ethereum 2.0, το οποίο επιδιώκει μεγάλες βελτιώσεις στην ταχύτητα του συνολικού δικτύου, κάνοντας σημαντικές αλλαγές στην αρχιτεκτονική του, οι οποίες δεν είναι συμβατές με το τωρινό. Ακόμα και αυτή η έκδοση όμως, θα συνεχίσει να χρησιμοποιεί την ίδια βάση για την εκτέλεση των συναλλαγών (execution layer), που σημαίνει ότι οι προαναφερθείσες εργασίες και η παρούσα διπλωματική συνεχίζουν να έχουν ισχύ σε αυτό το τμήμα του.

1.2 Αντικείμενο της διπλωματικής

Η διπλωματική αυτή εκτελείται με την τελευταία αυτή μέθοδο, της προανάκτησης, βασισμένη στον Erigon client [?], υλοποιώντας ένα νέο σύστημα ανάλυσης του κώδικα των contracts και σύνθεση απλούστερου με αποκλειστικό σκοπό την εκτέλεσή του για προανάκτηση. Η μέθοδος αυτή δεν απαιτεί αλλαγές στην επικοινωνία μεταξύ των συμμετεχόντων, ή στον ορισμό (specification) του Ethereum [2] του και επομένως μπορεί να υποστηρίζεται από καθένα client ξεχωριστά.

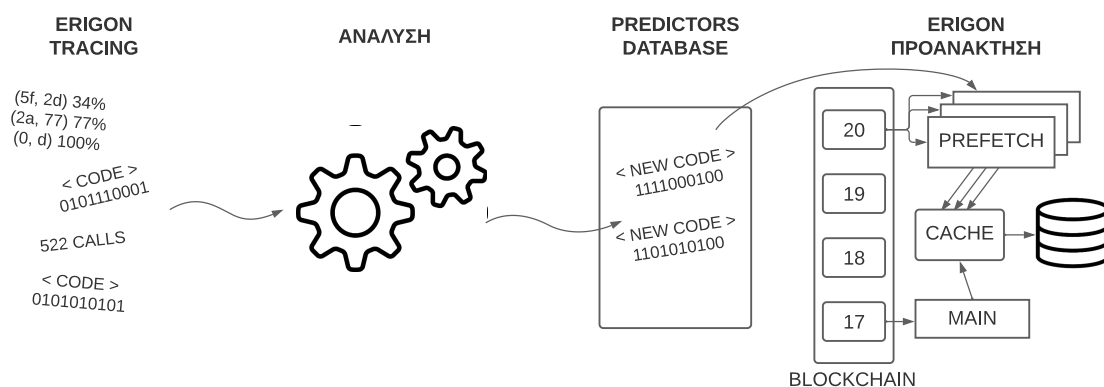


Figure 1.1: Υψηλού επιπέδου όψη του συστήματος, όπως θα αναλυθεί στο κεφάλαιο 4.

1.3 Οργάνωση του κειμένου

Η διπλωματική εργασία αποτελείται από 8 Κεφάλαια.

- Στο Κεφάλαιο 1 γίνεται η εισαγωγή του θέματος που εξετάζεται.
- Στο Κεφάλαιο 2 θα γίνει μια γενική επεξήγηση των συστημάτων αλυσίδων κοινοποιήσεων (blockchain).

- Στο Κεφάλαιο 3 αναλύεται η αρχιτεκτονική και υλοποιήσεις λογισμικού για Ethereum, εξετάζοντας τους 2 πιο δημοφιλείς clients [14], τον Go-ethereum και τον Erigon. Δίνεται περισσότερη έμφαση στο επίπεδο αποθήκευσης και των δομών δεδομένων τους.
- Στο Κεφάλαιο 4 περιγράφεται από ένα υψηλό επίπεδο η αρχιτεκτονική του συστήματος προανάκτησης. Αναφέρονται επίσης οι σχεδιαστικές επιλογές που έγιναν και περιορισμοί που τέθηκαν για τη συμπεριφορά του.
- Στο Κεφάλαιο 5 παρουσιάζεται αναλυτικά η υλοποίηση του συστήματος. Αναλύονται χωριστά τα κομμάτια από τα οποία αποτελείται, όπως περιγράφεται στο κεφάλαιο 4.
- Στο Κεφάλαιο 6 περιγράφεται η διαδικασία με την οποία διεκπεραιώθηκαν οι δοκιμές και μετρήσεις (benchmarking). Αναφέρονται τα χαρακτηριστικά του υλισμικού (hardware) και οι παραμετροποιήσεις (configuration) του συστήματος. Επίσης περιγράφεται ο τρόπος με τον οποίο πάρθηκαν οι μετρήσεις, καθώς και μια απλή εκτίμηση της ακρίβειάς τους.
- Στο Κεφάλαιο 7 παρουσιάζονται τα αποτελέσματα των μετρήσεων και γίνεται μια σύντομη αξιολόγηση του συστήματος.
- Στο Κεφάλαιο 8 συνοψίζονται τα αποτελέσματα και παραθέτονται κάποια βασικά συμπεράσματα. Γίνεται επίσης αναφορά και στις δυνατότητες που δημιουργούνται για επεκτάσεις και μελλοντική έρευνα πάνω στο θέμα.

Κεφάλαιο 2

Συστήματα αλυσίδων κοινοποιήσεων

Το κεφάλαιο αυτό περιέχει τις βασικές πληροφορίες που χρειάζονται στην υπόλοιπη εργασία. Το επόμενο κεφάλαιο με την ανάλυση σε βάθος να μπορεί να προσπεραστεί.

2.1 Σύντομο θεωρητικό υπόβαθρο

Οι αλυσίδες κοινοποιήσεων (blockchain), που θα εξεταστούν στα πλαίσια της εργασίας, υλοποιούν ψηφιακά λογιστικά βιβλία (ledgers). Αποτελούνται από μπλοκ εγγραφών (block) τα οποία συνδέονται μεταξύ τους (με κρυπτογραφικές μεθόδους) σε μορφή απλά συνδεδεμένης λίστας. Τα block περιέχουν συναλλαγές (transaction) και κάθε νέο block τοποθετείται αυστηρά μετά το πιο πρόσφατο, χωρίς να υπάρχει η δυνατότητα αλλαγή κάποιου προηγούμενου ή των περιεχομένων του (append only). Σε περίπτωση ύπαρξης δύο ή περισσότερων block που συνδέονται στο ίδιο προηγούμενο block, ένας κατανεμημένος αλγόριθμος συμφωνίας (consensus) επιλέγει ποιο απ' αυτά θα κρατηθεί.

Δύο απ' τις κατηγορίες στις οποίες διακρίνονται οι ledger που υλοποιούν είναι:

- Οι βασισμένοι σε αξόδευτες εξόδους συναλλαγών (Unspent Transaction Outputs, UTXO), όπως το Bitcoin [1]. Σε αυτή την κατηγορία οι συναλλαγές έχουν εισόδους και εξόδους με τις μελλοντικές συναλλαγές χρησιμοποιούν εξόδους παλαιότερων ως εισόδο, υπό την προϋπόθεση ότι κάθε έξοδος χρησιμοποιείται το πολύ μία φορά και το άθροισμα των ποσών των εισόδων ισούται με αυτό των εξόδων.
- Οι βασισμένοι σε λογαριασμούς (accounts), όπως το Ethereum [2]. Σε αυτή των κατηγορία διατηρείται για κάθε λογαριασμό το μη-αρνητικό υπόλοιπό του. Οι λογαριασμοί προσδιορίζονται από τη διεύθυνσή τους (address).

2.2 Έξυπνα συμβόλαια (smart contracts)

Πέραν της εκτέλεσης transaction για τη μεταφορά αξίας, ορισμένα blockchain κρυπτονομισμάτων παρέχουν τη δυνατότητα εκτέλεσης έξυπνων συμβολαίων. Στα πλαίσια της παρούσας εργασίας θα ασχοληθούμε αποκλειστικά με το Ethereum που υποστηρίζει αυτή τη δυνατότητα.

Τα έξυπνα συμβόλαια δημιουργούνται από τους χρήστες εξωτερικά του blockchain και εγκαθίστανται σε αυτό με ένα transaction δημιουργίας συμβολαίου (contract creation). Έτσι, ο κώδικας των συμβολαίων δεν βρίσκεται μέσα στον κώδικα του λογισμικού που χρησιμοποιούν οι συμμετάσχοντες στο δίκτυο. Μετά τη δημιουργία του, ένα συμβόλαιο μπορεί να

χρησιμοποιηθεί κάνοντας άλλα transaction τα οποία προορίζονται στη διεύθυνσή του. Επιπλέον, ένα συμβόλαιο έχει τη δυνατότητα να χρησιμοποιεί και άλλα υπάρχοντα συμβόλαια, κάνοντας κλήσεις (message calls) σε αυτά. Η διαδικασία αυτή απεικονίζεται στο σχήμα 3.1 και περιγράφεται στην παράγραφο 3.1.3 του επόμενου κεφαλαίου.

Η αρχιτεκτονική στην οποία βασίζονται είναι το EVM (Ethereum Virtual Machine), η οποία είναι turing complete και βασίζεται σε μοντέλο μηχανής στοίβας (stack machine). Η αρχιτεκτονική αυτή καθορίζει για κάθε συμβόλαιο, αποκλειστικό μη-πτητικό αποθηκευτικό χώρο, πρακτικά απεριόριστο ($2^{256} \cdot 32$ Byte).

Το σύνολο των λογαριασμών, των συμβολαίων και των αποθηκευτικών τους χώρων ονομάζεται *World State* και θα το αναφέρεται απλά ως state.

Κεφάλαιο 3

Αρχιτεκτονική και υλοποιήσεις λογισμικού για Ethereum

3.1 Δομή Ethereum

3.1.1 World state

Μιας και το Ethereum χρησιμοποιεί ledger βασισμένο σε λογαριασμούς (αντί για UTXO όπως το Bitcoin), χρειάζεται να διατηρεί μια συνεπή (consistent) και αντιγραφόμενη (replicated) δομή με την κατάσταση (World State) των λογαριασμών, συμβολαίων και του αποθηκευτικού χώρου των συμβολαίων (contract storage ή απλά storage). Αυτή η δομή πρέπει να είναι και επαληθεύσιμη, να μπορεί δηλαδή να δημιουργηθεί κρυπτογραφική απόδειξη των περιεχομένων της, η οποία απόδειξη να βρίσκεται μέσα στις επικεφαλίδες (headers) των block. Για την επίτευξη αυτού χρησιμοποιείται η δομή modified Merkle Patricia Trie.

modified Merkle Patricia Trie (mMPT ή MPT)

Η δομή αυτή ορίζεται αναλυτικά στο Yellowpaper του Ethereum [2]. Είναι για δενδρική δομή, η οποία λειτουργεί ως Key-Value store. Οι τιμές (Value) αποθηκεύονται στους κόμβους διακλάδωσης ή στα φύλλα του δέντρου, ενώ το κλειδί (Key) είναι το μονοπάτι από τη ρίζα στον κόμβο όπου αποθηκεύεται η αντίστοιχη τιμή. Υπάρχουν τρία είδη κόμβων:

- Κόμβοι διακλάδωσης βάζουν 1 δεκαδικό ψηφίο στο κλειδί.
- Κόμβοι επέκτασης βάζουν περισσότερα ψηφία.
- Κόμβοι φύλλα κρατάνε τιμές.

Επίσης επιτρέπεται και οι κόμβοι διακλάδωσης να έχουν τιμή. Οι ακμές του αποτυπώνονται με τον κατακερματισμό (hash) των κόμβων, όχι απλά με δείκτες - διευθύνσεις μνήμης. Επομένως, το hash της ρίζας (root hash) εξαρτάται από hash όλων των κόμβων και των τιμών του, το οποίο επιτρέπει την (πιθανολογική) επιβεβαίωση των περιεχομένων ολόκληρης της δομής απλά συγκρίνοντας το root hash. Το root hash τοποθετείται στην επικεφαλίδα του αντίστοιχου block. Υπάρχουν και άλλα πλεονεκτήματα και δυνατότητες που προσφέρει η δομή αυτή, τα οποία είναι ? out of scope ?.

Λογαριασμού (Accounts)

Το Yellowpaper περιγράφει το World State ως αντιστοίχιση διευθύνσεων (addresses) σε καταστάσεις λογαριασμών (account states, εδώ απλά λογαριασμοί ή accounts). Οι λογα-

ρισμοί (accounts) είναι τετράδες: (nonce, balance, storageRoot, codeHash). Η αντιστοίχιση γίνεται με το MPT, με το κλειδί να είναι η διεύθυνση του account (160-bit) και η τιμή είναι ο λογαριασμός (η τετράδα).

Συμβόλαια (Contracts)

Το πεδίο codeHash καθορίζει αν ο λογαριασμός πρόκειται για έξυπνο συμβόλαιο (contract). Στην περίπτωση που είναι το hash της κενής συμβολοσειράς - `HASH(())`, ο λογαριασμός είναι εξωτερικός (Externally Operated Account - EOA) και ανήκει πιθανώς σε κάποιον χρήστη. Σε contract, το codeHash είναι το hash της δυαδικής κωδικοποίησης (encoding) του εκτελέσιμου κώδικά του (runtime bytecode). Το κάθε contract έχει και μη-πιητικό (non-volatile) αποθηκευτικό χώρο (storage), με τη μορφή ζευγών:

slot index (32 Bytes) → content (32 Bytes)

που αποθηκεύονται ως ζεύγη key-value¹ σε MPT, το root hash του οποίου αποτελεί το πεδίο storageRoot.

3.1.2 Transaction

Οι συναλλαγές (transaction) παράγονται και υπογράφονται ψηφιακά από τους χρήστες του συστήματος, εξωτερικά απ' αυτό, και κοινοποιούνται (broadcast) στο δίκτυο του Ethereum, με σκοπό να συμπεριληφθούν τελικά σε κάποιο block από τους miners. Επομένως, transaction μπορούν να προέρχονται μόνο από EOA.

Σε αντίθεση, οι κλήσεις-μηνύματα (message calls) μπορούν να παραχθούν και εσωτερικά του συστήματος κατά την εκτέλεση ενός contract. Ένα message call μπορεί να παράξει περισσότερα ακόμα calls, όχι όμως transaction. Σε κάθε transaction αντιστοιχεί ένα αρχικό message call το οποίο γίνεται προς το contract η διεύθυνση του οποίου αναγράφεται στο πεδίο προορισμού (to) του transaction (εφόσον είναι contract). Αυτό αποτυπώνεται και στο σχήμα 3.1.

3.1.3 Δημιουργία συμβολαίων

Αν το πεδίο to λείπει από ένα transaction, τότε πρόκειται για (πιθανή) δημιουργία νέου contract. Το πεδίο data τότε ερμεινεύεται ως κώδικας ο οποίος εκτελείτε άμεσα (initialization bytecode) και δεν είναι ο κώδικας του νέου contract. Αντιθέτως, αφότου εκτελεστεί, πρόκειται να επιστρέψει τον κώδικα του νέου contract (runtime bytecode). (Αυτό το σχήμα επιτρέπει την αποδοτική υλοποίηση της συνάρτησης αρχικοποίησης (constructor) του contract, μιας και χρησιμοποιείται μόνο μία φορά και θα ήταν σπατάλη αν ο κώδικάς της έμενε στο τελικό contract.) Οι αναλύσεις που θα περιγραφούν στο υπόλοιπο μέρος της εργασίας αναφέρονται μόνο στο νέο αυτό κώδικα. Είναι επίσης δυνατό να δημιουργηθεί ένα contract, μέσω των εντολών CREATE, CREATE2 κατά τη διάρκεια εκτέλεσης άλλου contract, με το ίδιο τελικό αποτέλεσμα.

¹Για να αποφεύγονται επιθέσεις DOS στη δομή του MPT, δεν αποθηκεύεται (ως key) το slot, αλλά το hash του [15].

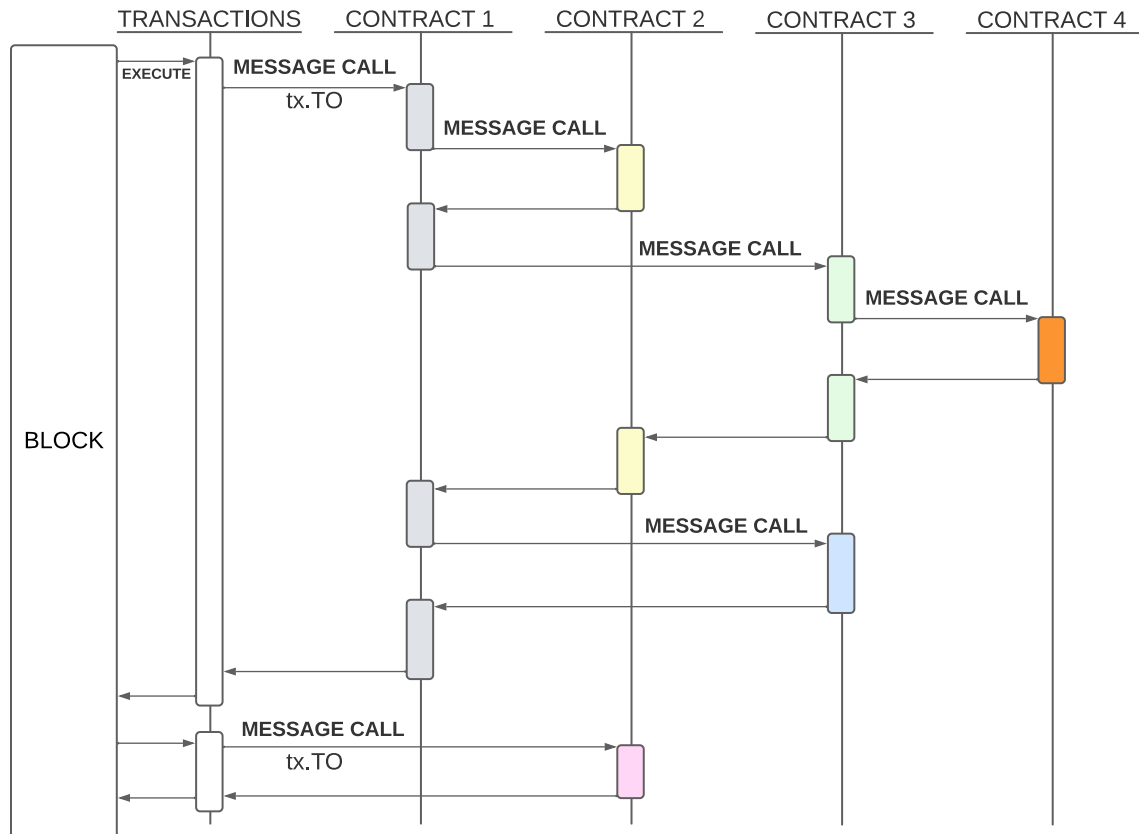


Figure 3.1: Αναπαράσταση δύο transaction και των message calls που προκαλούν.

3.1.4 Blockchain

Τα block αποτελούνται από δύο τμήματα: επικεφαλείδα (header) και σώμα (body).

Τα transaction τοποθετούνται στο block body σε μία απλή λίστα, αλλά ταυτόχρονα σχηματίζεται ένα MPT που τα περιέχει, το root hash του οποίου τοποθετείται στο header του block.

Τα blocks παράγονται από τους miners με μη ντετερμινιστικό τρόπο. Αυτό έχει σαν αποτέλεσμα να παράγονται σε ορισμένες περιπτώσεις ταυτόχρονα blocks τα οποία προφανώς δεν είναι συμβατά και δεν μπορούν να μπουν και τα δύο στο blockchain. Ο αλγόριθμος consensus επιλέγει πιο θα κρατηθεί, το/α άλλο/α όμως δεν απορρίπτονται τελείως αλλά το header τους περνάει (προαιρετικά) σε μία λίστα omers (ή uncles) στο body επόμενου block που θα παραχθεί. Το hash αυτής της λίστας τοποθετείται στο header του block. (τα transaction αυτών των block φυσικά δεν εκτελούνται)

3.2 Δομές δεδομένων

Σε αυτή την ενότητα θα περιγραφούν οι βασικές δομές δεδομένων που χρησιμοποιούν δύο δημοφιλείς clients, ο Go-ethereum και ο Erigon, και οι διαφορές μεταξύ τους. Στις δομές του Erigon που περιγράφονται εδώ βασίζεται και η υλοποίηση που περιγράφεται στο επόμενο κεφάλαιο.

