ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτιστοποίηση Πρόσβασης στο State του Ethereum μέσω Πολιτικών Prefetching και Caching

---

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Του

ΠΕΡΟΓΑΜΒΡΟΥ Σ. ΓΕΩΡΓΙΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης

Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2025

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

Βελτιστοποίηση Πρόσβασης στο State του Ethereum μέσω Πολιτικών Prefetching και Caching

---

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Του

ΠΕΡΟΓΑΜΒΡΟΥ Σ. ΓΕΩΡΓΙΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 03 Νοεμβρίου 2025

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.

Αθήνα, Νοέμβριος 2025

.....

Περόγαμβρος Γεώργιος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
3 Νοεμβρίου 2025

Περίληψη

Η παρούσα διπλωματική εργασία εστιάζει στη βελτιστοποίηση της πρόσβασης στο state του Ethereum μέσω τεχνικών prefetching και caching. Η έρευνα αναλύει το πρόβλημα του υψηλού I/O latency και της περιορισμένης τοπικότητας πρόσβασης στα δεδομένα του Ethereum, που αποτελούν βασικά εμπόδια για την επεκτασιμότητα του δικτύου. Αναπτύχθηκε προσομοιωτής σε Python, ο οποίος επιτρέπει την αξιολόγηση και σύγκριση διαφορετικών πολιτικών προφόρτωσης, συγκεκριμένα της ReadAhead (χρονική προφόρτωση) και της Per-Contract Top-K (στατιστική προφόρτωση ανά smart contract).

Τα πειραματικά αποτελέσματα, βασισμένα σε πραγματικά δεδομένα από το Ethereum mainnet μέσω BigQuery, δείχνουν ότι η πολιτική ReadAhead επιτυγχάνει ικανοποιητικό speedup με χαμηλό waste σε workloads με λιγότερα storage accesses, ενώ η Top-K προσφέρει καλύτερη απόδοση σε μεγαλύτερα workloads, που επικεντρώνονται στο επίπεδο του storage. Η εργασία τεκμηριώνει τη δυνατότητα των prefetch πολιτικών να μειώσουν ουσιαστικά τον χρόνο πρόσβασης στο state, ενισχύοντας τη συνολική αποδοτικότητα και την ενεργειακή βιωσιμότητα του Ethereum. Επιπλέον, εισάγει ένα αναπαραγώγιμο πειραματικό πλαίσιο, το οποίο μπορεί να επεκταθεί σε πιο προχωρημένα adaptive ή learning-based prefetching μοντέλα, προσφέροντας ερευνητική βάση για τη βελτιστοποίηση blockchain συστημάτων.

Λέξεις-κλειδιά: Ethereum, επίδοση blockchain, πρόσβαση state, προσωρινή αποθήκευση, προανάκτηση, ReadAhead, Top-K, επίπεδο εκτέλεσης, βελτιστοποίηση I/O, τοπικότητα δεδομένων

Abstract

This diploma thesis focuses on the optimization of Ethereum state access through prefetching and caching policies. The study addresses the challenge of high I/O latency and limited data locality, both of which constrain the scalability of Ethereum clients. A Python-based simulator was developed to evaluate and compare alternative prefetching policies, namely the ReadAhead (temporal look-ahead) and Per-Contract Top-K (statistical contract-level prefetching).

Experimental results using real Ethereum mainnet traces extracted via Google BigQuery demonstrate that the ReadAhead policy achieves an important speedup when used at workloads with fewer storage accesses, while the Top-K policy exhibits a better performance in bigger workloads that are heavier on a storage level. The thesis establishes that targeted prefetching strategies can significantly reduce state access latency, improving both the performance and the energy efficiency of Ethereum nodes. Moreover, it introduces a reproducible experimental framework for evaluating caching and prefetching mechanisms that could be used for future adaptive and machine-learning-driven prefetching models aimed at enhancing blockchain scalability and sustainability.

Keywords: Ethereum, blockchain performance, state access, caching, prefetching, ReadAhead, Top-K, execution layer, latency optimization, data locality

στους γονείς μου

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη για την επίβλεψη της παρούσας διπλωματικής. Ευχαριστώ ιδιαιτέρως την Δρ. Κατερίνα Δόκα για την καθοδήγηση και τη συνεχής υποστήριξη καθόλη τη διάρκεια της εν λόγω εργασίας. Τέλος, θέλω να αποδώσω ένα πολύ μεγάλο ευχαριστώ στους γονείς μου που μου συμπαραστέκονται και είναι πάντα δίπλα μου όλα αυτά τα χρόνια.

Αθήνα, Νοέμβριος 2025

Περόγαμβρος Γεώργιος

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κατάλογος Κωδικών

Κεφάλαιο 1 – Εισαγωγή

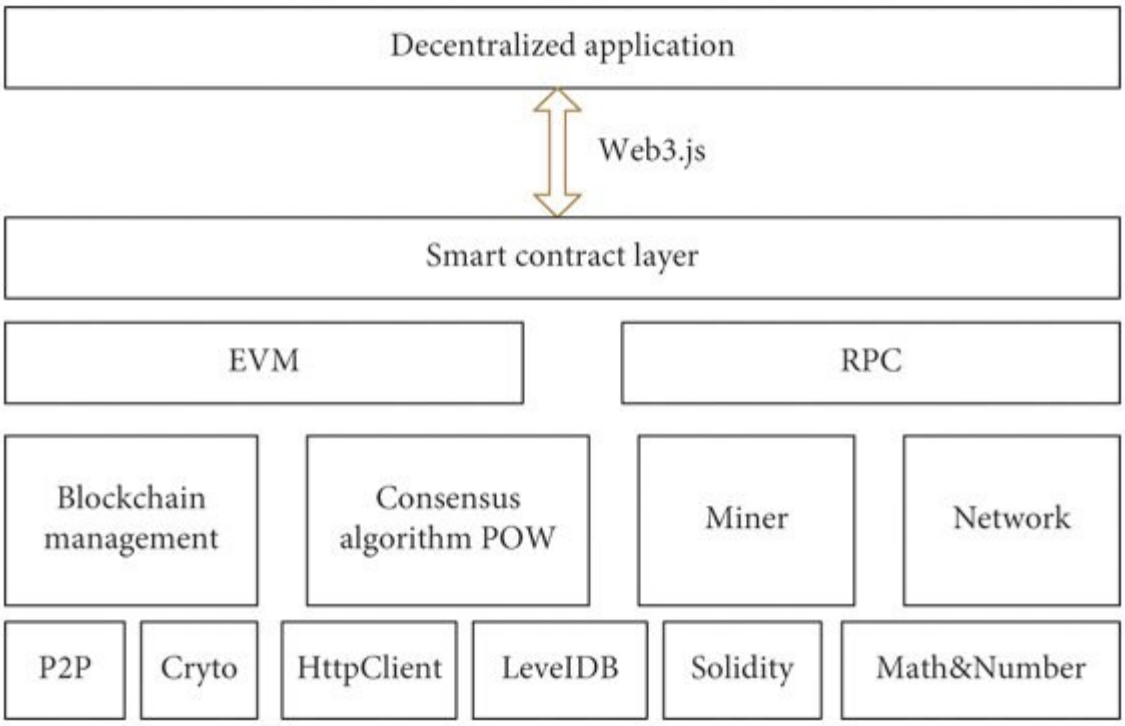
1.1 Κίνητρο και πρόβλημα

Η ανάπτυξη και η λειτουργία των σύγχρονων blockchain συστημάτων, και ειδικότερα του Ethereum, βασίζονται σε ένα ιδιαίτερα απαιτητικό μοντέλο αποθήκευσης και ανάκτησης δεδομένων, γνωστό ως *state model*. Σε αντίθεση με τις παραδοσιακές βάσεις δεδομένων, το state του Ethereum αποτελείται από εκατομμύρια *accounts* και *storage slots* που μεταβάλλονται συνεχώς καθώς εκτελούνται συναλλαγές και *smart contracts* (Wood, 2014). Κατά την επεξεργασία κάθε block, ο client (π.χ. Geth ή Nethermind) πρέπει να διαβάσει και να ενημερώσει μεγάλο αριθμό στοιχείων του state, συχνά σε μη διαδοχικές τοποθεσίες του αποθηκευτικού μέσου. Αυτή η διαδικασία προκαλεί έντονο *I/O bottleneck*, ιδιαίτερα όταν οι προσβάσεις δεν επωφελούνται από την cache ή όταν το αποθηκευτικό μέσο έχει υψηλό latency (Buterin et al., 2021).