3.2.1 Block

Υλοποίηση

Η δομή των block που περιγράφηκε προηγουμένως είναι κοινή και στους 2 client:

- Header:

Geth@12f0ff4/core/types/block.go:68-88

- Body:

Geth@12f0ff4/core/types/block.go:153-156

3.2.2 State

Go-ethereum

Ο client go-ethereum χρησιμοποιεί εσωτερικά τη δομή MPT όπως περιγράφηκε. Οι ακμές στο δέντρο υλοποιούνται εντός μνήμης (in-memory) με διευθύνσεις μνήμης, αλλά για να αποθηκευτούν χρειάζεται εναλλακτική μέθοδος. Αυτό γίνεται, χρησιμοποιώντας τα hash των κόμβων του δέντρου. Πιο συγκεκριμένα, χρησιμοποιώντας κάποια κοινή βάση δεδομένων τύπου KV-store, όπως τη LevelDB, η αποθήκευση επιτυγχάνεται θέτοντας ως τιμή τον σειριοποιημένο-κατά-RLP κόμβο και ως κλειδί το hash της τιμής. Στους ενδιαμέσους κόμβους (όχι φύλλα) η αναφορά στους κόμβους του επόμενου επιπέδου γίνεται με το hash τους. Αξίζει να σημειωθεί ότι κάθε hash² αποθηκεύεται δύο φορές συνολικά: - μία φορά ως κλειδί του αντίστοιχου κόμβου και - μία φορά μέσα στη σειριοποιημένη τιμή ενός ενδιαμέσου κόμβου. Επίσης εδώ γίνεται εμφανές πως μιας και το ίδιο το MPT αποτελεί ουσιαστικά KV-store, δημιουργείται συνολικά ένα σχήμα KV-store-in-KV-store για την αποθήκευση των accounts. Εντελώς ανάλογη είναι και η δομή για τα contract storage, με το root hash του storage-MPT του κάθε Contract να αποθηκεύεται στο αντίστοιχο account που περιγράφηκε προηγουμένως.

Υλοποίηση

Η δομή *stateDB* περιέχει τα *stateObjects*. Τα *stateObject* αντιστοιχούν 1-1 σε *Account* τα οποία περιγράφουν:

Geth@12f0ff4/core/state/statedb.go:59-65,77-80

Δημιουργούνται διαβάζοντας το αντίστοιχο *Account* από τη *stateDB*:

Geth@12f0ff4/core/state/statedb.go:489-493,503,513,517-518,535-537,541

Εδώ εφαρμόζεται μία in-memory read-write cache που ονομάζεται (στον κώδικα) *snapshot*³. Έχει πολλαπλά επίπεδα (layers) τα οποία περιέχουν *Accounts* και *Storage* που έχουν ήδη διαβαστεί ή τροποποιηθεί κατά την εκτέλεση. Η αναζήτηση γίνεται περώντας από τα

²εκτός της ρίζας

³Αν και in-memory, μπορεί να αποθηκευτεί κατά το κλείσιμο του client στον δίσκο, ώστε να διαβαστεί κατά την εκκίνηση μελλοντικής εκτέλεσης και να μην χρειαστεί να ξανα-δημιουργηθεί.

επίπεδα με τη σειρά, μέχρι να βρεθεί το κλειδί (διεύθυνση *Account* ή *Storage*) που αναζητείται. Αν δεν βρεθεί σε κανένα επίπεδο γίνεται ανάκτηση από το *Trie*. Λόγο του μεταβλητού και εν δυνάμει μεγάλου πλήθους των layers που πρέπει να εξεταστούν για ένα κλειδί που δεν υπάρχει στο *snapshot*, διατηρείται ένα φίλτρο Bloom, το οποίο αποφαινεται για την πιθανή⁴ ύπαρξη ή όχι του κλειδιού σε οποιοδήποτε layer, πριν ξεκινήσει η αναζήτηση. Έτσι, αρκετές άσκοπες αναζητήσεις παρακάμπνυνται.

Υλοποίηση αναζήτησης (παράδειγμα για *Account*):

```
Geth@12f0ff4/core/state/snapshot/difflayer.go:270-273,291-295,311-312,318,344-346,349
```

Αν αποτύχει:

```
Geth@12f0ff4/core/rawdb/accessors_snapshot.go:75-79
```

Σε περίπτωση που το *snapshot* δεν υπάρχει (πχ δεν έχει δημιουργηθεί ακόμα) ή δεν περιέχει το κλειδί που ζητείται, η ανάκτηση γίνεται από το αντίστοιχο *state Trie*:

```
Geth@12f0ff4/trie/trie.go:116-117,124,141-142,148-149
```

Η αναζήτηση αυτή πιθανών να προκαλέσει μία ή και παραπάνω αναγνώσεις απ'το δίσκο, με τη διαδικασία αναζήτησης που περιγράψαμε προηγουμένως. Η υλοποίηση φαίνεται παρακάτω:

```
Geth@12f0ff4/trie/trie.go:499-501
```

```
Geth@12f0ff4/trie/database.go:372,393-394
```

```
Geth@12f0ff4/ethdb/leveldb/leveldb.go:22,33,61,63,187-194
```

Τα *stateObject* ανακτούν δεδομένα *Storage* με τον ίδιο τρόπο:

```
Geth@12f0ff4/core/state/state_object.go:180,191,195,224,237,238,250,
```

Οι ανακτήσεις δεδομένων *Storage* προκαλούνται από εντολές SLOAD/SSTORE κατά την εκτέλεση κάποιου *Smart Contract*:

```
Geth@12f0ff4/core/vm/instructions.go:516-531
```

Επιπλέον ανακτήσεις, αυτή τη φορά για *Accounts*, γίνονται κατά την εκτέλεση των transaction καθώς και την ολοκλήρωση της επεξεργασίας ενός *Block*:

```
Geth@12f0ff4/core/state_processor.go:59,76,88-93,
```

```
Geth@12f0ff4/consensus/ethash/consensus.go:587-593,
```

```
Geth@12f0ff4/core/state/statedb.go:823-879,
```

```
Geth@12f0ff4/core/state/statedb.go:450-451,459,463,
```

Τέλος, η εγγραφή όλων των τροποποιήσεων στη βάση στο δίσκο γίνεται αργότερα, με κλήση στο *Trie*:

```
Geth@12f0ff4/trie/trie.go:517,527,547
```

⁴Σε περίπτωση που το κλειδί υπάρχει, η απάντηση είναι βέβαιη, ενώ σε περίπτωση που δεν υπάρχει, μπορεί να δοθεί ψευδώς θετική απάντηση ότι υπάρχει. Αυτή η συμπεριφορά του φίλτρου δεν επιρρεάζει την ορθότητα των αποτελεσμάτων.

Erigon

Ο client erigon ξεκίνησε από τροποποιήσεις του κώδικα του go-ethereum (fork) με σκοπό την επιτάχυνσή του [?], και επομένως οι δύο client έχουν πολλά κοινά κομμάτια. Κάποιες από τις διαφορές τους που αφορούν την εργασία αναλύονται παρακάτω. Μία βασική διαφορά του είναι ο χωρισμός της συνολικής δουλειάς του σε στάδια (stages), όπως το στάδιο μεταφόρτωσης (downloading) των header και body των block, το στάδιο εκτέλεσης (execution stage) και άλλα. Στο σχήμα 3.2 φαίνεται ενδεικτικά το ποσοστό του συνολικού χρόνου που διαρκεί το κάθε στάδιο, ενώ στο σχήμα 3.3 έχουν αφαιρεθεί τα στάδια downloading, που εξαρτώνται απ' το δίκτυο.

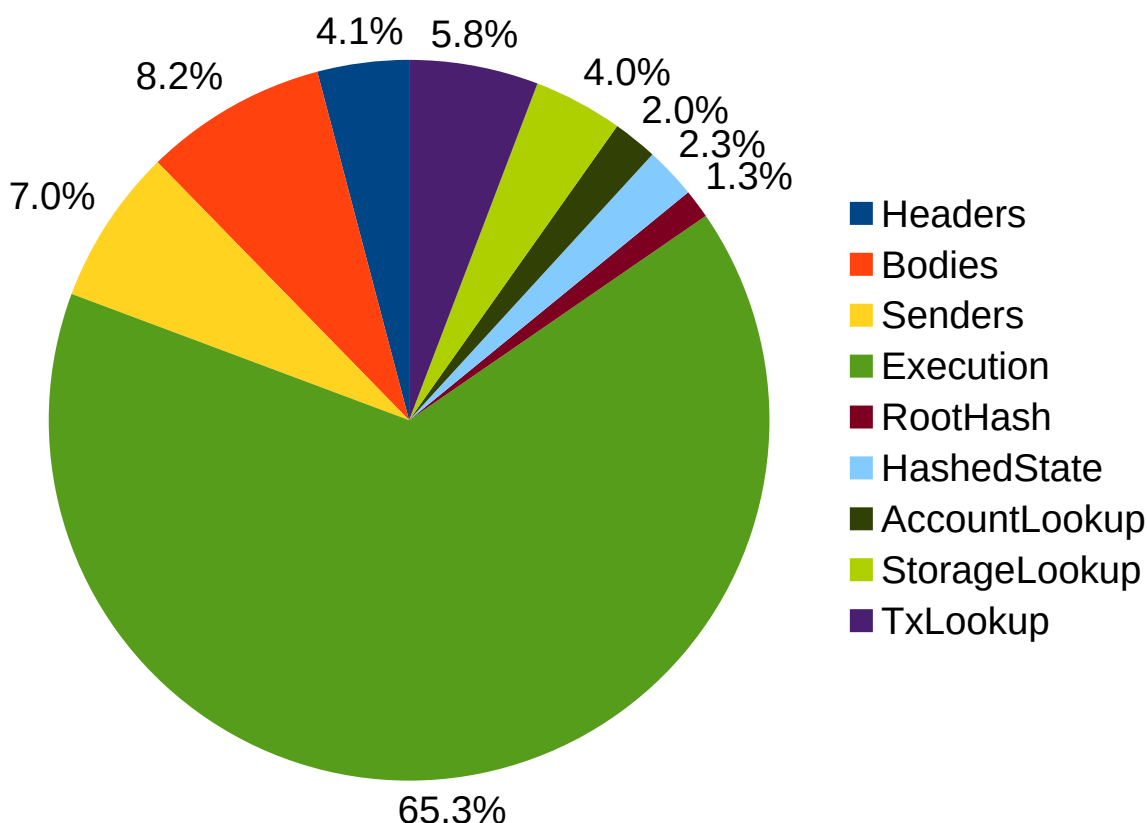


Figure 3.2: Ενδεικτική διάρκεια των σταδίων του Erigon.

Η εργασία αυτή ασχολείται αποκλειστικά με το στάδιο execution, μιας και διαρκεί το μεγαλύτερο χρόνο.

Άλλη μία διαφορά είναι ο τρόπος αποθήκευσης των δεδομένων. Ο Erigon δεν χρησιμοποιεί δομή MPT, αντιθέτως αποθηκεύει τα accounts και τα contract storage απ' ευθείας στη βάση δεδομένων, χωρίς τον υπολογισμό των αντίστοιχων hash, σε "πίνακα" (bucket) που ονομάζει "Plain State". Για την επιβεβαίωση των root hash που βρίσκονται μέσα στα block, προστίθεται ένα επιπλέον στάδιο κατά το οποίο υπολογίζονται τα hashes (και πιο σημαντικά το root hash), μαζικά, σαν να ήταν σε μορφή MPT.

Επίσης, χρησιμοποιεί μια write-cache, βασιζόμενη σε β-δέντρα, η οποία κρατάει στη μνήμη (write-back) όλες τις εγγραφές που πρόκειται να γίνουν στη βάση, μέχρι το μέγεθός της να υπερβεί κάποιο προδιαγεγραμμένο όριο. Όταν το υπερβεί, η εκτέλεση έρχεται σε παύση, οι εγγραφές καταχωρούνται στη βάση και η cache αδειάζει. Αυτό, από πειραματικές

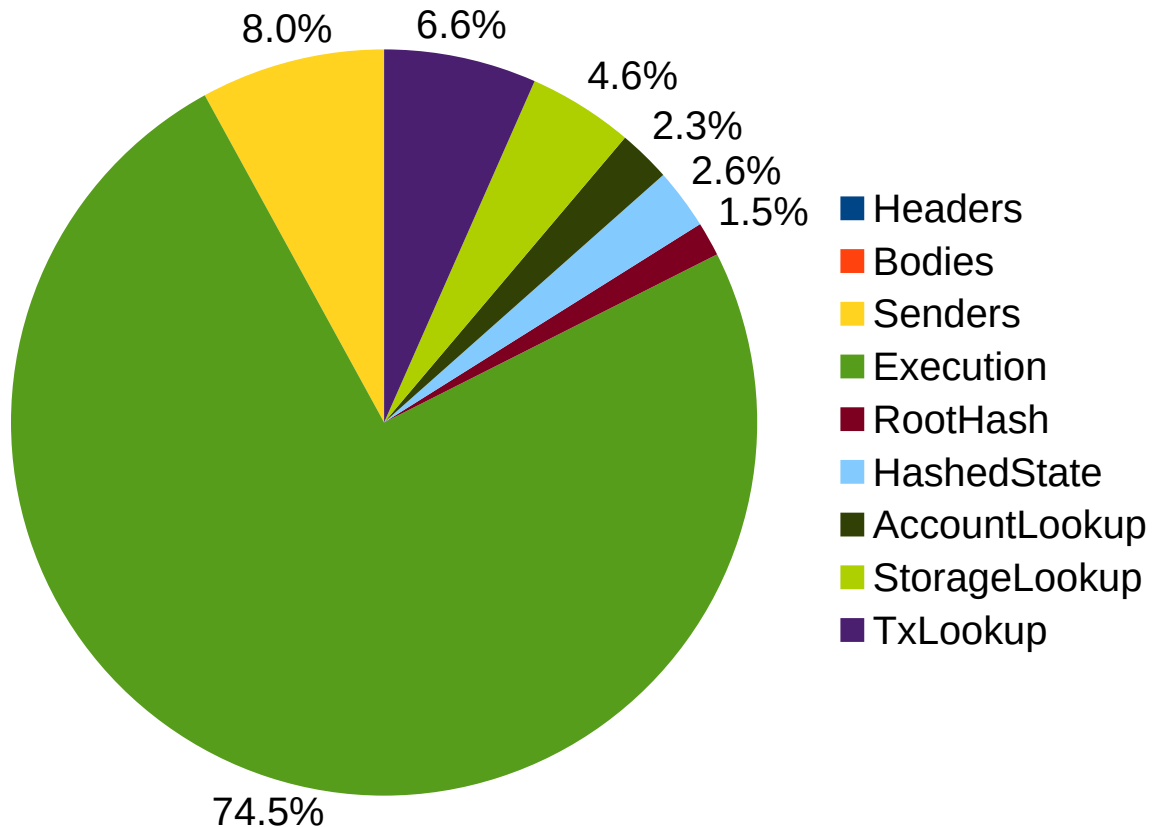


Figure 3.3: Ενδεικτική διάρκεια των σταδίων του Erigon, αγνοώντας τα στάδια μεταφόρτωσης.

δοκιμές φαίνεται συμβαίνει κάθε λίγες χιλιάδες block, αλλά κυμαίνεται αρκετά με βάση το περιεχόμενό τους. Από το τρόπο λειτουργίας των β-δέντρων, η διάσχιση στην cache και καταχώρηση στη βάση γίνεται με τα κλειδιά να είναι ταξινομημένα, το οποίο αυξάνει την ταχύτητα της διαδικασίας [16].

Τέλος η βάση δεδομένων που χρησιμοποιεί είναι η MDBX [17], παράγωγο έργο από την LMDB [18], η οποία λειτουργεί αποκλειστικά με αντιστοίχιση του αρχείου της στη μνήμη (memory mapping). Ως εκ τούτου, η cache του συστήματος αρχείων λειτουργεί αποδοτικά και ως read cache για τον Erigon, γεγονός που αξιοποιήθηκε από την παρούσα εργασία, όπως θα αναλυθεί σε επόμενο κεφάλαιο.

Υλοποίηση

Η οργάνωση της *State Database* του Erigon μοιάζει με αυτή του *Go-ethereum*, με αρκετές όμως σημαντικές διαφορές. Βλέπουμε και εδώ τα *stateObjects* τα οποία όμως περιέχονται στη δομή *IntraBlockState*, η οποία επίσης λειτουργεί ως read-write cache. Όπως φαίνεται και απ'το όνομά της, έχει ? SCOPE ? εντός ενός block, με την έννοια ότι οι τροποποιήσεις κρατώνται στη δομή αυτή μόνο για τη διάρκεια ενός block. Θα δούμε αργότερα τι γίνεται κατά την ολοκλήρωση του block.

Δομή *IntraBlockState*:

```
Erigon@inline_uv/core/state/intra_block_state.go:50-85
```

Η δημιουργία του *stateObject* γίνεται πάλι με ανάκτηση του αντίστοιχου *Account*:

```
Erigon@inline_uv/core/state/intra_block_state.go:553-554,557,559,560,569,
```

Η διαφορά γίνεται εμφανής στην ανάγνωση του *Account* μιας και εδώ δεν υπάρχει δομή *Trie*:

```
Erigon@inline_uv/core/state/plain_state_reader.go:70,73
```

Έτσι φτάνουμε στο επίπεδο της write-back write-cache που ονομάζεται (στον κώδικα) *mutation*. Αξίζει να σημειωθεί ότι οι αναγνώσεις πρέπει επίσης να κοιτάζουν τη write-cache μιας και μπορεί να έχει γίνει τροποποίηση που δεν έχει αποτυπωθεί ακόμα στην κύρια βάση.

```
Erigon@b1fef26/ethdb/olddb/mutation.go:126-135,
```

```
Erigon@b1fef26/ethdb/olddb/mutation.go:75-80
```

```
Erigon@b1fef26/ethdb/olddb/mutation.go:3,13,20-22
```

```
Erigon-lib@143cd51/kv/mdbx/kv_mdbx.go:872-879
```

Τα *stateObject* ανακτούν δεδομένα *Storage* με τον ίδιο τρόπο:

```
Erigon@inline_uv/core/state/state_object.go:159-160,162,174,
```

```
Erigon@inline_uv/core/state/state_object.go:184,186,188,211
```

```
Erigon@inline_uv/core/state/plain_state_reader.go:103,108
```

Όπως είδαμε και προηγουμένως, οι ανακτήσεις δεδομένων *Storage* προκαλούνται από εντολές *SLOAD/SSTORE*:

```
Erigon@inline_uv/core/vm/instructions.go:544-562
```

Επίσης, οι ανακτήσεις για *Accounts*, που γίνονται κατά την ολοκλήρωση της επεξεργασίας ενός *Block*:

```
Erigon@inline_uv/core/blockchain.go:90,123,159,178,
```

```
Erigon@inline_uv/core/blockchain.go:248,275,287
```

```
Erigon@inline_uv/core/state/intra_block_state.go:287-289
```

```
Erigon@inline_uv/core/state/plain_state_writer.go:125,127
```

Εδώ, που πρόκειται να γίνει η οριστικοποίηση του *Block*, πρέπει να γίνουν κάποιες εγγραφές. Οι εγγραφές αυτές θα καταλήξουν στη write-cache (*mutation*) αλλά, μιας και δεν υπάρχει η δομή *MPT* (όπως στο *go-ethereum*), η διαδικασία διαφέρει αρκετά. Η δομή *MPT* από μόνη της έχει την ιδιότητα ότι οι εγγραφές δεν πανω-γράφουν (*overwrite*) τις παλιές τιμές, αλλά, χάρη στην συνάρτηση κατακερματισμού⁵, παράγεται νέα (κενή) διεύθυνση στην οποία αποθηκεύονται. Αυτό δίνει τη δυνατότητα ιστορικής αναδρομής σε προηγούμενες καταστάσεις (*historic states*), απλά αλλάζοντας το *root hash*. Χωρίς το *MPT*, για να έχει ο *Erigon* την ίδια δυνατότητα, κρατάει πίνακες στη βάση με τις τροποποιήσεις που έχουν γίνει (*change-set*).

Η ενεργοποίηση είναι προαιρετική και γίνεται στην αρχή της εκτέλεσης:

```
Erigon@inline_uv/eth/stagedsync/stage_execute.go:200,228-232
```

⁵Τροποποιημένα δεδομένα δίνουν τελείως διαφορετικό αποτέλεσμα κατακερματισμού (*hash*) το οποίο χρησιμοποιείται ως νέα διεύθυνση, η οποία είναι άδεια (αγνοώντας συγκρούσεις κατακερματισμού).

Αν ενεργοποιηθεί, οι εγγραφές γίνονται με τον *changesetwriter*:

Erigon@inline_uv/core/state/change_set_writer.go:119,120,124,125,133,140,141

Η συνάρτηση *Encode()* κάνει απλή μετάφραση και ταξινόμηση:

Erigon@inline_uv/common/changeset/account_changeset.go:23-35

(υπάρχει και αντίστοιχη για *Storage*)

Η συνάρτηση *AppendDup()* είναι ισοδύναμη με την *Put()* στο *mutation* που είδαμε προηγουμένως.

Erigon@b1fef26/ethdb/olddb/mutation.go:204-206

Κεφάλαιο 4

Υλοποίηση συστήματος προανάκτησης

Στο κεφάλαιο αυτό περιγράφεται συνοπτικά η αρχιτεκτονική του συστήματος, καθώς και οι στόχοι και περιορισμοί που τέθηκαν για την υλοποίησή του. Η περιγραφή περιέχει τις βασικές πληροφορίες που χρειάζονται στο κεφάλαιο των αποτελεσμάτων, ώστε το επόμενο κεφάλαιο με την εκτενή περιγραφή να μπορεί να προσπεραστεί.