Το πρόβλημα επιτείνεται από το γεγονός ότι η *τοπικότητα πρόσβασης* στα δεδομένα του blockchain δεν είναι πάντα προφανής. Ενώ σε ορισμένες εφαρμογές (όπως οι ERC-20 συναλλαγές) παρατηρείται υψηλή επαναχρησιμοποίηση των ίδιων λογαριασμών ή *storage slots*, σε άλλες (όπως decentralized exchanges ή NFTs) η πρόσβαση είναι σχεδόν τυχαία και διασπαρμένη (Hias et al., 2024). Αυτή η ετερογένεια καθιστά δύσκολη την αποδοτική αξιοποίηση των μηχανισμών caching, καθώς οι παραδοσιακές πολιτικές όπως *Least Recently Used (LRU)* ή *Least Frequently Used (LFU)* συχνά αποτυγχάνουν να προλάβουν τις αλλαγές στα πρότυπα πρόσβασης (Bai et al., 2024).

Η υποκείμενη πρόκληση έγκειται στο πώς μπορεί να μειωθεί ο χρόνος πρόσβασης στα δεδομένα του blockchain, χωρίς να αλλοιώνεται η ορθότητα της εκτέλεσης. Οι *clients* του Ethereum λειτουργούν σε ένα περιβάλλον όπου κάθε block πρέπει να επεξεργαστεί εκατοντάδες χιλιάδες *state reads/writes* μέσα σε αυστηρά χρονικά όρια (π.χ. ~12 δευτερόλεπτα ανά block). Αν και τα συστήματα αρχείων και οι SSDs έχουν εξελιχθεί, η προσπέλαση τυχαίων κλειδιών (key-value lookups) σε βάσεις όπως το LevelDB ή το RocksDB εξακολουθεί να αποτελεί το κύριο σημείο καθυστέρησης (Kurisaka et al., 2025). Το *prefetching*, δηλαδή η προληπτική φόρτωση δεδομένων στη μνήμη πριν ζητηθούν, αποτελεί μια κλασική τεχνική για τη μείωση αυτών των καθυστερήσεων (Li et al., 2021). Ωστόσο, η εφαρμογή της σε περιβάλλοντα blockchain δεν έχει μελετηθεί επαρκώς, κυρίως λόγω της πολυπλοκότητας των ακολουθιών πρόσβασης και της έλλειψης σαφούς προβλεψιμότητας στις συναλλαγές.

Από το 2019 και μετά, με τη ραγδαία αύξηση της χρήσης του Ethereum η ανάγκη για αποδοτικότερη διαχείριση state data έγινε κρίσιμη. Οι clients αναγκάζονται να συντηρούν τεράστιες βάσεις (άνω των 500 GB για full nodes) και να εξυπηρετούν αναζητήσεις που απαιτούν δεκάδες χιλιάδες τυχαίες αναγνώσεις ανά δευτερόλεπτο (Ethereum Foundation, 2023). Σε αυτό το πλαίσιο, κάθε μικρή βελτίωση στον *cache hit rate* μπορεί να μεταφραστεί σε σημαντική μείωση του χρόνου επεξεργασίας block και, κατ' επέκταση, σε μεγαλύτερη επεκτασιμότητα του δικτύου (Ergen et al., 2024). Η δομή των δεδομένων και οι πολλαπλές αλληλεπιδράσεις μεταξύ των επιπέδων της αποθήκευσης και της εκτέλεσης δημιουργούν πολύπλοκα μονοπάτια I/O. Στην Εικόνα 1 φαίνεται η τυπική ιεραρχία που υιοθετείται από ένα Ethereum client, συμπεριλαμβανομένων των επιπέδων EVM, συμβολαίων, state storage και βάσεων δεδομένων (π.χ. LevelDB). Η εικόνα αυτή υπογραμμίζει πώς κάθε εντολή SLOAD/SSTORE μπορεί να προκαλέσει πολλαπλά hops μεταξύ των επιπέδων και να συμβάλλει στην καθυστέρηση ανάγνωσης (latency).



Εικόνα 1. Διάγραμμα της αρχιτεκτονικής του Ethereum, που απεικονίζει την ιεραρχία των επιπέδων — από τα συμβόλαια και το EVM έως τις βάσεις δεδομένων (π.χ. LevelDB) και τα I/O operations. (Πηγή: Xu, Gao & Xiao, 2022)

Η υπάρχουσα βιβλιογραφία προσφέρει παραδείγματα προφόρτωσης δεδομένων σε άλλα πεδία, όπως σε λειτουργικά συστήματα (read-ahead algorithms), βάσεις δεδομένων και distributed file systems, όπου οι πολιτικές προανάκτησης βελτιώνουν την απόδοση αξιοποιώντας τη χωρική και χρονική τοπικότητα (Zheng et al., 2017). Ωστόσο, οι ίδιες τεχνικές δεν μπορούν να

εφαρμοστούν αυτούσιες στο Ethereum, καθώς η δυναμική των *smart contracts* και η μη ντετερμινιστική φύση των *transactions* δημιουργούν πρόσθετες δυσκολίες. Η απουσία έρευνας για προσαρμοστικές πολιτικές προφόρτωσης που να βασίζονται στη δομή των συμβολαίων και στη στατιστική ανάλυση των προσπελάσεων συνιστά ένα σαφές ερευνητικό κενό.

Η παρούσα εργασία επιχειρεί να καλύψει αυτό το κενό, προτείνοντας και αξιολογώντας πολιτικές προφόρτωσης προσαρμοσμένες στο περιβάλλον του Ethereum. Ειδικότερα, το κίνητρο πηγάζει από την ανάγκη να μελετηθεί αν απλές, αλλά καλά παραμετροποιημένες στρατηγικές, όπως η Read-Ahead, που προφορτώνει δεδομένα των επόμενων blocks, ή η Per-Contract Top-K, που βασίζεται στις πιο συχνές προσπελάσεις ανά συμβόλαιο, μπορούν να επιτύχουν ουσιαστική βελτίωση της απόδοσης χωρίς αύξηση του κόστους μνήμης ή πολυπλοκότητας. Μέσω προσομοίωσης και ανάλυσης μετρικών όπως *hit rate*, *coverage* και *waste*, η έρευνα επιδιώκει να αποδείξει ότι στοχευμένες τεχνικές caching μπορούν να μετατρέψουν το blockchain από ένα σύστημα περιορισμένο από I/O σε ένα περιβάλλον με αυξημένη προβλεψιμότητα και ταχύτερη εκτέλεση (Hias et al., 2024; Kurisaka et al., 2025).

Η σημασία του προβλήματος εκτείνεται πέρα από τη θεωρητική απόδοση. Επηρεάζει την ενεργειακή αποδοτικότητα, το κόστος λειτουργίας των κόμβων και την αποκέντρωση του δικτύου. Όσο πιο αποδοτικά διαχειρίζεται ένας client το state, τόσο λιγότερους πόρους απαιτεί, κάτι που επιτρέπει τη συμμετοχή μικρότερων κόμβων και ενισχύει τη βιωσιμότητα του οικοσυστήματος (Jung et al., 2023). Συνεπώς, η μελέτη της προφόρτωσης δεν είναι απλώς ένα τεχνικό πρόβλημα βελτιστοποίησης, αλλά ένα ζήτημα κλιμάκωσης και ισότητας συμμετοχής στο blockchain δίκτυο.