4.1 Σκοπός και περιορισμοί

Όπως περιγράφεται στο προηγούμενο κεφάλαιο, η διαδικασία του συγχρονισμού στον client *erigon* διεκπεραιώνεται σε διακριτά στάδια. Μιας και το στάδιο εκτέλεσης των συναλλαγών (*Execution stage*) διαρκεί για τον περισσότερο χρόνο της συνολικής διαδικασίας, η συγκεκριμένη εργασία περιορίστηκε στην επιτάχυνση αυτού του σταδίου και μόνο. Επομένως, κάθε επιλογή που εξετάζεται αναφέρεται στο συγκεκριμένο στάδιο και τα αποτελέσματα που αναφέρονται στα υπόλοιπα κεφάλαια, αγνοούν το χρόνο εκτέλεσης των υπόλοιπων σταδίων, κάποια εκ των οποίων (πχ μεταφόρτωσης του blockchain - *Downloading stage*) θα ήταν ανώφελο να μετρηθούν μιας και εξαρτώνται από συνθήκες δικτύου που *vary widely* ? χρονικά και γεωγραφικά.

Ο σκοπός της εργασίας είναι η βελτίωση της ταχύτητας αυτού του σταδίου με την παράλληλη προανάκτηση προβλεπόμενων εγγραφών από τη βάση δεδομένων. Το σύστημα περιορίζεται σε κοινό και εμπορικά διαθέσιμο υλισμικό (*hardware*). Επίσης, για να διασφαλιστεί η ασφάλεια της διαδικασίας εκτέλεσης των *contract*, λόγω της κρίσιμης φύσης της, οι ενέργειες της προανάκτησης δεν θα πρέπει να καθορίζουν τα δεδομένα που χρησιμοποιούνται και αποθηκεύονται στο *World State*. Έτσι, τυχών σφάλματα (*bugs*) ή αδυναμίες (*vulnerabilities*) σε αυτό το κομμάτι δεν θα *threat to compromise* την ασφάλεια ολόκληρου του συστήματος, και η εκτέλεση θα μπορεί σε κάθε περίπτωση να προχωρά κανονικά, πιθανών επιρρεασμένη μόνο ως προς το χρόνο εκτέλεσης. Σε καμία περίπτωση δεν θα πρέπει το κομμάτι της προανάκτησης να έχει τη δυνατότητα να τροποποιήσει δεδομένα του *World State*. Αυτό περιορίζει τη δυνατότητα προ-επεξεργασίας δεδομένων που θα μπορούσε να αποφέρει καλύτερες επιδόσεις αλλά θα έκανε πιο πολύπλοκη τη λογική του κρίσιμου σταδίου της εκτέλεσης που θα δημιουργούσε περισσότερες θέσεις για πιθανά σφάλματα. Μοναδική εξέραιση στα παραπάνω θα αποτελέσει η προανάκτηση των *block* η οποία όμως γίνεται ντετερμινιστικά και χωρίς αλλαγές από τον αρχικό κώδικα, παραμόνο η τοποθέτησή της σε ξεχωριστό νήμα (*thread*).

Τέλος, σε πιθανή κρίσημη αποτυχία και διακοπή της διαδικασίας προανάκτησης, η κύρια εκτέλεση του client θα συνεχίζεται σαν να μην υπήρχε εξ' αρχής.

4.2 Υψηλού επιπέδου περιγραφή

Η εργασία αποτελείται από 3 συνολικά κομμάτια που θα εξεταστούν παρακάτω: - Ιχνογράφηση (Tracing): Συλλογή στατιστικών και κώδικα των contracts - Ανάλυση (Analysis): Ανάλυση των contracts που συλλέχθηκαν στην ιχνογράφηση και σύνθεση των *predictors* - Προανάκτηση (Prefetching): Προανάκτηση και υποθετική εκτέλεση των *predictors* που δημιουργήθηκαν στην ανάλυση

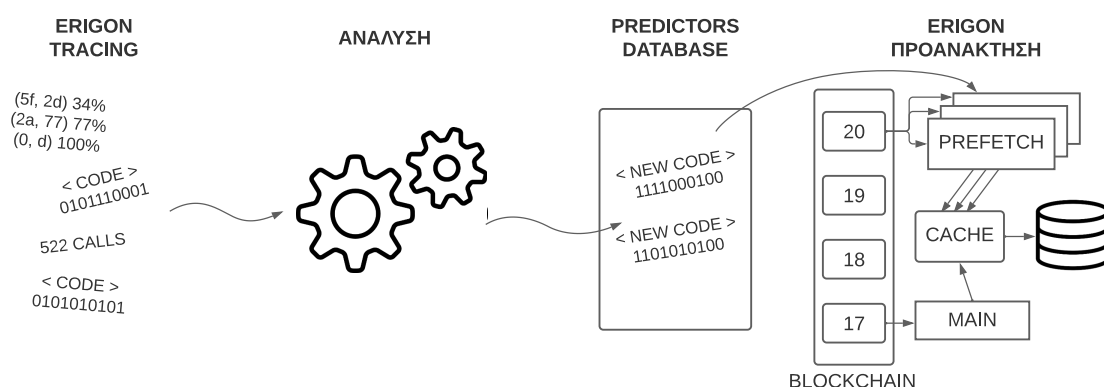


Figure 4.1: Υψηλού επιπέδου όψη του συστήματος.

Κεντρική αρμοδιότητα του συστήματος είναι προανάκτηση από τη βάση δεδομένων, των λογαριασμών, του κώδικα των contract καθώς και τις διευθύνσεις storage που αυτά χρησιμοποιούν. Η ανάκτηση των λογαριασμών και contracts που αναγράφονται πάνω στα transaction αποτελεί την απλή περίπτωση. Πέραν αυτών, γίνεται ανάκτηση και άλλων λογαριασμών και contracts που μπορεί να χρησιμοποιεί κατά την εκτέλεσή του ένα transaction, όπως φάνηκε και από τα εμφωλευμένα message calls σε προηγούμενο κεφάλαιο (σχήμα 3.1). Για τη δεύτερη αυτή περίπτωση, εκτελούνται υποθετικά οι *predictors*. Κάθε *predictor* αποτελεί σύντομο πρόγραμμα, με κώδικα παράγωγο από το contract από το οποίο προέρχεται. Η διαφορά του με το αρχικό είναι ότι έχουν κρατηθεί μόνο εντολές χρήσιμες για τους σκοπούς της προανάκτησης και έχουν αφαιρεθεί τμήματα που θεωρούνται σπάνια, με βάση τα δεδομένα που συλλέγονται στο *tracing*.

Η εκτέλεση της προανάκτησης γίνεται λίγα block πιο μπροστά από την κύρια εκτέλεση. Αυτή η διαφορά είναι περιορισμένη εκ των προτέρων και η μέγιστη τιμή της αναφέρεται ως *readahead*, η οποία περιγράφεται πιο αναλυτικά στην παράγραφο 5.3.2. Στην περίπτωση που απεικονίζεται στο σχήμα 4.1, η διαφορά αυτή είναι ίση με 3.

Τα δεδομένα που προανακτώνται δεν χρησιμοποιούνται κατά την κύρια εκτέλεση. Ο μόνος τρόπος που επιρρεάζουν την εκτέλεση είναι χρονικά, "ζεσταίνοντας" την cache του συστήματος αρχείων.

Μία πιο αναλυτική απεικόνιση του συστήματος προανάκτησης φαίνεται στο σχήμα 5.9 και περιγράφεται στην ενότητα 5.3.

Τα κομμάτια της ιχνογράφησης και ανάλυσης στις δοκιμές του επόμενου κεφαλαίου εκτελούνται ξεχωριστά, για λόγους απλότητας, αλλά σε μια πραγματική υλοποίηση αναμένουμε να τρέχουν ταυτόχρονα, παράλληλα με την κύρια εκτέλεση. Ο χρόνος της κύριας εκτέλεσης είναι σημαντικά μεγαλύτερος από το χρόνο που διαρκεί η ανάλυση για τα contract της αντίστοιχης περιόδου.

Κεφάλαιο 5

Αναλυτική περιγραφή του συστήματος

Στο κεφάλαιο αυτό περιγράφεται λεπτομερώς το σύστημα που υλοποιήθηκε, όπως και η λογική και υποθέσεις στις οποίες βασίστηκαν οι επιλογές που έγιναν κατά τη σχεδίαση.

5.1 Tracing

Σε αυτό το κομμάτι γίνεται συλλογή δεδομένων κατά την πραγματική εκτέλεση των transaction.

Πιο συγκεκριμένα, καταγράφονται τα code hashes των contract που εκτελούνται καθώς και το πλήθος των κλήσεων (message call) που γίνονται στο καθένα, σε μια δομή αντιστοίχισης από code hash σε πλήθος κλήσεων.

```
var CONTRACT_CODE_COUNT = map[Hash]uint{}
```

Επίσης, δεδομένου ότι το code hash έχει πάνω από ένα προκαθορισμένο όριο κλήσεων, αποθηκεύεται και ο κώδικάς του, ο οποίος θα περάσει στον αναλυτή, όπως περιγράφεται στο επόμενο στάδιο *Analysis*.

```
var CONTRACT_CODE = map[Hash][]byte{}
```

Τέλος, για κάθε εντολή διακλάδωσης που εκτελείται καταγράφεται η διεύθυνση του program counter πριν και μετά την εκτέλεση, καθώς και το code hash και ένας μοναδικός αριθμός που αντιστοιχεί στο message call που εκτελείται εκείνη τη στιγμή. Αποθηκεύεται το πλήθος αυτών των τετράδων.

```
// code hash -> source -> destination -> call id -> counter
var JUMP_DST_CALLCOUNT = map[Hash]map[uint32]map[uint32]map[int]uint{}
```

Στην μέχρι τώρα υλοποίηση, το πλήθος αυτό αγνοείται και κρατάται μόνο η ύπαρξή του (δηλαδή αν είναι διάφορο του 0), ώστε να γίνει καταμέτρηση μόνο ως προς το πλήθος των κλήσεων και όχι των διακλαδώσεων. Η καταμέτρηση αφορά το κάθε code hash ξεχωριστά και γίνεται μετά την εκτέλεση για λόγους απλότητας, αλλά δεν υπάρχει λόγος για τον οποίο δεν μπορεί να γίνει σε προκαθορισμένα χρονικά διαστήματα ή όταν ξεπεραστεί κάποιο όριο. Σκοπός τη καταμέτρησης είναι η συλλογή για κάθε code hash που θα αναλυθεί των συχνών διακλαδώσεων. Η συλλογή αυτή είναι προαιρετική αλλά είναι σημαντική για καλά αποτελέσματα στο στάδιο της ανάλυσης, οπότε σε όλα τα πειράματα της εργασίας είναι ενεργοποιημένη. Ο λόγος που κρατάται προαιρετική είναι για πολύ λίγα contracts

στα οποία δεν υπάρχουν πολλές κλήσεις για να εξαχθούν χρήσιμα δεδομένα και τα οποία επομένως περνάνε στην ανάλυση χωρίς γνωστές διακλαδώσεις.

5.2 Ανάλυση

Το κομμάτι της ανάλυσης αποτελεί το μεγαλύτερο μέρος αυτής της εργασίας. Ο σκοπός του είναι να διαβάσει τον κώδικα (runtime bytecode) των contract καθώς συλλέγονται στο προηγούμενο στάδιο, καθώς και (προαιρετικά) τις γνωστές διακλαδώσεις για υποβοήθηση, και να παράγει ένα νέο κώδικα για το καθένα, που εδώ ονομάζουμε ? προβλεπτή ? (*predictor*). Οι *predictors* χρησιμοποιούνται στο επόμενο στάδιο για την προανάκτηση των δεδομένων που προβλέπεται ότι θα χρησιμοποιήσει η κάθε κλήση (message call).

Η ανάλυση γίνεται στατικά¹, ξεκινώντας με disassembly του EVM κώδικα που βρίσκεται σε δυαδική αναπαράσταση (*bytecode*).

TODD

Μετά χωρίζεται σε basic block (σχήμα 5.1). Αρκετά χρήσιμη για αυτό το στάδιο είναι η επιβολή της εντολής *JUMPDEST* του EVM σε όλες τις διευθύνσεις στις οποίες επιτρέπεται να φτάσει η ροή της εκτέλεσης από εντολή άλματος.

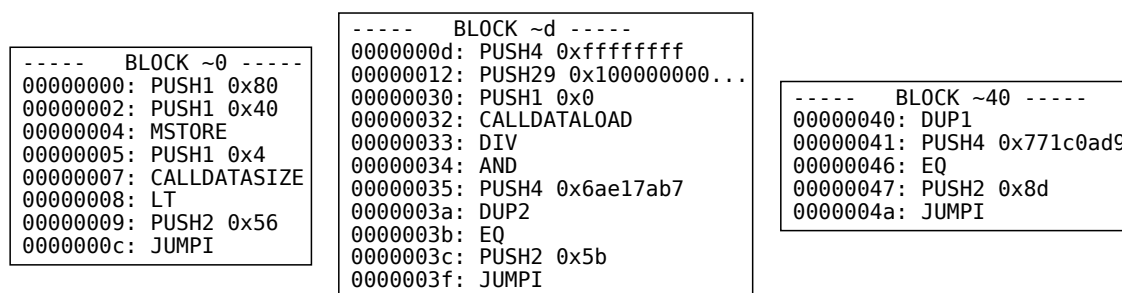


Figure 5.1: .

Στη συνέχεια (σχήμα 5.2), οι εντολές που περιέχουν τα basic block μετατρέπονται σε μορφή Static Single Assignment (SSA) [19].

Εδώ φαίνεται πώς γίνεται και μετατροπή της αρχιτεκτονικού μοντέλου από στοίβας του EVM σε άπειρων καταχωριτών του SSA, εξαλείφοντας εντολές όπως *PUSH*, *POP* και *SWAP*, ενώ δημιουργούνται νέες ψευδο-εντολές Φ (ϕ). Σε αυτό το σημείο ξεκινάει και η πρώτη υποθετική βελτιστοποίηση (speculative optimization) που κάνουμε με το σημάδεμα (marking) για παράκαμψη (skip) των block που περιέχουν εντολές που αυθαίρετα θεωρούμε σπάνιες (*CREATE*, *CREATE2*, *INVALID*, *REVERT*, *SELFDESTRUCT*). Block που έχουν σημειωθεί για παράκαμψη θα αγνοούνται.

Η διαδικασία συνεχίζει με τη σύνδεση των block (σχήμα 5.3) για τη δημιουργία μιας πρώτης εικόνας του γράφου ελέγχου ροής (CFG). Ας σημειωθεί ότι ο CFG σε αυτό το στάδιο δεν είναι ο πραγματικός αλλά μια εκτίμηση. Γενικώς, σε επόμενα βήματα θα γίνονται αλλαγές στην εκτίμηση του CFG για να φτάσει πιο κοντά στον πραγματικό, αλλά ο βέβαιος

¹εκτός από τις γνωστές διακλαδώσεις που αναφέρθηκαν προηγουμένως και θα μπορούσε κανείς να ισχυριστεί ότι πρόκειται και για δυναμική ανάλυση

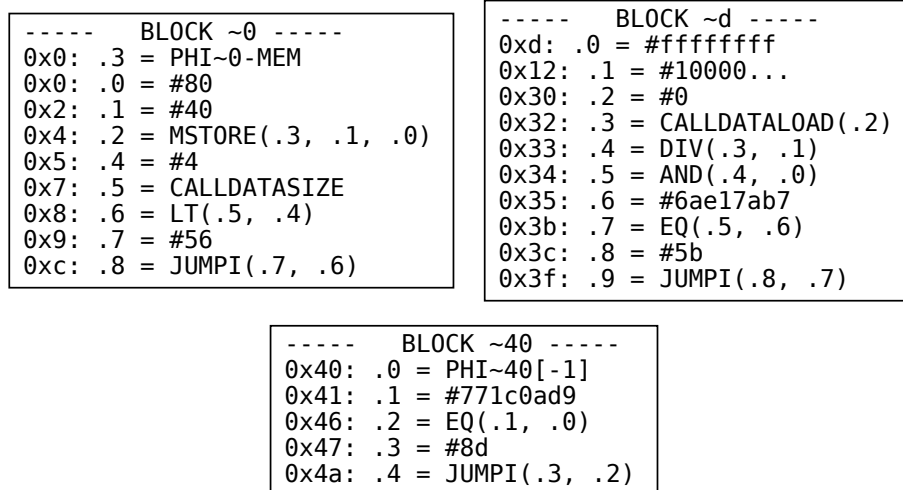


Figure 5.2: .

προσδιορισμός του είναι αδύνατος μιας και το EVM είναι αρχιτεκτονική turing complete [2, απόδειξη σε παράρτημα]. Εδώ χρησιμοποιείται και η πληροφορία για τις γνωστές διακλαδώσεις (αν υπάρχουν) από το προηγούμενο στάδιο. Κατά τη διαδικασία αυτή, οι ψευδο-εντολές Φ αποκτούν και της παραμέτρους (argument/operand) τους.

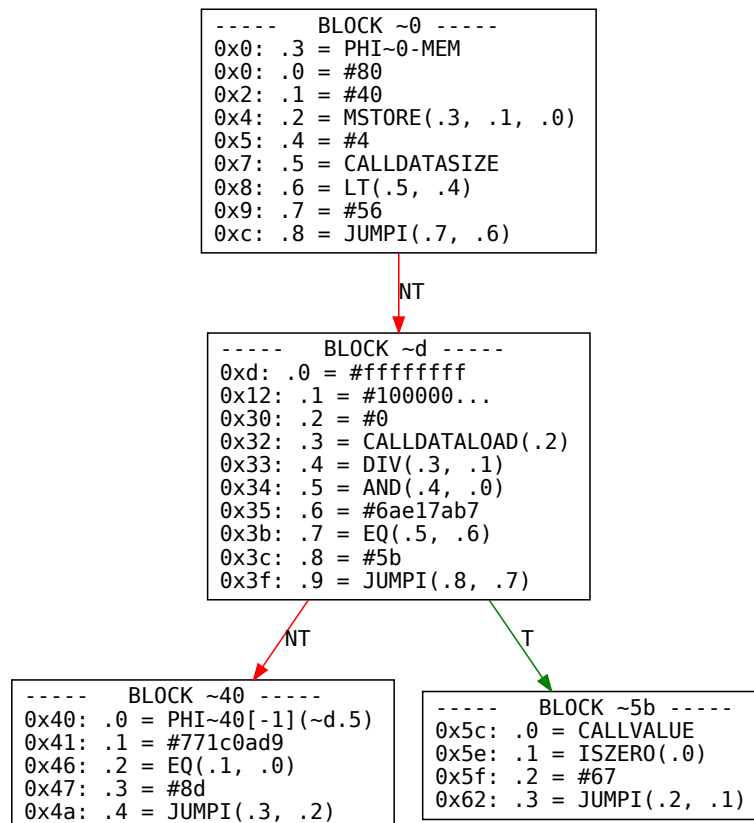


Figure 5.3: .

Ακολουθεί το βήμα της 'βελτιστοποίησης' (σχήμα 5.4), το οποίο είναι το μεγαλύτερο χρονικά και από θέμα έκτασης κώδικα, το οποίο εκτελεί και άλλες λειτουργίες πέραν της βελτιστοποίησης. Χρησιμοποιεί τον αλγόριθμο *worklist* [19] για να λύσει, μεταξύ άλλων,

τα παρακάτω dataflow προβλήματα [20]:

- Αντικατάσταση σταθερών (constant folding) - Αλγεβρική απλοποίηση - Σύνολα πιθανών τιμών Το τελευταίο χρειάζεται για τον προσδιορισμό των πιθανών τιμών των παραμέτρων των εντολών διακλάδωσης (*JUMP*, *JUMPI*), και συγκεκριμένα των πιθανών διευθύνσεων άλματος, σε περίπτωση που δεν είναι γνωστές, για να γίνουν οι αντίστοιχες τροποποιήσεις στην εκτίμηση του CFG. Πέραν αυτών, γίνεται αντικατάσταση και ορισμένων εντολών με σταθερές που είναι ήδη γνωστές, όπως η ταυτότητα αλυσίδας (*CHANID*), η τιμή του program counter (*PC*) και δεδομένα σχετικά με τον κώδικα που αναλύεται (*CODESIZE*, *CODECOPY*).

Επιπλέον, γίνεται και αναδρομική επέκταση του σημαδέματος των block για παράκαμψη σε αυτά που καταλήγουν αναγκαστικά σε σημαδεμένα block, καθώς και εξάλειψη των απρόσιτων block (unreachable code elimination), τα οποίο μπορεί να έχει προκληθεί από τη διαδικασία σημαδέματος.