Η έρευνα αυτή, επομένως, επιδιώκει να δώσει απάντηση στο κεντρικό ερώτημα: *Μπορούν στοχευμένες πολιτικές προφόρτωσης να μειώσουν ουσιαστικά το I/O latency στους Ethereum clients χωρίς υπερβολική σπατάλη πόρων;* Η διερεύνηση αυτού του ερωτήματος μπορεί να προσφέρει νέα γνώση στην περιοχή του *blockchain systems optimization*, συμβάλλοντας τόσο στη θεωρητική κατανόηση του προβλήματος όσο και στην πρακτική βελτίωση των υπαρχόντων clients (Buterin et al., 2021; Ergen et al., 2024).

1.2 Σκοπός, και στόχοι της εργασίας

Η εργασία στοχεύει να προτείνει, να υλοποιήσει και να συγκρίνει δύο κύριες πολιτικές προφόρτωσης:

α) τη Read-Ahead Policy, η οποία βασίζεται στην προληπτική ανάγνωση δεδομένων από τα επόμενα blocks με σταθερή απόσταση (*look-ahead distance*)· και

β) την Per-Contract Top-K Policy, η οποία χρησιμοποιεί στατιστικά ανά συμβόλαιο για να προβλέπει ποια storage slots είναι πιθανότερο να προσπελαστούν στο άμεσο μέλλον.

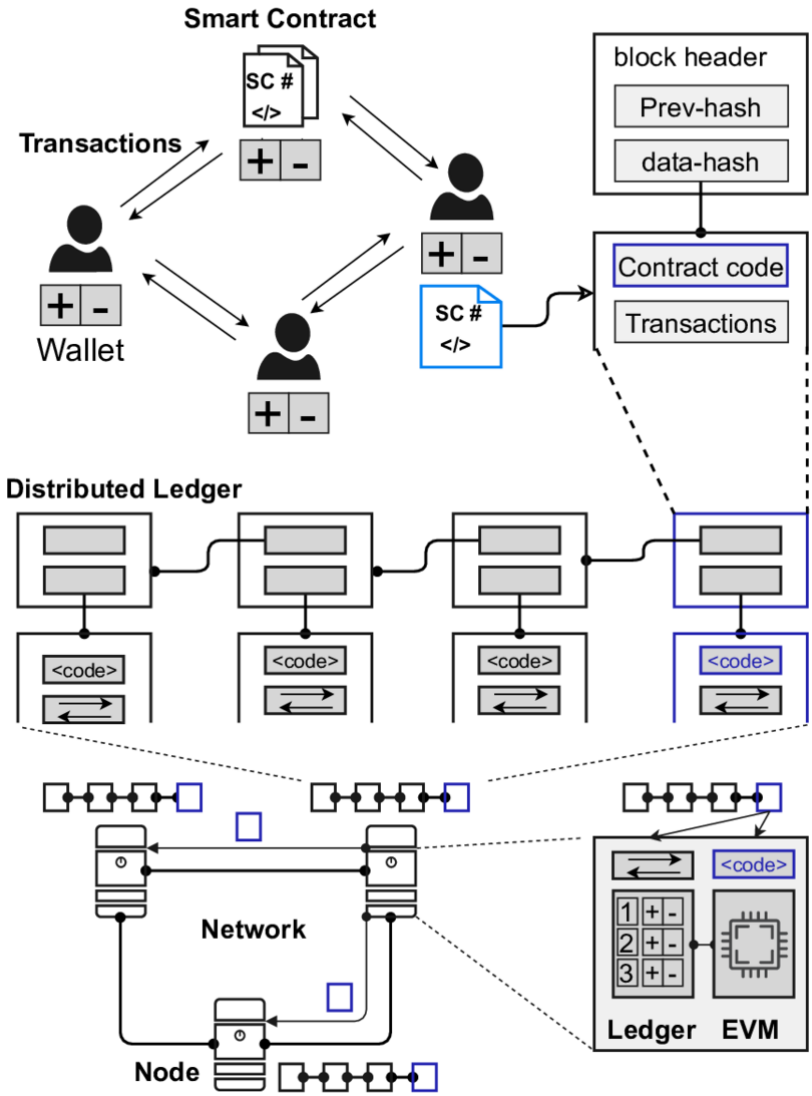
Μέσα από έναν προσομοιωτή caching που αναπτύχθηκε στο πλαίσιο της εργασίας, οι πολιτικές αυτές αξιολογούνται υπό ελεγχόμενες συνθήκες, με κοινά workloads, ώστε να εξασφαλιστεί *fair comparison* και αναπαραγωγιμότητα αποτελεσμάτων.