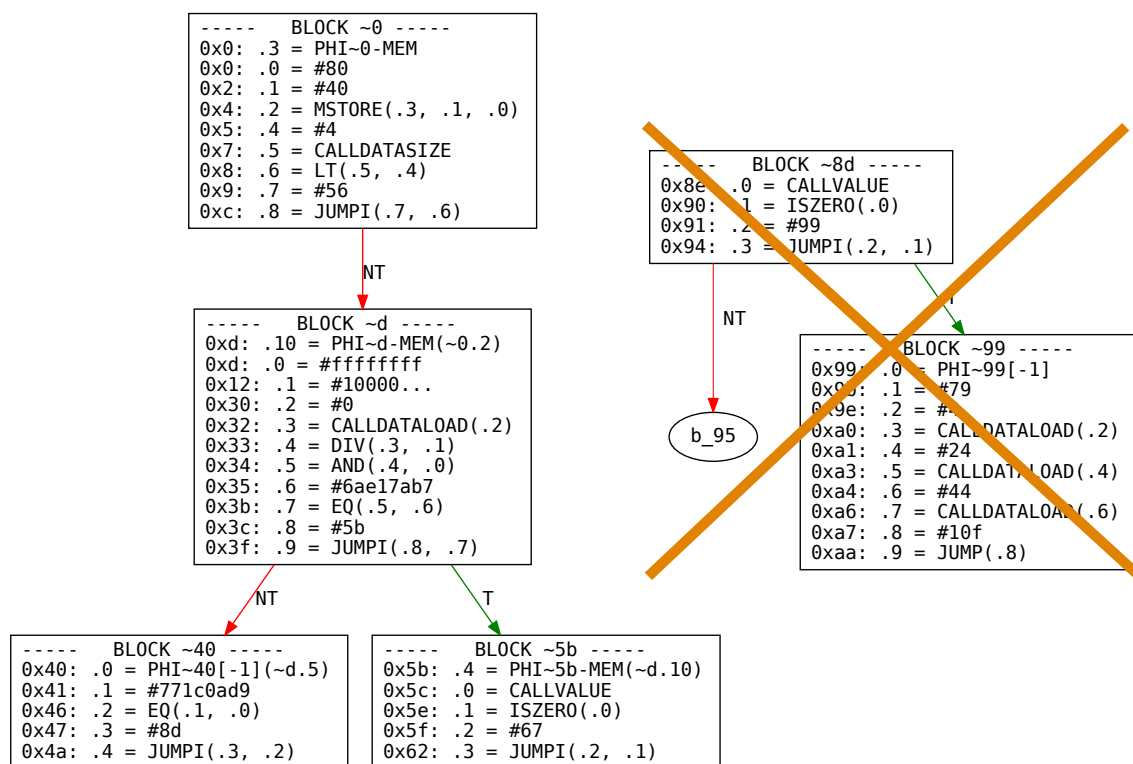


Figure 5.4: .

Η διαδικασία συνεχίζει με την επιλογή των εντολών (σχήμα 5.5) που χρειάζεται να υπ-
άρχουν στην έξοδο, δηλαδή στους predictors. Αυτές είναι εντολές που είτε χρησιμοποιούν τη
μνήμη *storage* του constact (*SLOAD*, *SSTORE*), είτε προκαλούν νέα message calls (*CALL*,
CALLCODE, *DELEGATECALL*, *STATICCALL*), είτε τερματίζουν την κλήση και επιστρέφουν
δεδομένα (*RETURN*, *REVERT*). Από αυτές, αγνοήται η εντολή *REVERT* που προηγουμένως
χαρακτηρίστηκε ως σπάνια και επομένως δεν συμπεριλήφθηκε στην ανάλυση. Όπως εί-
ναι προφανές, δεν αρκεί μόνο η επιλογή αυτών των εντολών, αλλά χρειάζονται και όλες οι
υπόλοιπες από τις οποίες εξαρτώνται, καθώς και τα block στα οποία περιλαμβάνονται, με
την προσθήκη των απαραίτητων εντολών διακλάδωσης και εξαρτήσεων αυτών. Επομένως
προκύπτει ένα νέο (backward) dataflow πρόβλημα, το οποίο επιλύεται πάλι με τον αλ-

γόριθμο worklist. Μία χρήσιμη παρενέργεια αυτής της διαδικασίας είναι και η εξάλειψη άχρηστου κώδικα (dead code elimination), μιας και εντολές που δεν χρησιμοποιούνται (από το πρόγραμμα εξόδου) δεν πρόκειται να επιλεγούν.

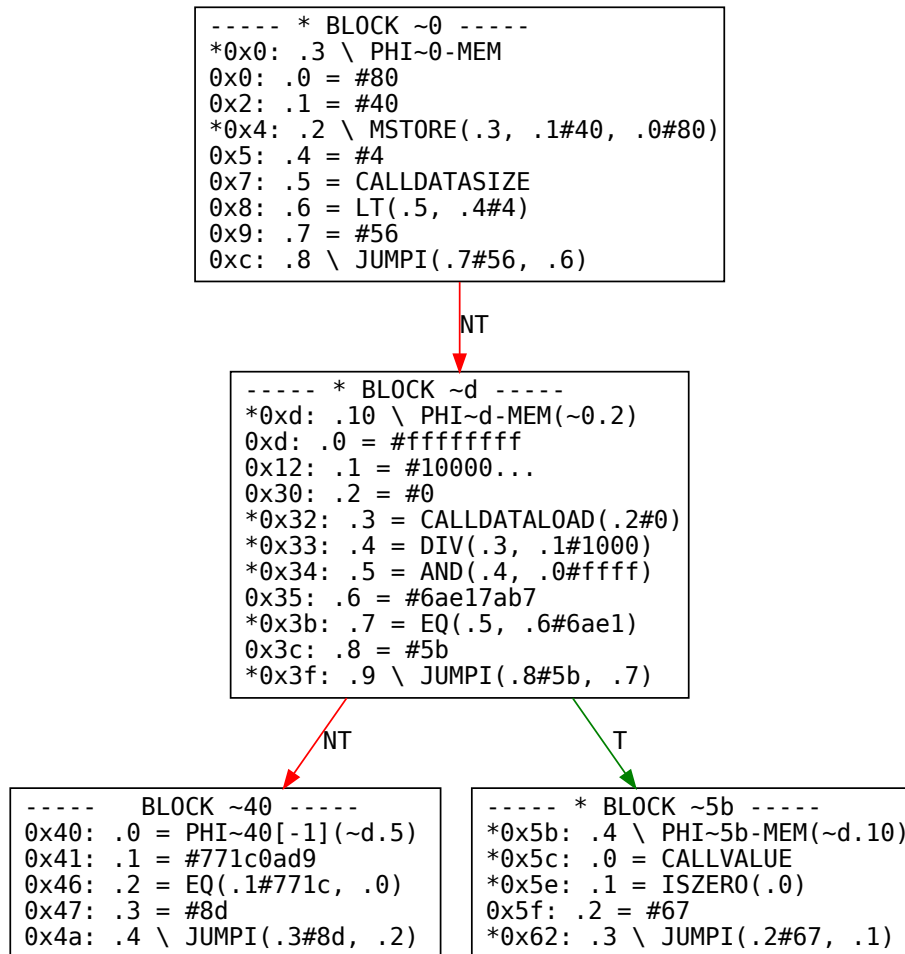


Figure 5.5: Οι επιλεγμένες εντολές έχουν ένα αστεράκι * στα αριστερά των διευθύνσεων, ομοίως και τα επιλεγμένα block αριστερά του ονόματός τους.

Στο επόμενο βήμα (σχήμα 5.6), γίνεται νέα εκτέλεση του αλγορίθμου worklist (forward dataflow), για την αρίθμηση των εκφράσεων των επιλεγμένων εντολών (global value numbering).

Ταυτόχρονα (σχήμα 5.7), προσδιορίζεται σε κάθε block το σύνολο των διαθέσιμων αριθμημένων εκφράσεων-εντολών (available expressions) για την εξάλειψη κοινών υποεκφράσεων (common subexpression elimination).

Με βάση την αρίθμηση γίνεται και υπολογισμός των τελικών εκφράσεών τους (όνομα εντολής και παράμετροι, σχήμα 5.8), που θα αποτελέσουν τις εντολές του προγράμματος εξόδου (predictor). Οι εντολές αυτές διαφέρουν από τις εντολές του EVM, αφενώς γιατί δεν είναι σε αρχιτεκτονική στοίβας αλλά καταχωρητών, και αφετέρου γιατί κάποιες από αυτές δεν υπάρχουν στο σύνολο εντολών (ISA) του EVM. Παράδειγμα τέτοιων εντολών είναι οι εντολές Φ (PHI), εισαγωγής σταθερών (CONSTANT) και αντιγραφής (COPY), που θα αναφερθούν παρακάτω. Θα αναφέρουμε τη νέα αρχιτεκτονική συνόλου εντολών που προκύπτει ως EVM-like, μιας και μοιάζει αρκετά με την αρχική.

```

----- ON MAP -----
1 = #40
2 = #80
3 = V~0.2-MSTORE(v~0.3-PHIxb232-0B, #40, #80)-xad80-NV
4 = #10000...
5 = #0
6 = #6ae17ab7
7 = #5b
8 = #ffffffff
9 = V~d.3-CALLDATALOAD(#0)-x15b2
10 = V~d.4-DIV(v~d.3-CALLDATALOADx15b2, #10000...)-x4ea2
11 = V~d.5-AND(v~d.4-DIVx4ea2, #ffffffff)-x4954
12 = V~d.7-EQ(v~d.5-ANDx4954, #6ae17ab7)-x30c9
13 = V~d.9-JUMPI(#5b, v~d.7-EQx30c9)-x2f1e-NV
14 = #67
15 = V~5b.0-CALLVALUE()-x78d0
16 = V~5b.1-ISZERO(v~5b.0-CALLVALUEx78d0)-x8a44
17 = V~5b.3-JUMPI(#67, v~5b.1-ISZEROx8a44)-x9d52-NV

```

Figure 5.6: .

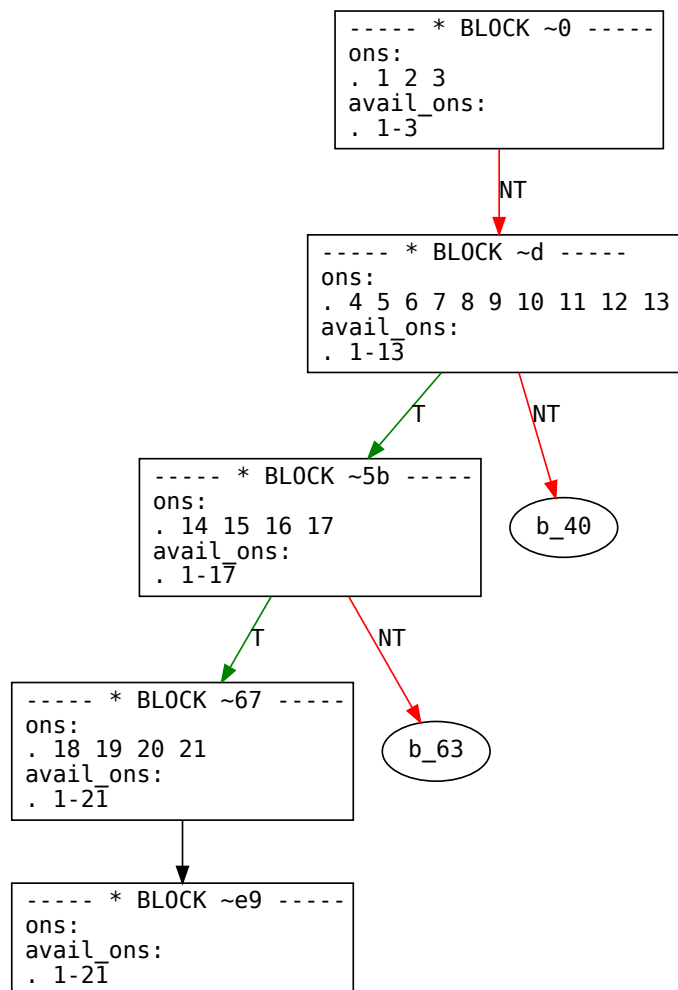


Figure 5.7: .


```

----- ON CALCS -----
0 = ON_0_RESERVED
1 = #40
2 = #80
3 = MSTORE 0 1 2
4 = #10000...
5 = #0
6 = #6ae17ab7
7 = #5b
8 = #ffffffff
9 = CALLDATALOAD 5
10 = DIV 9 4
11 = AND 10 8
12 = EQ 11 6
13 = JUMPI 7 12
14 = #67
15 = CALLVALUE
16 = ISZERO 15
17 = JUMPI 14 16
18 = #24

```

Figure 5.8: Με την αρίθμηση των επιλεγμένων εντολιών και υπολογισμό των εκφράσεών τους ως προς την αρίθμηση, δεν χρειάζονται ποια οι αρχικές.

Με τις επιλεγμένες εντολές έτοιμες, έρχεται το τελευταίο βήμα, της σύνθεσης των predictors, δηλαδή του κώδικα EVM-like που θα δοθεί ως έξοδος. Η διαδικασία περνά από όλα τα επιλεγμένα block με τη σειρά που έχουν στον αρχικό κώδικα, δηλαδή σύμφωνα με τις διευθύνσεις τους.

Σε κάθε block, γράφεται αρχικά η επικεφαλίδα του με το όνομά του (διεύθυνση έναρξης του στον αρχικό κώδικα) ακολουθούμενη από μία λίστα από τα ονόματα των block τα οποία είναι ? προγενέστερα ? (predecessor) αυτού στον CFG, δηλαδή αυτά από τα οποία μπορεί η ροή ελέγχου να φτάσει άμεσα στο παρόν block. Οι λόγοι για την ύπαρξη της επικεφαλίδας είναι τρεις. - Χρειάζεται να υπάρχει η αρχική διεύθυνση του block, μιας και δεν έχει γίνει κάποια μετάφραση από τις διευθύνσεις της εισόδου στις νέες για την έξοδο. - Οι εντολές είναι ακόμα σε μορφή SSA και θα μείνουν ακόμα και στην τελική έξοδο. Αυτό συνεπάγεται ότι θα μείνουν και οι εντολές Φ, για τον υπολογισμό των οποίων είναι αναγκαίο να συμπεριλαμβάνεται πληροφορία σχετικά με την διεύθυνση (block) προέλευσης των αλμάτων. - Η επικεφαλίδα κάνει πιο εύκολη την ανάγνωση του παραγώμενου κώδικα. Η έξοδος είναι σε μορφή απλού κειμένου για να είναι δυνατή η εξέτασή του και ο εντοπισμός σφαλμάτων κατά την ανάπτυξη του αναλυτή. Στο στάδιο μετα-επεξεργασίας των predictors θα γίνει απλοποίηση των επικεφαλίδων και θα αντικατασταθούν από απλούστερες εντολές.

Μετά την επικεφαλίδα τοποθετούνται όλες οι εντολές εισαγωγής σταθερών (CONSTANT).

Ακολουθούν οι εντολές Φ (PHI) για τις οποίες χρειάζεται ιδιαίτερη προσοχή. Σε αντίθεση με τις υπόλοιπες εντολές, οι εντολές Φ πρέπει να υπολογιστούν όλες ταυτόχρονα [19]. Η ταυτόχρονη εκτέλεσή τους όμως θα επέφερε μεγάλες αλλαγές στη λογική του ερμηνευτή (interpreter) που θα χρησιμοποιηθεί στο στάδιο της προανάκτησης, μιας και η λογική που ακολουθεί είναι να εκτελεί κάθε εντολή ξεχωριστά, χωρίς ειδική μεταχείριση κάποιας κλάσης εντολών. Επομένως, επιλέγουμε να βρούμε ένα τρόπο να εκτελούνται οι εντολές Φ ξεχωριστά, αλλά το αποτέλεσμα τους να είναι το ίδιο με το να είχαν εκτελεστεί ταυτόχρονα, όπως το υποθετικό μοντέλο που έχουμε ακολουθήσει απαιτεί.

Το πρόβλημα προκύπτει όταν υπάρχουν εξαρτήσεις μεταξύ τους, όπως φαίνεται στο

παρακάτω παράδειγμα:

1 = PHI 5 7

4 = PHI 1 8

5 = PHI 1 9

Μία τέτοια κατάσταση μπορεί να προκύψει εύκολα από την εντολή *SWAP* του EVM.

Η εξάρτηση μεταξύ των εντολών 1 και 4 λύνεται εύκολα με αλλαγή της σειράς τους, αλλά η κυκλική εξάρτηση των 1 και 5 δεν μπορεί να λυθεί με οποιαδήποτε αναδιάταξη. Είναι αναγκαία η εισαγωγή μιας εντολής αντιγραφής (*COPY*) σε ένα προσωρινό καταχωρητή:

0 = COPY 5

4 = PHI 1 8

5 = PHI 1 9

1 = PHI 0 7

Για την επίλυση αυτού του προβλήματος χρησιμοποιήθηκε ένας απλός αλγόριθμος για την εύρεση κύκλων στο γράφο των εντολών Φ και την αφαίρεση (αφθαίρετα) ακμών ώστε να γίνει μη κυκλικός². Κάθε ακμή που αφαιρείται αντιστοιχίζεται σε μία νέα εντολή *COPY* που παράγεται. Το γεγονός ότι ο γράφος γίνεται ακυκλικός επιτρέπει την τοπολογική του διάταξη, η οποία εγγυάται ότι η ανάθεση ενός καταχωρητή θα έπεται όλων των χρήσεων του από τις εντολές Φ , λύνοντας το πρόβλημα. Αξίζει να σημειωθεί ότι η εγγύηση αυτή είναι ακριβώς αντίθετη απ' ότι συμβαίνει με τις υπόλοιπες εντολές, στις οποίες η χρήση έπεται του ορισμού (στη μορφή SSA, η ανάθεση είναι ταυτόσημη με το ορισμό).

Τελευταίες τοποθετούνται οι υπόλοιπες εντολές του block. Στο τέλος μπορεί να χρειαστεί να τοποθετηθεί και μια εντολή άλματος χωρίς συνθήκη, σε περίπτωση που το επόμενο επιλεγμένο block δεν είναι το επόμενο block στον αρχικό κώδικα.

Αυτή η διαδικασία μας δίνει τον τελικό predictor σε αναπαράσταση απλού κειμένου.

TODO

Τέλος, για πιο αποδοτική εκτέλεση του predictor στο επόμενο μέρος της προανάκτησης, περνάει από ένα στάδιο μετά-επεξεργασίας. Σε αυτό το στάδιο εντοπίζονται όλες οι επικεφαλίδες των block και αντικαθίστανται από μία μόνο εντολή *BLOCKID* με παράμετρο το όνομα (αρχική διεύθυνση) του block, η οποία υποδεικνύει στον ερμηνευτή το όνομα του block που εκτελεί. Οι επικεφαλίδες που αφαιρέθηκαν χρησιμοποιούνται για τη δημιουργία του *block table*, μιας αντιστοίχισης από διευθύνσεις αρχικού κώδικα σε διευθύνσεις του predictor και ονόματα block προορισμού.

```
type BlockTableEntry struct {
    Index int      // Διεύθυνση στον κώδικα του predictor
    Edges [uint16] // Ονόματα επιτρεπόμενων block προορισμού, με την ίδια σειρά με τις παραμέτρους των PHI
}

type BlockTable map[uint16]BlockTableEntry
//          |
//          '--- διεύθυνση του αρχικού κώδικα
```

²Το γενικό πρόβλημα είναι το *feedback arc set*, η εύρεση του ελαχίστου είναι NP-hard [21].

Ο λόγος που χρειάζεται ο block table και η εντολή *BLOCKID* είναι γιατί οι διευθύνσεις των εντολών διακλάδωσης (*JUMP*, *JUMPI*) παίρνουν ακόμα τιμές που αναφέρονται στις διευθύνσεις του αρχικού κώδικα, οπότε χρειάζεται αυτό το επίπεδο μετάφρασης των διευθύνσεων. Η εναλλακτική λύση της αλλαγής των μεταβλητών που καταλήγουν σαν παράμετροι των εντολών διακλάδωσης θα είχε περιορισμένο αποτέλεσμα, διότι όπως προαναφέρθηκε, η αρχιτεκτονική EVM είναι turing complete, οπότε είναι αδύνατη η μετατροπή στη γενική περίπτωση (οι διευθύνσεις μπορούν να προκύπτουν στον αρχικό κώδικα ως αποτέλεσμα οποιουδήποτε υπολογισμού).

Η τελική ενέργεια αυτού του σταδίου είναι η σειριοποίηση (serialization) και κωδικοποίηση (encoding) σε διυαδική μορφή του predictor μαζί με το block table, για να αποθηκευτεί στη βάση δεδομένων.

```
# Kyber: Reserve
INFO cli | Analyzing code/h_1527....runbin.hex
INFO cli | Output is code/h_1527....evmlike
INFO cli | Skip is enabled
INFO cli | Using known jump edge file ../jump_edges.json
WARNING cli | Code hash h_1527... not found in jump edges file
INFO cli | Will pick SLOAD,SSTORE,CALL,CALLCODE,DELEGATECALL,STATICCALL,RETURN
INFO assembly.disassemble | At byte 8277 ValueError('Bad_opcode_238')
INFO analysis.optimize | Running optimizer for up to 26554 ms, around 265540 updates
INFO analysis.optimize | Reached 0 updates
INFO analysis.optimize | Optimizer complete after 15079 updates

real 0m0.213s
user 0m0.210s
sys 0m0.004s

# ENS: Base Registrar Implementation
INFO cli | Analyzing code/h_1555.runbin.hex
INFO cli | Output is code/h_1555.evmlike
INFO cli | Skip is enabled
INFO cli | Using known jump edge file ../jump_edges.json
INFO cli | Will pick SLOAD,SSTORE,CALL,CALLCODE,DELEGATECALL,STATICCALL,RETURN
INFO assembly.disassemble | At byte 10360 ValueError('Bad_opcode_44')
INFO analysis.optimize | Running optimizer for up to 30720 ms, around 307200 updates
INFO analysis.optimize | Reached 0 updates
INFO analysis.optimize | Optimizer complete after 624 updates

real 0m0.065s
user 0m0.062s
sys 0m0.004s
```

Εκτύπωση (log) του αναλυτή για δύο τυχαίους κώδικες contract, ακολουθούμενο από το συνολικό χρόνο που χρειάστηκε. Η προειδοποίηση (warning) αναφέρει ότι για το συγκεκριμένο δεν υπήρχαν στοιχεία γνωστών δικλαδώσεων, παρ' όλ' αυτά η ανάλυση συνέχισε κανονικά. Σε αντίθεση, το δεύτερο που έχει στοιχεία (δεν έχει προειδοποίηση) ολοκληρώνεται αρκετά πιο γρήγορα και με λιγότερες επαναλήψεις (updates) στη worklist της βελτιστοποίησης (optimizer). Το σφάλμα ValueError δείχνει ότι ο κώδικας στην είσοδο περιέχει δεδομένα που δεν είναι έγκυρες εντολές και αγνοούνται, στη συγκεκριμένη περίπτωση είναι μετα-δεδομένα

CBOR (Solidity contract metadata) [22].

5.3 Προανάκτηση

Το τελευταίο κομμάτι του συστήματος είναι αυτό της προανάκτησης. Το κομμάτι αυτό αποτελεί επέκταση του *erigon client* και διαμορφώνεται σε δύο τμήματα. - Το πρώτο τμήμα αποτελεί τροποποίηση του *execution stage* στον κώδικα του *erigon* - Το δεύτερο τμήμα είναι ένα επιπλέον *package* στο ίδιο *repository* με τον *erigon* το οποίο προσθέτει τη δυνατότητα υποθετικής εκτέλεσης των *predictors* που παρήχθησαν από την ανάλυση Πέραν αυτών, υπάρχουν και λίγες ακόμα τροποποιήσεις, κυρίως στη *write-cache* του *erigon*, για να υποστηρίζεται η ταυτόχρονη εκτέλεση του κυρίως νήματος που διεκπερεώνει την πραγματική εκτέλεση των *transaction* καθώς και των νημάτων που κάνουν την προανάκτηση. Ο διαχωρισμός των δύο τμημάτων, εκτός από την αντιπροσώπευση των αλλαγών στον κώδικα (*code base*), δείχνει και την διαφορά της βεβαιότητας της προανάκτησης:

-το πρώτο τμήμα περιέχει κώδικα που διαβάζει δεδομένα όπως τα μελλοντικά *block* και οι διευθύνσεις αποστολέα (*from*) και παραλήπτη/δικαιούχου (*to*) των *transaction*, τα οποία είναι σίγουρο πως θα χρειαστούν στο κύριο νήμα, - ενώ το δεύτερο περιέχει κώδικα που εκτελεί *predictors* και επομένως τα δεδομένα που ανακτά μπορεί να μην χρειάζονται τελικά στην πραγματική εκτέλεση.

Μιας και το σύστημα είναι εγγενώς πολυνηματικό, είναι σκόπιμο να περιγραφεί η διάρθρωσή του ως προς τα νήματα που εκτελούνται και τις λειτουργίες που εκτελούν. Η περιγραφή που ακολουθεί αποτυπώνεται και στο σχήμα 5.9.

Η επικοινωνία μεταξύ των νημάτων γίνεται με τα κανάλια (*channel*) της γλώσσας *Go*, ενώ ο έλεγχος ταυτόχρονης πρόσβασης (*concurrency control*) σε κοινές δομές δεδομένων επιτυγχάνεται και με άλλες μεθόδους ανάλογα την περίπτωση.

Όπως και στην αρχική έκδοση (δηλαδή χωρίς τις τροποποιήσεις) του *erigon*, υπάρχει ένα κύριο νήμα (*main thread*), το οποίο εκτελεί σειριακά τα *transaction* που περιέχονται στα *block* προς εκτέλεση. Το νήμα αυτό έχει μία τοπική (*thread local*) *cache* για αναγνώσεις και εγγραφές (*read-write*), τύπου *IntraBlockState* η οποία κρατά εσωτερικά τις εγγραφές (*write-back*) μέχρι το τέλος του *block* (όπως περιγράφηκε σε προηγούμενο κεφάλαιο).

5.3.1 Νήμα προανάγνωσης

Τα *block* που επεξεργάζεται το κύριο νήμα προέρχονται από ένα άλλο νήμα, το νήμα προανάγνωσης *block*. Το νήμα αυτό διαβάζει *block* από τη βάση δεδομένων, τα αποσειριοποιεί και απο-κωδικοποιεί (*decode*), και για κάθε *transaction* που περιέχεται διαβάζει τη διεύθυνση του αποστολέα, η οποία έχει ήδη υπολογιστεί από τις μεταβλητές *v*, *r*, *s* των αντίστοιχων υπογραφών σε προγενέστερο στάδιο (*Senders stage*). Τα *block* που ετοιμάζει δίνονται στο κύριο νήμα (ως δείκτες) μέσω του καναλιού *blockChan*, το οποίο καθορίζει και τη σχετική θέση σε αριθμό *block* των δύο νημάτων. Εκτός από τη μεταβίβαση των *block*, έχει και μία δεύτερη λειτουργία, να κατανέμει τα *transaction* που περιέχονται στα νήματα που εκτελούν την προανάκτηση, μέσω του καναλιού *txChan*. Τα κανάλια θα περιγραφούν με περισσότερη λεπτομέρεια στη συνέχεια.

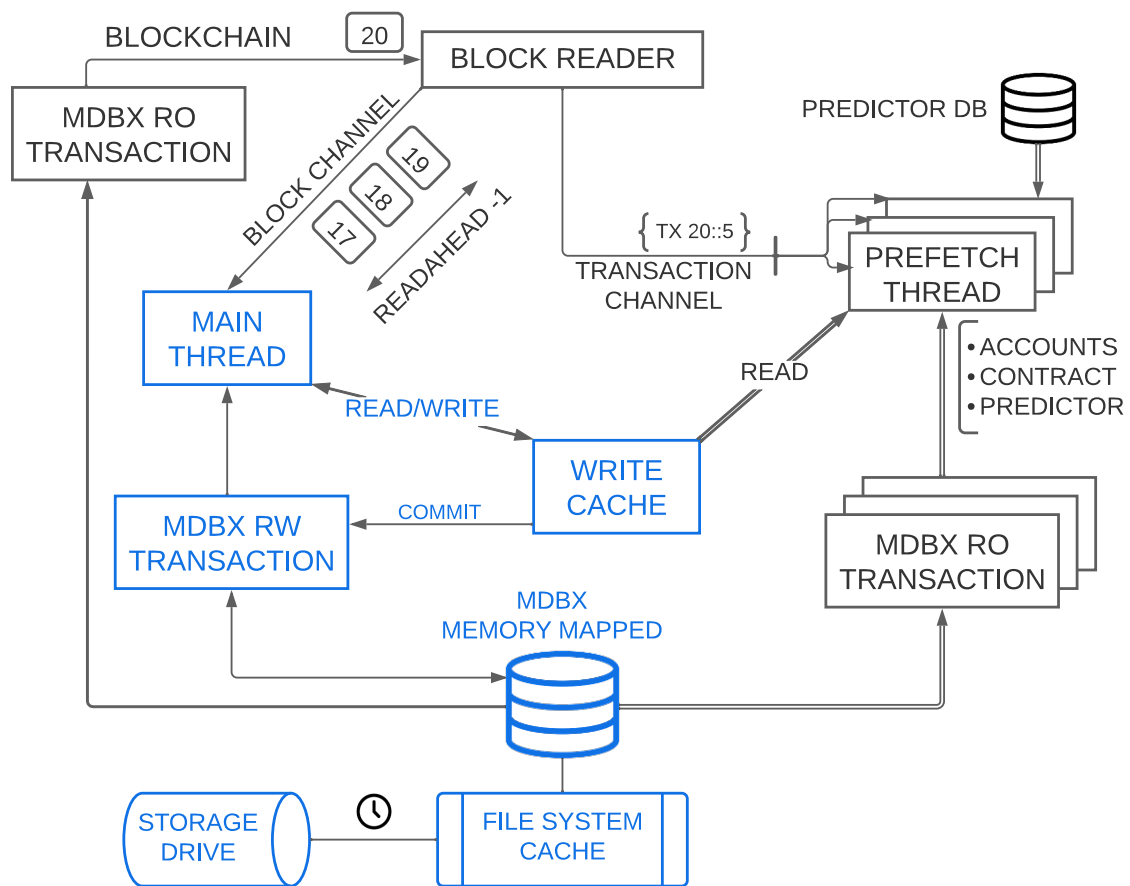


Figure 5.9: Σχηματική αναπαράσταση του υποσυστήματος προανάκτησης. Τα μπλε κουτιά αντιπροσωπεύουν ήδη υπάρχουσες δομές, ενώ τα γκρι είναι αυτά που προστέθηκαν.

Λαμβάνοντας transaction από το νήμα προανάγνωσης block, τα νήματα προανάκτησης έχουν ως στόχο να διαβάσουν δεδομένα τα οποία θα είναι χρήσιμα για την πραγματική εκτέλεση του transaction, ? ahead of time ? ώστε, όταν το κύριο νήμα τα χρειαστεί, να έχουν ήδη διαβαστεί από το μέσο αποθήκευσης στο οποίο αποθηκεύεται η βάση δεδομένων του *client*. Τα ζεύγη κλειδιού-τιμής (key-value pair) που διαβάζονται αναγκάζουν το σύστημα αρχείων να φέρει την σελίδα όπου είναι αποθηκευμένα, στην cache του στην κύρια μνήμη (*file system cache*), με αποτέλεσμα μεταγενέστερες αναγνώσεις των ίδιων κλειδίων³, να διεκπερεώνονται ταχύτερα, χωρίς να χρειάζεται η μεγαλύτερης αναμονής (*higher latency*) ανάγνωση από το αποθηκευτικό μέσο. Αυτός ακριβώς είναι ο σκοπός της εργασίας για να επιταχύνει την εκτέλεση του κύριου νήματος.

5.3.2 Νήματα προανάκτησης

Το καθένα από τα νήματα προανάκτησης έχει μια δικιά του read-write cache τύπου *Intra-BlockState*, όπως και το κύριο νήμα, με μία σημαντική διαφορά. Στο κύριο νήμα τα δεδομένα είναι τελικά, δηλαδή αντιπροσωπεύουν την πραγματική τιμή των κλειδίων που κρατάνε και, εφόσον το block είναι έγκυρο, οι εγγραφές θα καταλήξουν στη βάση δεδομένων⁴ με την ολοκλήρωση του block. Σε αντίθεση, τα δεδομένα στην cache των νημάτων προανάκτησης είναι υποθετικά και προσωρινά, επειδή αφενός είναι βασισμένα σε ? stale ? τιμές από αναγνώσεις που έχουν γίνει εκτός της σειράς εκτέλεσης του κυρίου νήματος και αφετέρου παράγονται από τους predictors που δεν εγγυόνται για την εγγυρότητά τους. Επίσης, κατά την ανάγνωση ενός transaction που προέρχεται από νέο block σε σχέση με αυτό του προηγούμενου transaction, η cache των νημάτων αυτών απορρίπτεται ώστε οι επόμενες αναγνώσεις να ξαναανακτήσουν τα κλειδιά από τη βάση και να κρατήσουν τις νεότερες τιμές τους, που πιθανώς έχουν τροποποιηθεί από το κύριο νήμα.

Τα νήματα προανάκτησης ασχολούνται με τρεις τύπους ανάκτησης, οι οποίοι μπορεί να μην είναι εφικτοί για όλα τα transaction: - ανάγνωση των λογαριασμών του αποστολέα (*from*) και παραλήπτη (*to*), εφόσον υπάρχει - ανάγνωση του κώδικα του παραλήπτη, εφόσον ο λογαριασμός δείχνει ότι είναι smart contract - εκτέλεση του predictor που αντιστοιχεί στο code hash του παραλήπτη, εφόσον υπάρχει, με τα δεδομένα του transaction ως είσοδο

Κατά την εκτέλεση των predictor, κάποια δεδομένα που διαβάζουν είναι άκυρα (πχ εκτός ορίων μνήμης) ή δεν είναι διαθέσιμα (πχ αποτέλεσμα κλήσης που δεν έχει predictor) ή θέλουμε να προσποιηθούμε ότι δεν είναι (πχ BALANCE). Σε αυτές τις περιπτώσεις, η τιμή θεωρείται άγνωστη. Τελεστές που ενεργούν πάνω σε άγνωστη τιμή παράγουν επίσης άγνωστη τιμή. Αν ο έλεγχος φτάσει σε εντολή διακλάδωσης με συνθήκη μια άγνωστη τιμή, τότε επιλέγεται αυθαίρετα η απόφαση βάσει του προορισμού της διακλάδωσης. Αν ο προορισμός είναι έγκυρος και δεν οδηγεί άμεσα σε τερματισμό, τότε ακολουθήται. Ο σκοπός είναι να συνεχίσει να εκτελείται, για να προανακτήσει όσο περισσότερες διευθύνσεις μπορεί. Αυτό ορισμένες φορές οδηγεί σε ατέρμονες βρόγχους (*infinite loop*), για τον οποίο λόγο κάθε εκτέλεση predictor έχει ένα μέγιστο όριο βημάτων, το οποίο όταν υπερβεί διακόπτεται άμεσα. Είναι αντίστοιχο με το όριο GAS της πραγματικής εκτέλεσης και μάλιστα ο υπολογισμός του

³και συγκενικών, που τυχαίνει και βρίσκονται στην ίδια σελίδα

⁴για την ακρίβεια στην write cache (*mutation*), αλλά αυτό δεν επηρεάζει την πρόταση που διατυπώνεται

βασίζεται σε αυτό.

Τόσο το πλήθος των νημάτων προανάκτησης, όσο και οι τύποι ανάκτησης που κάνουν είναι ελεγχόμενοι από σταθερές δηλωμένες πριν την μεταγλώττιση του προγράμματος (compile-time constants). Ομοίως ελεγχόμενη είναι και η ύπαρξη της προανάκτησης εξ' αρχής, με δυνατότητα απενεργοποίησης όλων των βοηθητικών νημάτων για πειραματισμό και προσδιορισμό της επίπτωσης κάθε κομματιού της προανάκτησης στην τελική επίδοση του συστήματος.

Κανάλι των block

Ελεγχόμενη είναι και η χωρητικότητα του καναλιού των block η οποία καθορίζει και τον μέγιστο αριθμό των block που μπορούν να υπάρχουν μεταξύ των βοηθητικών νημάτων και του κυρίως νήματος. Πιο αναλυτικά, αφότου το νήμα προανάγνωσης block διαβάσει ένα block και στείλει τα transaction που περιέχει στα νήματα προανάκτησης, προσπαθεί να στείλει αυτό το block στο κύριο νήμα. Για να μπορέσει να το στείλει πρέπει το πλήθος των block που περιέχει το κανάλι να είναι μικρότερο της χωρητικότητάς του. Διαφορετικά, περιμένει το κύριο νήμα να διαβάσει ένα block από το κανάλι για να δημιουργηθεί χώρος. Η εγγραφή στο κανάλι δηλαδή γίνεται με blocking τρόπο. Αυτό έχει σαν αποτέλεσμα το κομμάτι της προανάκτησης να μην μπορεί να αποκλίνει από το κύριο νήμα πέραν κάποιου προκαθορισμένου πλήθους block. Η συμπεριφορά αυτή είναι από το σχεδιασμό του συστήματος μιας και τα δεδομένα στα οποία βασίζονται οι predictors στα νήματα προανάκτησης είναι αυτά που έχει παράξει το κύριο νήμα, επαυξημένα με τις τροποποιήσεις που έχει κάνει το νήμα προανάκτησης στη διάρκεια ενός μόνο block. Επομένως, οι τροποποιήσεις που πρόκειται να γίνουν στα ενδιάμεσα block δεν είναι διαθέσιμες, γεγονός που μειώνει την ακρίβεια του *World State* το οποίο "βλέπουν" οι predictors. Αυτή η συμπεριφορά των blocking εγγραφών σε κανάλια από το νήμα προανάγνωσης εφαρμόζεται και στην περίπτωση του καναλιού των transaction (*txChan*), το οποίο δημιουργεί την περίπτωση το νήμα προανάγνωσης να περιμένει τα νήματα προανάκτησης πριν προχωρήσει στην ανάγνωση του επόμενου block, ενώ το κύριο νήμα έχει ολοκληρώσει όλα τα διαθέσιμα block. Αυτό προφανώς περιορίζει την ταχύτητα εκτέλεσης του συστήματος και πρέπει να αποφεύγεται. Τέτοιες περιπτώσεις όμως είναι πιο πιθανές όταν η χωρητικότητα του καναλιού των block είναι μικρότερη, καθώς μια μεγαλύτερη χωρητικότητα απορροφά πιο εύκολα τις διακυμάνσεις στο χρόνο εκτέλεσης της προανάκτησης. Έτσι δημιουργείται ένα tradeoff μεταξύ μικρότερης χωρητικότητας με πιο κέραια δεδομένα στους predictors έναντι μεγαλύτερης χωρητικότητας με μικρότερη πιθανότητα περιορισμού του κυρίως νήματος, το οποίο θα εξεταστεί με δοκιμές σε επόμενο κεφάλαιο. Σε αυτό το θέμα πρέπει να διευκρινιστεί το μέτρο της μέγιστης διαφοράς (σε block) μεταξύ της θέσης προανάκτησης και του κυρίως νήματος, την οποία ονομάζουμε *readahead* (RA). Η χωρητικότητα του καναλιού τίθεται ως $RA - 1$ για $RA > 0$ ενώ είναι ίση με 0 για $RA = 0$, μιας και εκτός από τα block μέσα στο κανάλι υπάρχει και ένα ακόμα το οποίο βρήσκειται στο νήμα προανάγνωσης. Υπάρχει όμως μία ακόμα διαφορά μεταξύ $RA = 0$ και $RA = 1$, παρότι και στις δύο περιπτώσεις η χωρητικότητα του καναλιού είναι 0. Στην περίπτωση που $RA = 0$, το νήμα προανάγνωσης περιμένει να δώσει το block στο κύριο thread πριν δώσει κάποιο transaction στα κανάλια προανάκτησης, ενώ στην περίπτωση που $RA \geq 1$, το νήμα

προανάγνωσης προσπαθεί αρχικά να δώσει το block χωρίς να περιμένει (non-blocking), στη συνέχεια δίνει τα transaction και στο τέλος αν δεν είχε δώσει το block περιμένει να το δώσει. Ας σημειωθεί ότι η περίπτωση $RA = 0$ διαφέρει επίσης από την περίπτωση απενεργοποίησης συνολικά της προανάγνωσης, μιας και στην πρώτη η ανάγνωση του block γίνεται από το βοηθητικό νήμα προανάγνωσης και όχι από το κύριο, δημιουργώντας ένα σχήμα pipeline, ενώ στη δεύτερη η ανάγνωση γίνεται μόνο από το κύριο νήμα. Η διαφορές αυτές αποτυπώνονται και στα σχήματα 5.10, 5.11, 5.12 και 5.13.

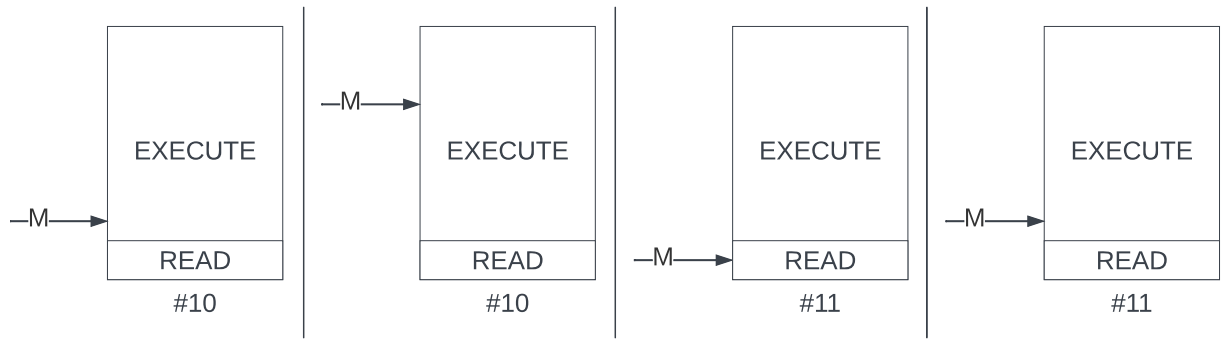


Figure 5.10: .

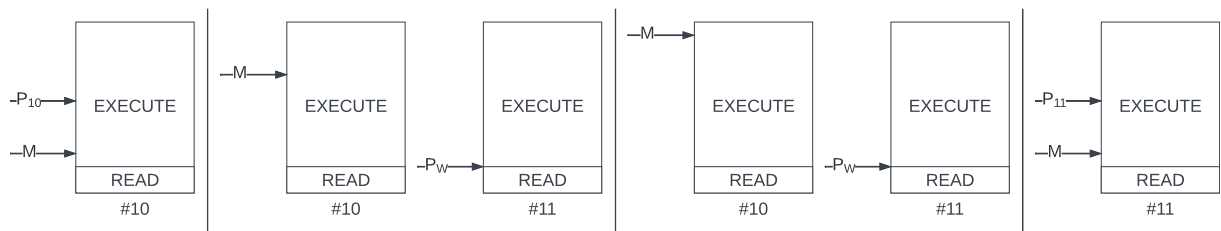


Figure 5.11: .

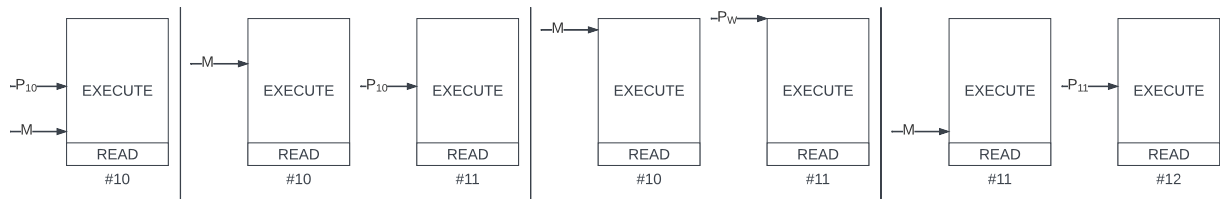


Figure 5.12: .

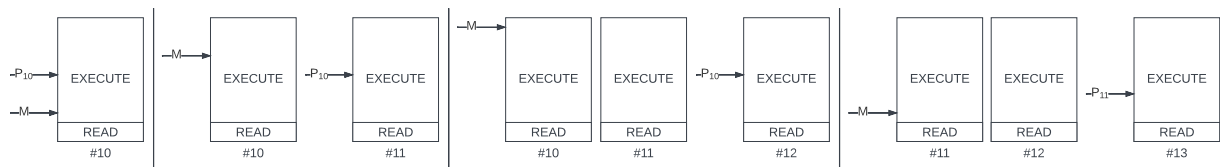


Figure 5.13: .

Κανάλι των transaction

Το κανάλι των transaction έχει χωρητικότητα 0, και οι αναγνώσεις και εγγραφές σε αυτό γίνονται με blocking τρόπο. Εγγραφές γίνονται μόνο από το ένα νήμα προανάγνωσης ενώ

αναγνώσεις από όλα τα νήματα προανάκτησης, δηλαδή είναι κοινό για τα νήματα αυτά. Η κατανομή των transaction στα νήματα γίνεται με απροσδιόριστο⁵ τρόπο μεταξύ αυτών που είναι διαθέσιμα για να δεχτούν το transaction κάθε φορά. Έτσι, το φορτίο της προανάκτησης ισοκατανέμεται χρονικά στα αρμόδια νήματα.

Κοινή πρόσβαση στη βάση και write cache

Όπως έχει προαναφερθεί, ο *erigon client* χρησιμοποιεί μια write-back write-cache για όλες τις εγγραφές στη βάση (*mutation*), η οποία καταχωρήται στη βάση και αδειάζει μετά από αρκετές χιλιάδες block. Επομένως οι αναγνώσεις από τη βάση, για να έχουν τις πιο πρόσφατες τιμές, πρέπει να περάσουν πρώτα από την write-cache. Η υλοποίησή της είναι εντός κύριας μνήμης (*in-memory*) και είναι βασισμένη σε δομή β-δέντρου (*b-tree*). Η δομή αυτή δεν έχει έλεγχο για ταυτόχρονη πρόσβαση, οπότε για να μπορεί να χρησιμοποιηθεί και από τα βοηθητικά νήματα, γίνεται μια μικρή τροποποίηση για προσθήκη ενός *spinlock* που επιτρέπει ταυτόχρονη πρόσβαση από πολλούς αναγνώστες (*reader*, *Get()*) ή αποκλειστικά πρόσβαση από έναν τροποποιητή (*writer*, *Put()*). Το κλειδωμα με *spinlock* επιλέχθηκε, μιας και οι αναζητήσεις στην write-cache διεκπερεώνονται πολύ σύντομα, γίνονται πολύ συχνά και οι συγκρούσεις (ταυτόχρονη πρόσβαση από *reader* και *writer*) είναι σπάνιες (ο χρόνος αναζήτησης στη write-cache είναι πολύ μικρό ποσοστό του συνολικού χρόνου).

Όταν το κλειδί που αναζητείται δεν είναι στη write-cache, η αναζήτηση συνεχίζεται στη βάση δεδομένων. Κάθε νήμα έχει ένα δικό του (*database*) transaction, με αυτό που κρατάται από το κύριο νήμα να μπορεί να κάνει αναγνώσεις και εγγραφές, ενώ αυτά που ανήκουν στα βοηθητικά νήματα να μπορούν να κάνουν μόνο αναγνώσεις. Λόγω της write-back φύσης της write-cache, ακόμα και το κύριο νήμα χρησιμοποιεί το *database transaction* μόνο για αναγνώσεις, μέχρι τη στιγμή που χρειάζεται να καταχωρήσει την write-cache όταν το μέγεθός της υπερβεί ένα εκ των προτέρων προσδιορισμένο κατώφλι.

Ο *erigon* χρησιμοποιεί ως βάση δεδομένων την MDBX που αποτελεί τροποποίηση της LMDB [17]. Η LMDB [18] λειτουργεί εξ' ολόκληρου με αντιστοίχιση του αρχείου της από το δίσκο στην κύρια μνήμη (κλήση συστήματος *mmap*). Δεν περιέχει κάποιο επίπεδο cache, αντιθέτως χρησιμοποιείται η ήδη υπάρχουσα cache του συστήματος αρχείων (*page cache*) την οποία διαχειρίζεται το λειτουργικό σύστημα. Επομένως, αναγνώσεις από ένα νήμα (*database transaction*) προκαλούν αυτόματα αλλαγές στην *page cache* και επιρρεάζουν (χρονικά) τις αναγνώσεις των υπολοίπων. Όσο για τον έλεγχο πρόσβασης, χρησιμοποιεί την μέθοδο MVCC (*multiversion concurrency control*), επιτρέποντας ταυτόχρονη πρόσβαση από πολλούς αναγνώστες και έναν τροποποιητή (*writer*) χωρίς κλειδώματα (*lock-free*). Επομένως δεν χρειάζεται κάποια τροποποίηση για την υποστήριξη των βοηθητικών νημάτων.

⁵οι προδιαγραφές της γλώσσας *go* δεν αναφέρουν τι γίνεται στην περίπτωση που δύο ή περισσότεροι αναγνώστες αναμένουν να διαβάσουν από το ίδιο κανάλι [GOCHAN], αλλά αυτό δεν επιρρεάζει την ανάλυσή που περιγράφεται

Κεφάλαιο 6

Περιγραφή Benchmarking

Στο κεφάλαιο αυτό θα γίνει αναλυτική περιγραφή του εξοπλισμού και λογισμικού που θα χρησιμοποιηθεί για την μέτρηση της επίδοσης και αξιολόγηση του συστήματος, καθώς και τις μεθόδους που θα ακολουθηθούν και παραμέτρους που θα εξεταστούν για την διεκπεραίωση των πειραμάτων και εξαγωγή των μετρήσεων.

6.1 Εξοπλισμός

Οι μετρήσεις που πρόκειται να γίνουν μετρώνται εν μέρη σε χρόνο εκτέλεσης και εξαρτώνται σε μεγάλο βαθμό από το υλισμικό (hardware) του συστήματος όπου εκτελούνται, γεγονός που θα επιβεβαιωθεί και στα αποτελέσματά τους, στο επόμενο κεφάλαιο. Ακολουθεί η περιγραφή του hardware που χρησιμοποιήθηκε, με εξέταση των επιδόσεων (benchmarking) των σημαντικότερων κομματιών (components).

6.1.1 Κύρια κομμάτια του υπολογιστή

Επεξεργαστής

Ο επεξεργαστής είναι το μοντέλο Ryzen 9 3900X της AMD. Οι 12 φυσικοί πυρήνες του είναι οργανωμένοι σε 2 διαφορετικά τμήματα ημιαγωγού (semiconductor dies), καθένα εκ των οποίων περιέχει 2 ομάδες πυρήνων (core complex - CCX) που μοιράζονται την ίδια L3 cache μεγέθους 16 MiB. Υποστηρίζει ταυτόχρονη πολυνηματική εκτέλεση (simultaneous multithreading - SMT) με 2 νήματα ανά πυρήνα (2-way SMT), η οποία είναι ενεργοποιημένη σε όλες τις δοκιμές. Το λειτουργικό σύστημα βλέπει 2 πυρήνες για κάθε ένα φυσικό πυρήνα και σε κάθε ζεύγος οι πυρήνες του θα λέγονται sibling cores. Κάθε CCX επομένως έχει 3 φυσικούς πυρήνες και φαίνεται σαν να έχει 6. Στο σχήμα 6.1 φαίνεται ο χρόνος απόκρισης μιας κατεύθυνσης (one-way latency) της επικοινωνίας μεταξύ όλων των πυρήνων:

Είναι προφανής η οργάνωση των πυρήνων σε ομάδες CCX καθώς και η αντιστοιχία των sibling cores. Θα επιλέξουμε να περιορίσουμε όλες τις δοκιμές στους πυρήνες 3, 4 και 5 που ανήκουν στο ίδιο CCX (CCX1), μαζί με τους sibling πυρήνες τους 15, 16, και 17 αντίστοιχα (σχήμα 6.2).

Το κύριο νήμα εκτέλεσης θα είναι περιορισμένο στον πυρήνα 3, ενώ τα υπόλοιπα στους 4, 5, 16, 17. Ο sibling πυρήνας του 3 (ο 15) θα παραμένει κενός για να μην επιρρεάζεται η εκτέλεση του κύριου νήματος που βρίσκεται στον ίδιο φυσικό πυρήνα. Ο περιορισμός των νημάτων στους πυρήνες στην γλώσσα go απαιτεί και το "κλείδωμα" των go-routines

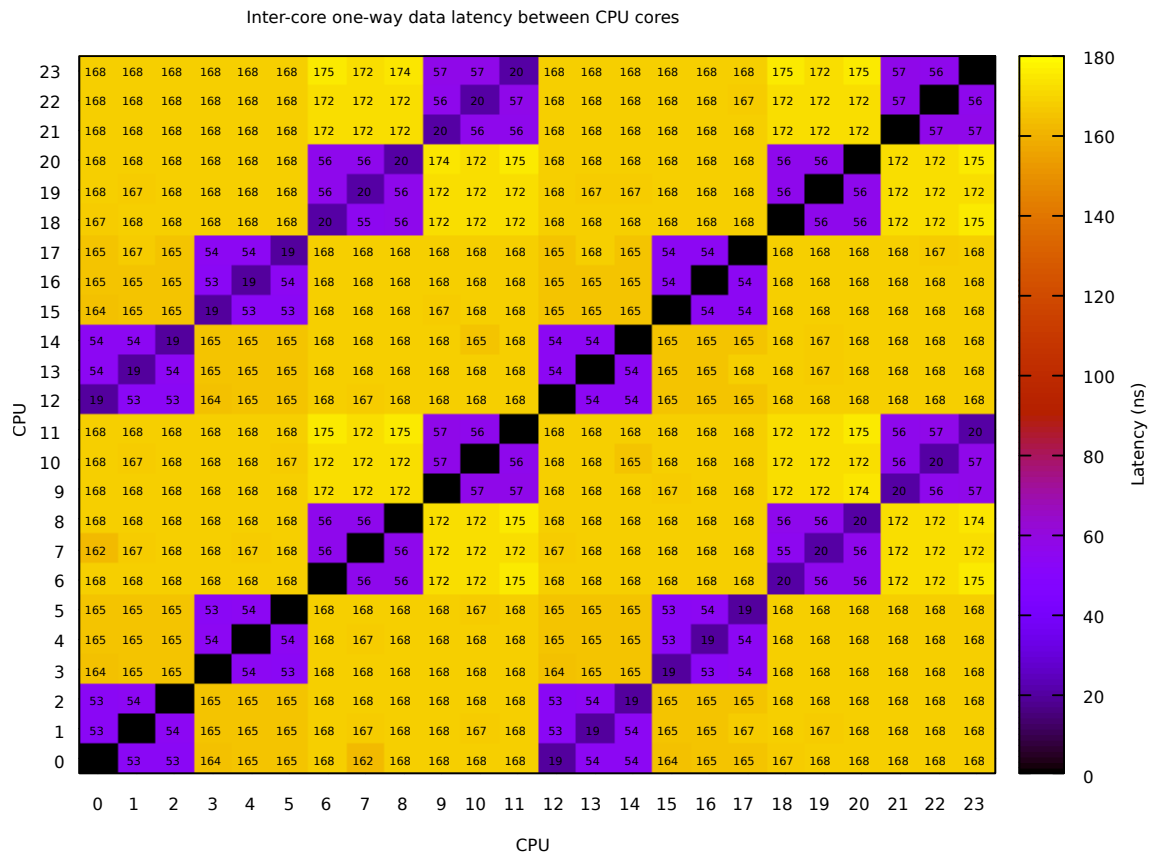


Figure 6.1: .

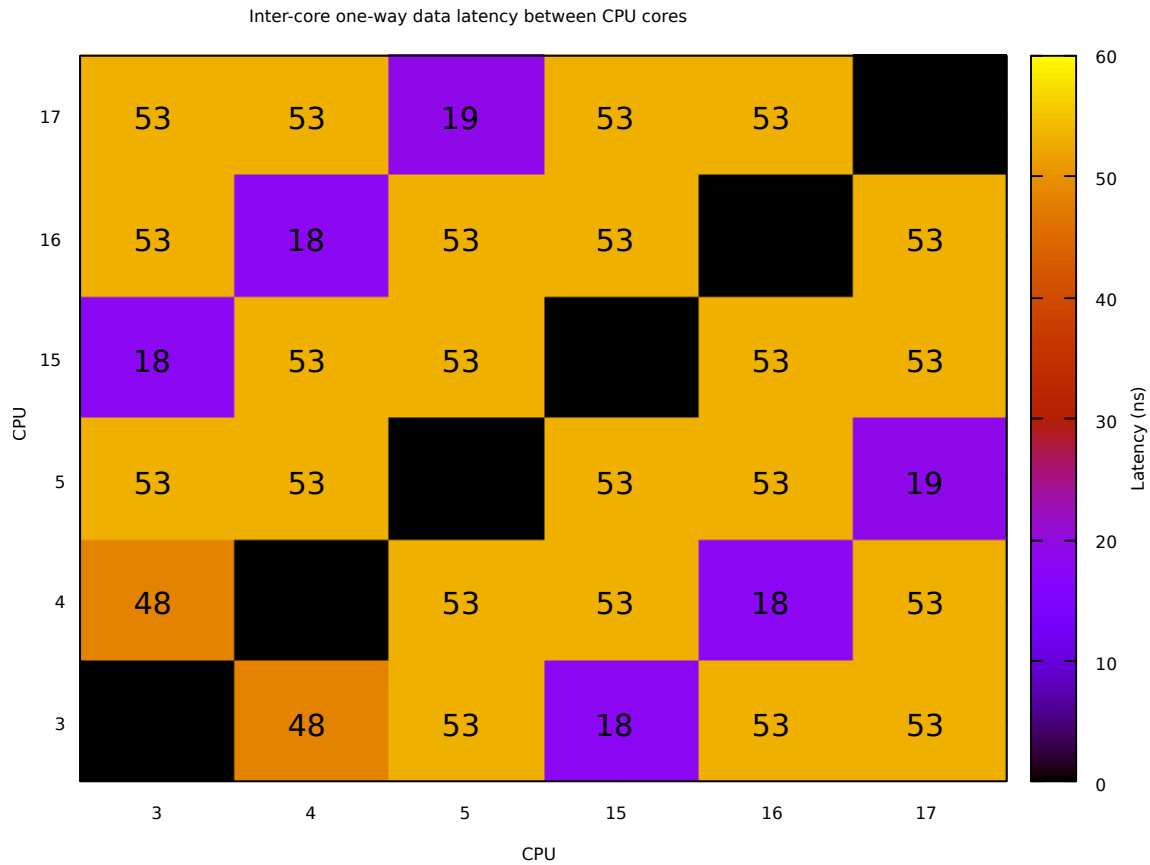


Figure 6.2: .

που περιέχουν τον κώδικα των νημάτων που αναφέρθηκαν, σε μοναδικά δικά τους νήματα λειτουργικού (OS threads), διαφορετικά ο scheduler της go θα ήταν ελεύθερος να αλλάξει τη θέση τους [23]. Ο περιορισμός των OS thread σε πυρήνες γίνεται με την κλήση συστήματος (system call) `sched_setaffinity` που είναι διαθέσιμη μέσω βιβλιοθήκης της go [24].

```
func setCPU(cores ...int) {
    t := unix.CPUSet{}
    for _, core := range cores {
        t.Set(core)
    }
    unix.SchedSetaffinity(0, &t)
}

func lockThreadAndCPU(cores ...int) {
    runtime.LockOSThread() // Also needed for unique PID/TID, for perf-utils
    setCPU(cores...)
}
```

Ο garbage collector της go, μπορεί να χρησιμοποιήσει οποιουσδήποτε από τους 6 πυρήνες. Σε κάθε περίπτωση, ολόκληρη η διεργασία του `erigon` (όλα τα νήματα) είναι περιορισμένη εξ' αρχής στους 6 αυτούς πυρήνες με τη χρήση του προγράμματος `taskset` [25]:

```
taskset -c 3-5,16-17 ./build/bin/erigon ....
```

Κύρια μνήμη (RAM)

Η κύρια μνήμη αποτελείται από 2 DIMM DDR4 32 GiB το καθένα (μοντέλο TED432G3200C2201 της TEAM GROUP), με ονομαστική συχνότητα ρολογιού 1600 MHz (3200 MT/s) και χρονισμούς κατά JEDEC DDR4-3200 (22-22-22-52). Στις δοκιμές θα γίνουν αλλαγές τόσο στη συχνότητα ρολογιού όσο και στους χρονισμούς για τη διαπίστωση πιθανής επίπτωσης της ταχύτητας μεταφοράς (memory bandwidth) και του χρόνου απόκρισης (memory latency). Επίσης, δεν θα είναι διαθέσιμη ολόκληρη η μνήμη, αλλά αντιθέτως θα περιορίζεται η διαθέσιμη μνήμη πριν την εκτέλεση του `erigon` σε διάφορα επίπεδα. Για τον περιορισμό της διαθέσιμης μνήμης, ξεκινάει πρώτα μία άλλη διεργασία που δεσμεύει όσο ποσό μνήμης χρειάζεται ώστε αυτή που υπολοιπείται να ισούται με το ποσό που ορίζει η κάθε δοκιμή. Ο κώδικας αυτής της διεργασίας είναι σε Python και βρίσκεται σε παράρτημα της εργασίας. Η λειτουργία επέκτασης της φυσικής μνήμης χρησιμοποιώντας το δίσκο (swap) είναι απενεργοποιημένη.

Μέσα αποθήκευσης

Το λειτουργικό σύστημα, ο κώδικας του συστήματος και το εκτελέσιμο βρίσκονται όλα σε μία μονάδα SSD που δεν χρησιμοποιείται για κάποιον άλλο σκοπό. Η βάση δεδομένων που περιλαμβάνει το *World State* καθώς και η βάση με τους *peridictors* βρίσκονται σε μία άλλη μονάδα SSD, που υπάρχει αποκλειστικά για την αποθήκευση αυτών των βάσεων. Ο `erigon client` κατά την εκτέλεσή του χρησιμοποιεί μόνο αυτές τις δύο βάσεις και επομένως μόνο στο μέσο αποθήκευση όπου βρίσκονται. Ένα αντίγραφο και των δύο βάσεων βρίσκεται σε μία δεύτερη μονάδα SSD, διαφορετικών προδιαγραφών.

Η πρώτη μονάδα έχει διεπαφή PCI-express (πρωτόκολλο NVME) ενώ η δεύτερη έχει SATA, και στο εξής θα αναφέρονται απλώς ως `nvme` και `sata`.

- Ο nvme είναι το μοντέλο SX8200 Pro της Adata, χωρητικότητας 2TB και είναι η έκδοση με τον ελεγκτή SM2262EN της Silicon Motion, μνήμη flash τύπου TLC (triple-level cells), διεπαφή PCI-Express v3 με 4 lanes, και είναι συνδεδεμένος στο PCIe bus του επεξεργαστή.
- Ο sata είναι το μοντέλο P210S2TB25 της Patriot, χωρητικότητας επίσης 2TB, με μνήμη flash τύπου QLC (quad-level cell), διεπαφή SATA III και είναι συνδεδεμένος στο chipset X570 της μητρικής στο οποίο συνδέεται ο επεξεργαστής με διεπαφή PCIe v4 x4.

Στο σχήμα 6.3 αποτυπώνεται ο χρόνος απόκρισής τους σε αναγνώσεις, από δοκιμή με το πρόγραμμα fio με 1 thread και blocking, 4KiB random read I/O. Είναι προφανές ότι ο χρόνος απόκρισής του sata είναι σημαντικά μεγαλύτερος από τον προηγούμενο, για τον οποίο λόγο επιλέχθηκε και γίνονται δοκιμές και στους δύο.

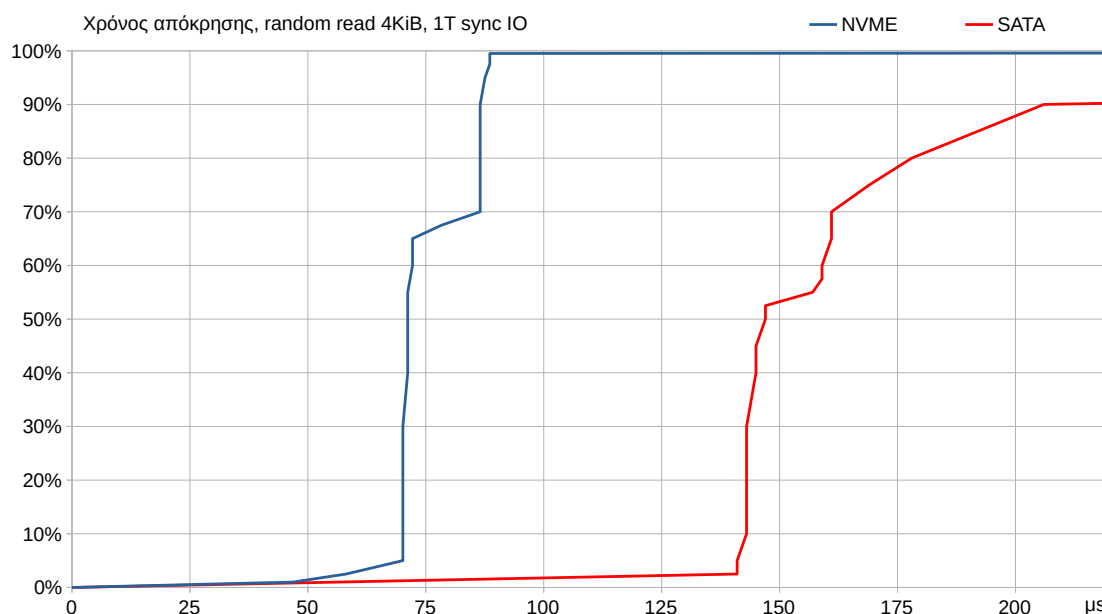


Figure 6.3: Κατανομή χρόνου απόκρισης των δύο δίσκων. Ο nvme έχει περίπου το μισό από τον sata.

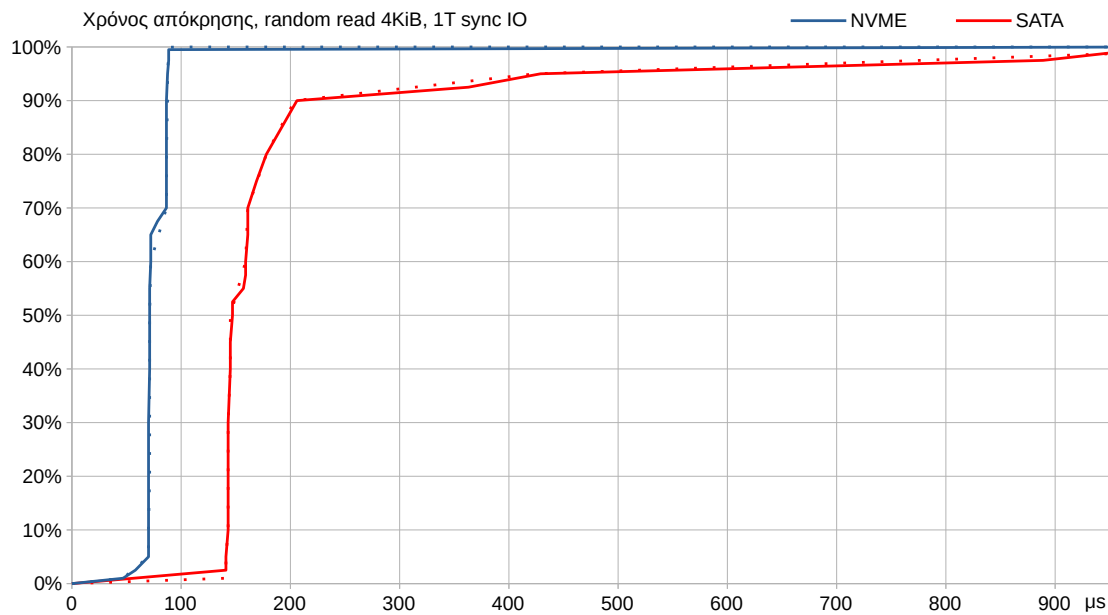


Figure 6.4: Σε αντίθεση με του nvme, η κατανομή του sata έχει «heavy tail»

6.2 Μέθοδος δοκιμών

Οι δοκιμές γίνονται σε Ubuntu server 21.04 με kernel Linux 5.11.0-49-generic. Πέραν του λειτουργιού και του SSH daemon, δεν τρέχουν άλλες διεργασίες ταυτόχρονα με τις δοκιμές. Πριν κάθε εκτέλεση του erigon, άδειαζε η page cache του συστήματος:

```
echo 3 >/proc/sys/vm/drop_caches
```

Οι παράμετροι του κώδικα του erigon που αλλάζουν σε κάθε δοκιμή είναι compile-time σταθερές, και επομένως κάθε φορά γίνεται ξανά compile. Ύστερα δεσμεύεται κύρια μνήμη ώστε τελικά η διαθέσιμη να ισούται με την επιθυμητή. Τέλος, εκτελείται ο erigon για το προκαθορισμένο πλήθος 30000 block. Οι μετρήσεις χρόνου γίνονται όλες εσωτερικά του, με χρήση του μετρητή TSC του επεξεργαστή [26]:

```
TEXT ·Measure(SB),NOSPLIT,$0-8
```

```
    RDTSCP
```

```
    LFENCE
```

```
    SHLQ    $32, DX
```

```
    ADDQ    DX, AX
```

```
    MOVQ    AX, ret+0(FP)
```

```
    RET
```

```
func Tick(index int) {
    t := int64(Measure())
    atomic.AddInt64(&all_ticks[index], t)
    atomic.AddInt64(&all_counts[index], 1)
}

const TSC_FREQ_100MHZ = 38 // 3.8 GHz
// ...
```

```

return fmt.Sprintf("%14d_\\%14d_\\%14d",
    ((ticksA - ticksB) * 10) / (TSC_FREQ_100MHZ * countsA), // μέσος όρος
    countsA, // πλήθος μετρήσεων
    (ticksA - ticksB) / (TSC_FREQ_100MHZ * 100), // συνολικός χρόνος
)
// ...

```

Μετρήσεις γίνονται πριν και μετά την εκτέλεση ενός block (άρα αναμένεται το πλήθος ζευγών μετρήσεων να είναι 30000), καθώς και σε άλλα σημεία, πχ στις αναγνώσεις από τη βάση. Κρατάται ο συνολικός χρόνος.

6.3 Ακρίβεια μετρήσεων

Για μια απλή εκτίμηση της ακρίβειας των μετρήσεων έγιναν 10 όμοιες δοκιμές με τον δίσκο sata και άλλες τόσες με τον nvme. Η κάθε δοκιμή διαρκούσε περίπου 10 με 15 λεπτά. Στο σχήμα 6.5 φαίνεται το ιστόγραμμα των ταχυτήτων εκτέλεσης.

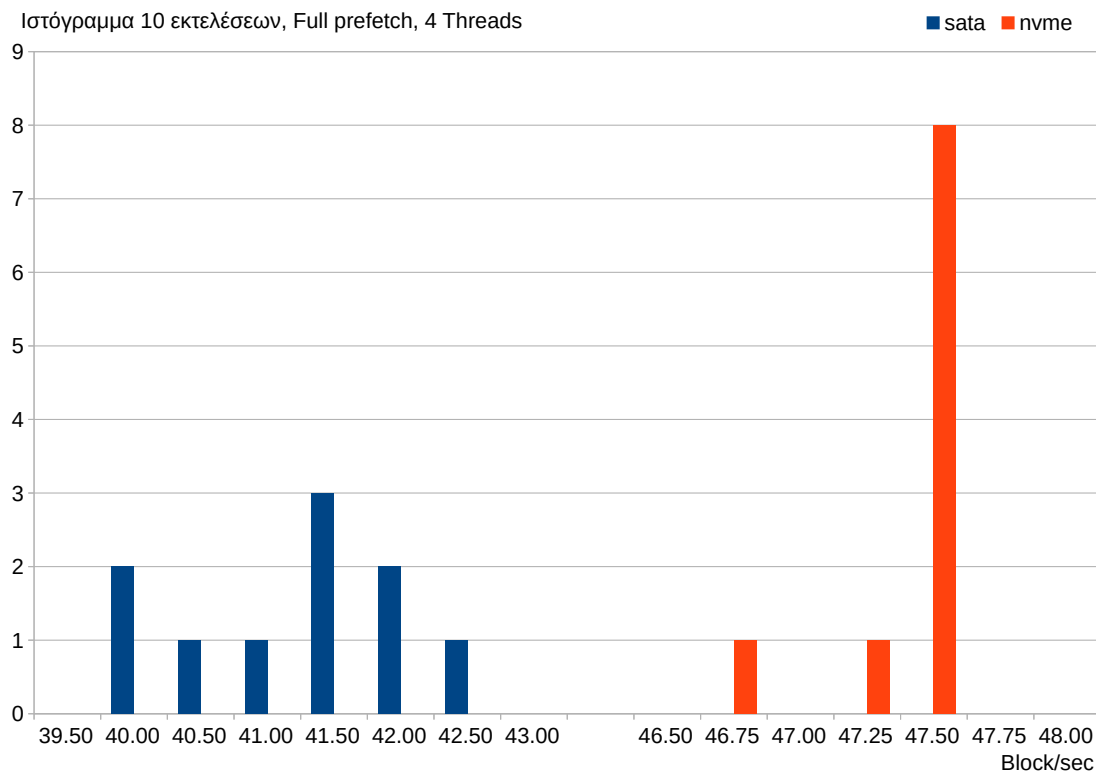


Figure 6.5: Κατανομή της ταχύτητας εκτέλεσης των 10 ίδιων δοκιμών για κάθε δίσκο, με 4 νήματα προανάκτησης.

Εκτιμούμε την τυπική απόκλιση ως: ΤΥΠΟΣ s

- Σχετική τυπική απόκλιση για τον sata: $s \approx 2,2\%$
- Σχετική τυπική απόκλιση για τον nvme: $s \approx 0,5\%$

Η πραγματική τιμή της τυπικής απόκλισης ενδέχεται να είναι μεγαλύτερη από την παραπάνω εκτίμηση.

Αποτελέσματα

7.1 Ποσοστά επιτυχίας predictors

Για την αξιολόγηση των predictor ως προς τις διευθύνσεις τύπου storage τις οποίες προβλέπουν, ορίζουμε τις παρακάτω μετρικές:

- $Coverage = TP/P$,
όπου TP (True Positive) το πλήθος των διευθύνσεων που προβλέπονται και όντως χρησιμοποιούνται στην πραγματική εκτέλεση
και P (actual Positive) το πλήθος των διευθύνσεων που χρησιμοποιούνται στην πραγματική εκτέλεση
- $Overhead = FP/P$,
όπου FP (False Positive) το πλήθος των διευθύνσεων που προβλέπονται αλλά δεν χρησιμοποιούνται στην πραγματική εκτέλεση

Τα πλήθη διευθύνσεων μετρώνται ανά Transaction. Μια διεύθυνση που χρησιμοποιείται 2 φορές στο ίδιο Transaction μετράει ως 1, ενώ αν χρησιμοποιείται 2 φορές σε 2 διαφορετικά Transaction την κάθε φορά μετράει ως 2.

7.1.1 Επίδραση του Readahead και του πλήθους νημάτων προανάκτησης

Coverage

με NVME

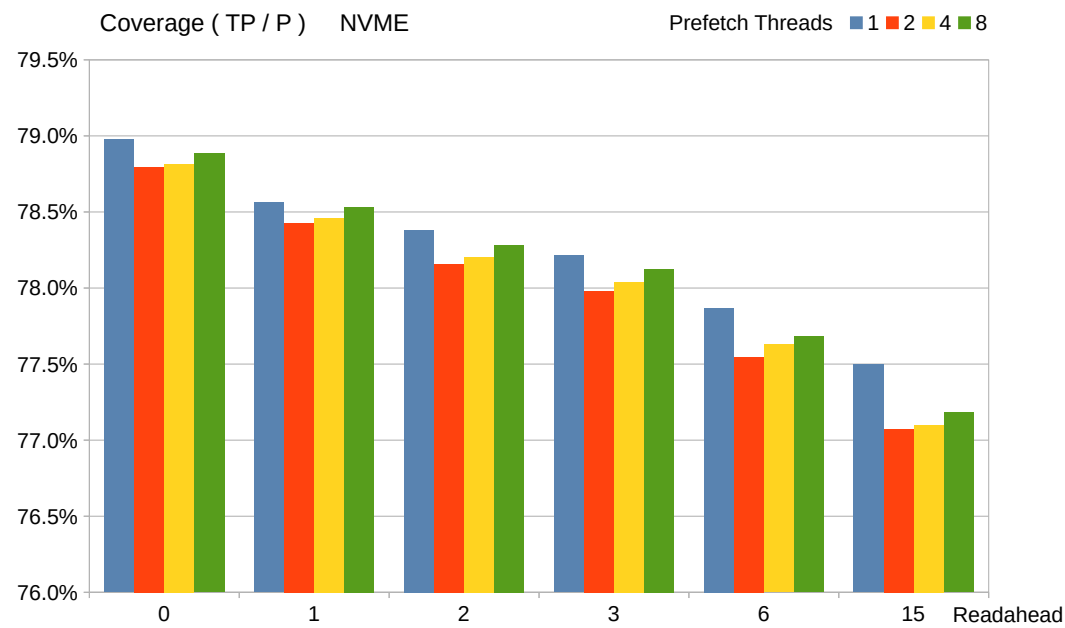


Figure 7.1: .

με SATA

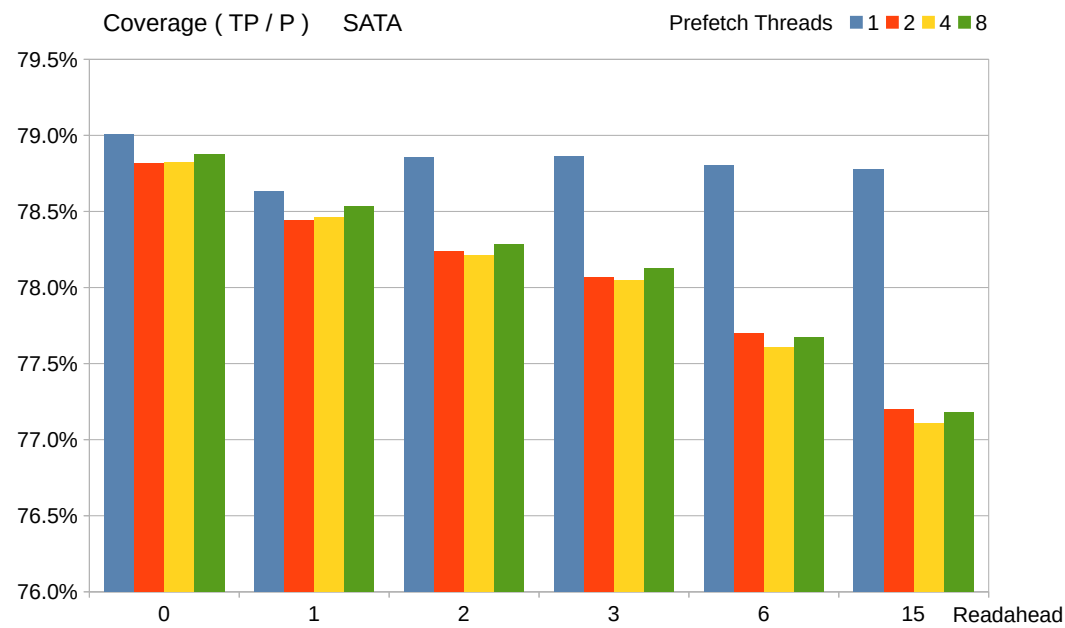


Figure 7.2: .

Overhead

με NVME

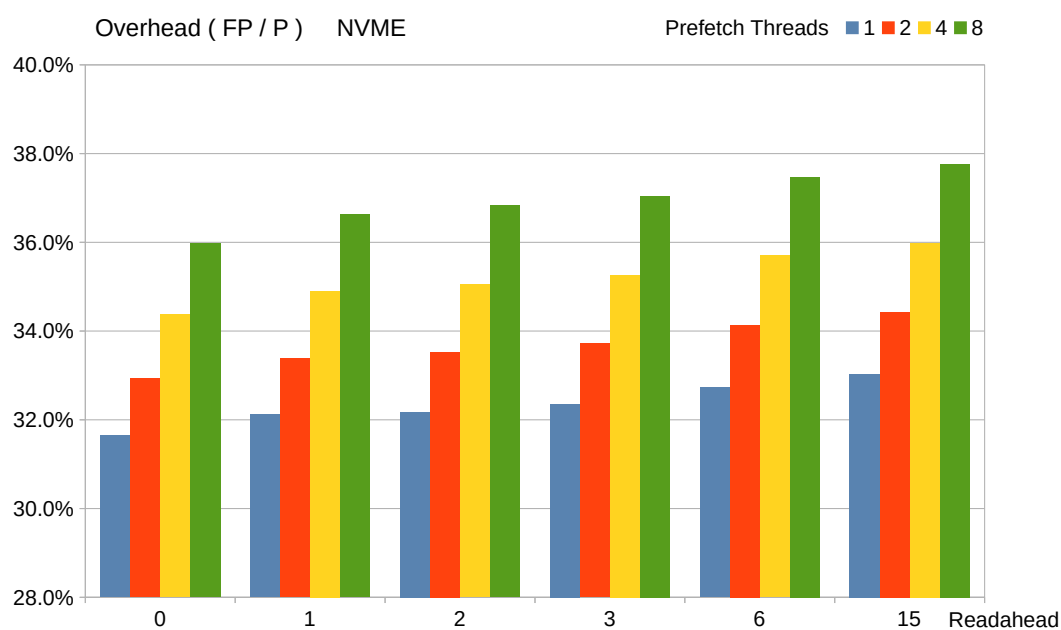


Figure 7.3: .

με SATA

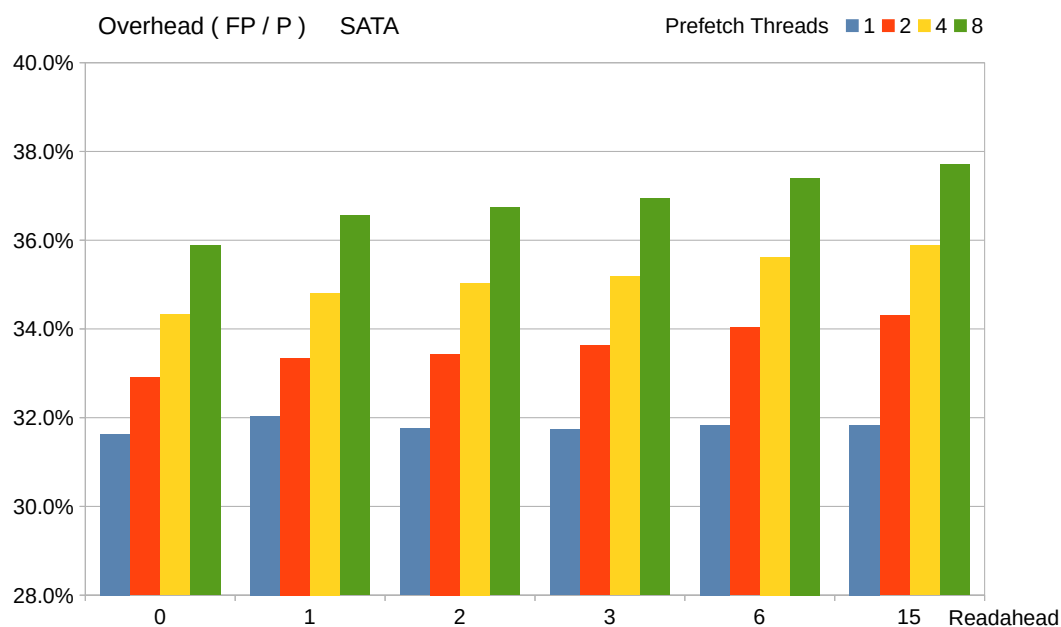


Figure 7.4: .

Μιας και το Readahead είναι ένα όριο (μέγιστο) και όχι επιβαλλόμενο, όταν τα νήματα προανάκτησης αργούν, η πραγματική απόσταση είναι μικρότερη από αυτό, με αποτέλεσμα η αύξησή του να μην επιφέρει κάποια πραγματική αλλαγή.

7.2 Ταχύτητα εκτέλεσης

7.2.1 Speedup σε τυπικές συνθήκες

Προσομοιάζουμε ένα "τυπικό" σύστημα αφήνοντας 16 GiB διαθέσιμη μνήμη, πλήθος πυρήνων όπως περιγράφηκε προηγουμένως, τους χρονισμούς στις αναγραφόμενες τιμές τους, θέτουμε Readahead = 2 και δοκιμάζουμε στους 2 δίσκους, ενεργοποιώντας περισσότερους τύπους προανάκτησης κάθε φορά:

με NVME

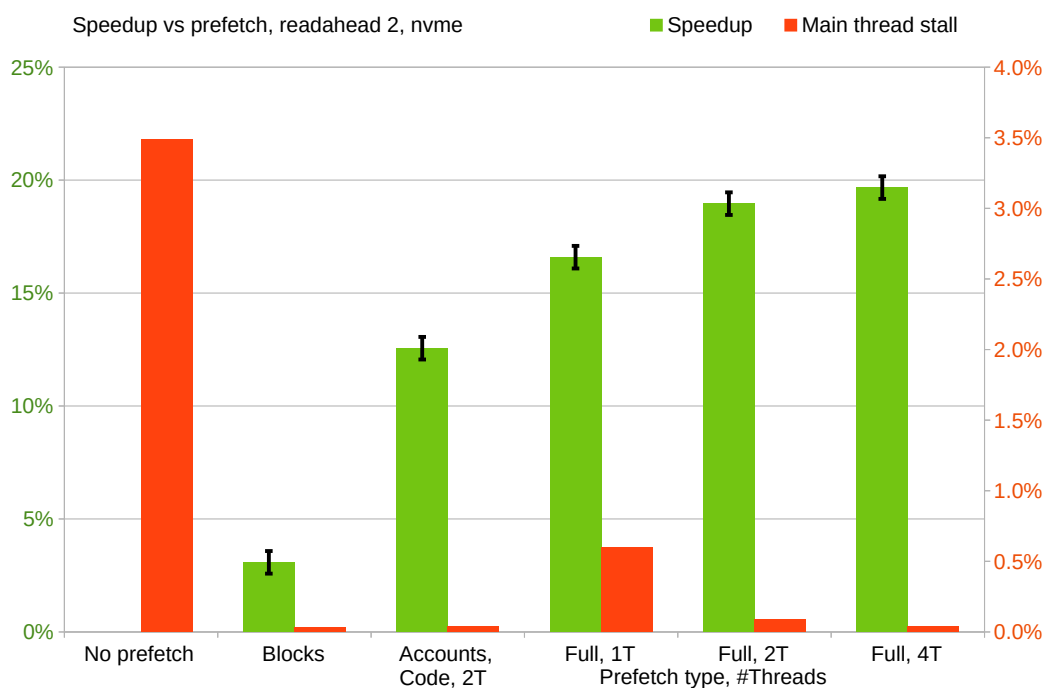


Figure 7.5: .

με SATA

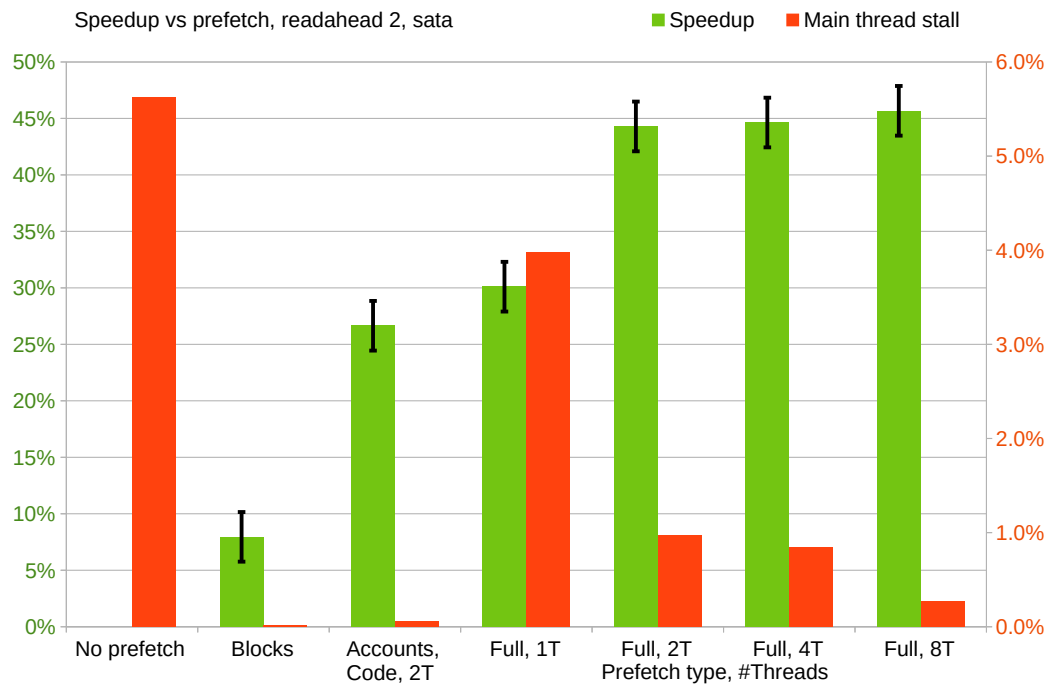


Figure 7.6: .

Εδώ αναγράφεται και το "Main thread stall" δηλαδή ο χρόνος που το κύριο νήμα περιμένει το νήμα προανάγνωσης για να του δώσει το επόμενο block. Αν τα νήματα προανάκτησης που τρέχουν τους predictor αργούν, τότε το κύριο αναγκάζεται να τα περιμένει. Σε περίπτωση που δεν γίνεται καθόλου προανάκτηση, η τιμή αυτή αντιπροσωπεύει το χρόνο ανάγνωσης του block από το ίδιο το κύριο νήμα.

Βλέπουμε ότι και στις δύο περιπτώσεις υπάρχει σημαντική βελτίωση, με μεγαλύτερη σε κάθε περίπτωση στον - πιο αργό - SATA SSD. Ακόμα και η σχετικά απλή προανάγνωση των block και προανάκτηση των λογαριασμών και κώδικα contract αποφέρει ένα μεγάλο μέρος του συνολικού speedup. Αντιθέτως, η αύξηση του αριθμού των νημάτων πέραν των 2 δεν επιφέρει σημαντικές διαφορές. Αυτό φαίνεται και από το "main thread stall" που είναι σχετικά μικρό σε αυτές τις περιπτώσεις.

7.2.2 Επίδραση της κύριας μνήμης

Σε αυτές τις δοκιμές περιορίζεται η διαθέσιμη μνήμη σε διαφορετικά ποσά (8GiB, 12GiB, 16GiB, 32GiB), καθώς και οι χρονισμοί της. Συγκεκριμένα, στις όλες τις δοκιμές όπου δεν αναφέρεται, η συχνότητα ρολογιού της είναι 1600 MHz (ρυθμός μετάδοσής 3200MT/s) και ο χρόνος απόκρισης (CAS latency) 22 κύκλοι (13,75 ns). Γίνεται μια δοκιμή με συχνότητα 1067 MHz (2133 MT/s) και latency 15 κύκλους (14,0 ns). Άλλη μια δοκιμή με συχνότητα 1067 MHz (2133 MT/s) και latency 22 κύκλους (20,5 ns).

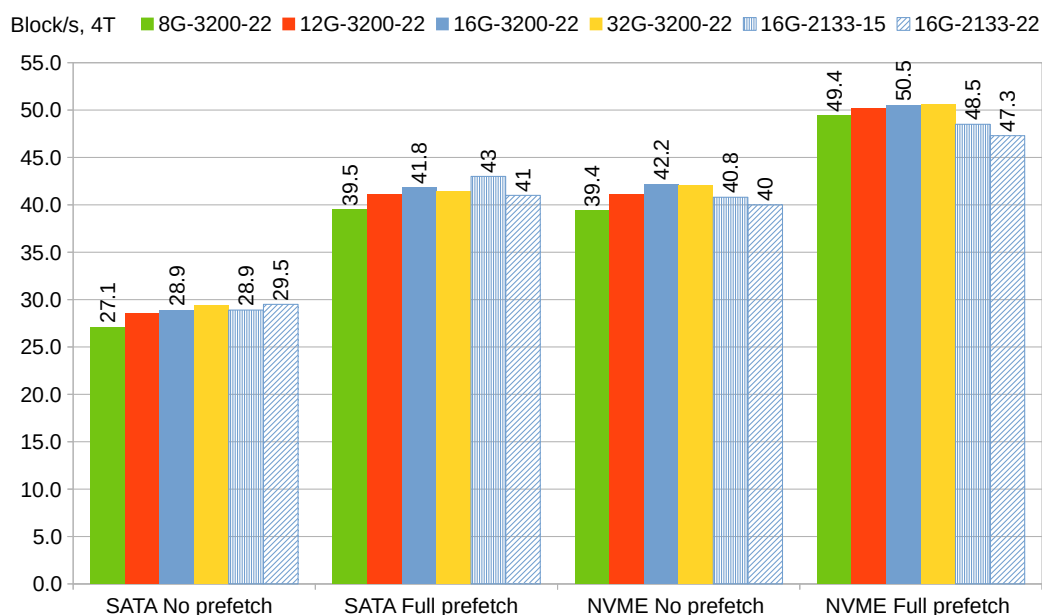


Figure 7.7: .

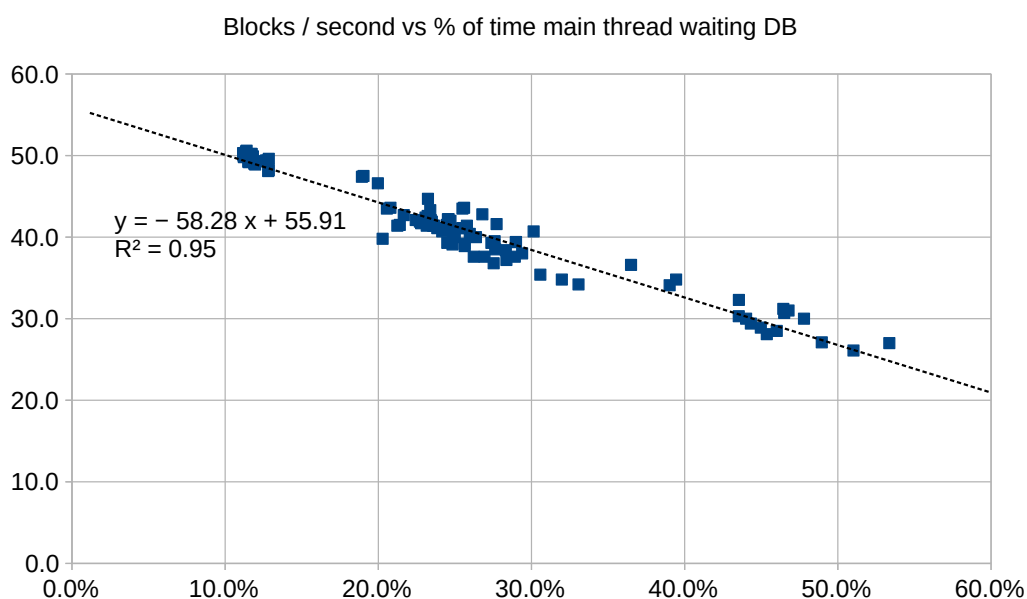
Φαίνεται πως η επίδραση του μεγέθους της μνήμης είναι σχετικά μικρή (έως 7% +/- 1%, "NVME No prefetch") και γίνεται λιγότερο σημαντική με την προανάκτηση ενεργοποιημένη (2% +/- 1%, "NVME Full prefetch"), ιδιαίτερα στην περίπτωση του nvme. Οι χρονισμοί της δεν φαίνεται να επιρρεάζουν στην περίπτωση του sata (δεδομένου και του σφάλματος μέτρησης), ενώ έχουν μετρήσιμη διαφορά (3% έως 7% +/- 1%) με τον nvme.

7.2.3 Επίδραση ταχύτητας CPU (TODO verify)

7.2.4 Ταχύτητα εκτέλεσης | Ποσοστό χρόνου στη βάση δεδομένων

Σε όλες τις δοκιμές¹ έχει συλλεχθεί και ο χρόνος που ξοδεύει το κύριο νήμα όσο βρίσκεται σε κώδικα της βάσης δεδομένων (MDBX). Είναι ο χρόνος ανάγνωσης δεδομένων που δεν υπάρχουν στην write-cache. Μπορούμε να ? plotarουμε ? όλες τις δοκιμές σε ένα γράφημα σύγκρισης του χρόνου αυτού με την ταχύτητα εκτέλεσης (σχήμα 7.8).

¹εκτός από τις δοκιμές διαφορετικών χρονισμών μνήμης και χρονικών περιόδων (τελευταίες)



Όπως αναμέναμε υπάρχει πολύ στενή συσχέτιση των δύο μετρήσεων. Επεκτείνοντας την $y = -58.28x + 55.91$ trend line, εκτιμούμε άνω όριο της ταχύτητας που θα είχε το σύστημα με τέλεια προανάκτηση, ως 56 block ανά δευτερόλεπτο, 11% μεγαλύτερη από την υψηλότερη μέτρηση ("NVME Full prefetch") και 33% από την αντίστοιχη χωρίς προανάκτηση.

7.2.5 Δοκιμή σε άλλες χρονικές περιόδους

Όλες οι δοκιμές μέχρι εδώ έχουν γίνει ξεκινώντας από το block με αριθμό (block height) 10.500.000, το οποίο αντιστοιχεί στα μέσα του Ιουλίου 2020, κατά την απότομη αύξηση σε transaction του 2020 [3]. Οι δοκιμές με το δίσκο nvme επαναλήφθηκαν για 15 ακόμα περιόδους από την άνοιξη του 2019 μέχρι του 2021. Παρακάτω φαίνεται το speedup για 2 τύπος προανάκτησης, με το δίσκο nvme:

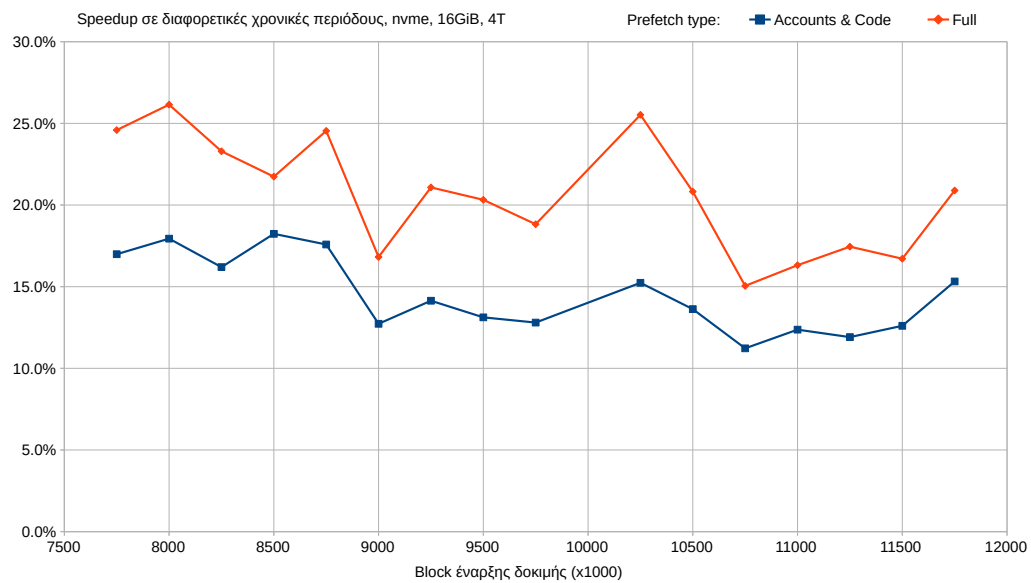


Figure 7.9: .

Αν και υπάρχει μια ελαφρώς πτωτική τάση στη διάρκεια του 2019, η βελτίωση που παρέχει το σύστημα παραμένει σχετικά σταθερή ως προς το χρόνο. Βέβαια, ο όγκος και σύνθεση των transaction διαφέρει ανά περίοδο [3], οπότε είναι αναμενόμενο ότι υπάρχουν διακυμάνσεις, ιδιαίτερα για την περίπτωση πλήρους προανάκτησης (full prefetch).

Κεφάλαιο 8

Επίλογος

Bibliography

- [1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. <https://bitcoin.org/bitcoin.pdf>, archived: <https://web.archive.org/web/20220228183629/https://bitcoin.org/bitcoin.pdf>. Ημερομηνία πρόσβασης: 05-03-22.
- [2] Gavin Wood. *Ethereum: a Secure Decentralised Generalised Transaction Ledger*. <https://github.com/ethereum/yellowpaper/blob/dbdce900343fe3ed2a33dc09bb746b3fa373dc0a/paper.pdf>. Έκδοση Berlin, 21-02-22.
- [3] *Ethereum Daily Transactions Chart*. <https://etherscan.io/chart/tx>, archived: <https://web.archive.org/web/20220216181803/https://etherscan.io/chart/tx>. Ημερομηνία πρόσβασης: 05-03-22.
- [4] *Ethereum Avg. Transaction Fee historical chart*. <https://bitinfocharts.com/comparison/ethereum-transactionfees.html#3y>, archived: <https://web.archive.org/web/20220305165612/https://bitinfocharts.com/comparison/ethereum-transactionfees.html#3y>. Ημερομηνία πρόσβασης: 05-03-22.
- [5] *Go-Ethereum client*. <https://github.com/ethereum/go-ethereum/tree/v1.10.6>. Έκδοση: Terra Nova (v1.10.6), 22-07-21.
- [6] *Erigon client*. <https://github.com/ledgerwatch/erigon/tree/v2021.08.05>. Ημερομηνία έκδοσης: 05-08-21.
- [7] Soujanya Ponnappalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram και Michael Wei. *RainBlock: Faster Transaction Processing in Public Blockchains*. 2021 *USENIX Annual Technical Conference (USENIX ATC 21)*, σελίδες 333-347. USENIX Association, 2021.
- [8] Vikram Saraph και Maurice Herlihy. *An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts*. CoRR, abs/1901.01376, 2019.
- [9] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor και Archit Soman. *An Efficient Framework for Concurrent Execution of Smart Contracts*. CoRR, abs/1809.01326, 2018.
- [10] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy και Eric Koskinen. *Adding concurrency to smart contracts*. *Distributed Comput.*, 33(3-4):209-225, 2020.
- [11] Nadi Sarrar. *On transaction parallelizability in Ethereum*. CoRR, abs/1901.09942, 2019.

- [12] *Prefetching στον Go-Ethereum client.* <https://github.com/ethereum/go-ethereum/commit/bb9631c399392577b1d69a1e8f88a2ccbd05e4e1>.
- [13] *Ethereum upgrades - Ethereum 2.0.* <https://ethereum.org/en/upgrades/>, archived: <https://web.archive.org/web/20220211015539/https://ethereum.org/en/upgrades/>. σημειώσεις μετονομασίας: <https://notes.ethereum.org/@timbeiko/great-renaming>, Ημερομηνία πρόσβασης: 05-03-22.
- [14] *Ethereum Mainnet Statistics.* <https://ethernodes.org/>, archived: <https://web.archive.org/web/20220227125253/https://ethernodes.org/>. Ημερομηνία πρόσβασης: 05-03-22.
- [15] *Ethereum design rationale, Merkle Patricia Trees.* <https://eth.wiki/en/fundamentals/design-rationale#merkle-patricia-trees>, archived: <https://web.archive.org/web/20220306094658/https://eth.wiki/en/fundamentals/design-rationale#merkle-patricia-trees>. Ημερομηνία πρόσβασης: 06-03-22.
- [16] *Erigon, inserting into database in sorted order.* <https://github.com/ledgerwatch/erigon-lib/blob/143cd510bd602fa4999bf79de55ae240eeaa181b/etl/README.md>. Ημερομηνία πρόσβασης: 05-03-22.
- [17] *MDBX.* <https://github.com/erthink/libmdbx>. Ημερομηνία πρόσβασης: 05-03-22.
- [18] Howard Chu. *MDB: A memory-mapped database and backend for OpenLDAP. Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, τόμος 35, 2011. Μετονομάστηκε σε LMDB: https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database#cite_ref-5.
- [19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman και F. Kenneth Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
- [20] *Dataflow Analysis.* https://en.wikipedia.org/wiki/Data-flow_analysis, archived: https://web.archive.org/web/20220209182204/https://en.wikipedia.org/wiki/Data-flow_analysis. Ημερομηνία πρόσβασης: 05-03-22.
- [21] *Feedback Arc Set.* https://en.wikipedia.org/wiki/Feedback_arc_set, archived: https://web.archive.org/web/20220305184813/https://en.wikipedia.org/wiki/Feedback_arc_set. Ημερομηνία πρόσβασης: 05-03-22.
- [22] *Solidity contract metadata.* <https://docs.soliditylang.org/en/v0.8.12/metadata.html>, archived: <https://web.archive.org/web/20220305184124/https://docs.soliditylang.org/en/v0.8.12/metadata.html>. Ημερομηνία πρόσβασης: 05-03-22.
- [23] *Συνάρτηση LockOSThread της Go.* <https://pkg.go.dev/runtime#LockOSThread>, archived: <https://web.archive.org/web/20220209141857/https://pkg.go.dev/runtime#LockOSThread>. Ημερομηνία πρόσβασης: 05-03-22.

- [24] *Συνάρτηση SchedSetaffinity στην Go.* <https://pkg.go.dev/golang.org/x/sys/unix#SchedSetaffinity>, archived: <https://web.archive.org/web/20211127063716/https://pkg.go.dev/golang.org/x/sys/unix>. Ημερομηνία πρόσβασης: 05-03-22.
- [25] *Σελίδα οδηγιών (manpage) του taskset.* <https://man7.org/linux/man-pages/man1/taskset.1.html>, archived: <https://web.archive.org/web/20220216023452/https://man7.org/linux/man-pages/man1/taskset.1.html>. Ημερομηνία πρόσβασης: 05-03-22.
- [26] *Time Stamp Counter.* https://en.wikipedia.org/wiki/Time_Stamp_Counter, archived: https://web.archive.org/web/20220217175243/https://en.wikipedia.org/wiki/Time_Stamp_Counter. Ημερομηνία πρόσβασης: 05-03-22.