Ειδικότεροι στόχοι:

1. Ανάλυση του προβλήματος τοπικότητας στο state του Ethereum, με έμφαση στη χωρική και χρονική συσχέτιση των προσπελάσεων σε *accounts* και *storage slots*.
2. Σχεδιασμός και ανάπτυξη ενός προσομοιωτή προφόρτωσης, ο οποίος να επιτρέπει την αξιολόγηση διαφορετικών πολιτικών με ρυθμιζόμενες παραμέτρους.
3. Υλοποίηση της πολιτικής Read-Ahead.
4. Υλοποίηση της πολιτικής Per-Contract Top-K, με ενσωμάτωση offline στατιστικής ανάλυσης μέσω Python εργαλείου (analyze.py), που υπολογίζει τις πιο συχνές θέσεις αποθήκευσης ανά συμβόλαιο.
5. Πειραματική αξιολόγηση των δύο πολιτικών σε συνθετικά και πραγματικά workloads, που προέρχονται από το Ethereum mainnet μέσω BigQuery.
6. Συγκριτική μελέτη και ανάλυση trade-offs μεταξύ των πολιτικών, με στόχο να εντοπιστεί πότε και υπό ποιες συνθήκες κάθε προσέγγιση είναι πιο αποδοτική.
7. Ανάδειξη της πρακτικής χρησιμότητας των αποτελεσμάτων και προτάσεις για ενσωμάτωση των ευρημάτων σε πραγματικούς blockchain clients, με απώτερο σκοπό την επιτάχυνση του execution layer.

1.3 Κενά στη βιβλιογραφία

Παρά τη ραγδαία εξέλιξη της τεχνολογίας blockchain και την αυξανόμενη ερευνητική δραστηριότητα γύρω από την επεκτασιμότητα και την αποδοτικότητα των δικτύων, η έρευνα για τεχνικές caching και prefetching στο επίπεδο εκτέλεσης (execution layer) παραμένει περιορισμένη. Οι περισσότερες μελέτες εστιάζουν σε ζητήματα συναλλακτικής επεκτασιμότητας, συμφωνίας (consensus) και βέλτιστης αποθήκευσης δεδομένων (storage optimization), αφήνοντας το πρόβλημα της *πρόσβασης στο state* ουσιαστικά υποεξερεύνητο (Hias et al., 2024). Η κατάσταση αυτή δημιουργεί σημαντικά ερευνητικά κενά, τα οποία η παρούσα εργασία επιδιώκει να καλύψει. **Η εικόνα 2** απεικονίζει μια τυπική αρχιτεκτονική blockchain με έμφαση στο πού "κάθε smart contract" μεταβαίνει σε state storage — τονίζοντας το κενό ανάμεσα στη λογική του συμβολαίου και το επίπεδο αποθήκευσης.



Εικόνα 2. Αρχιτεκτονικό διάγραμμα που απεικονίζει τα επίπεδα blockchain και τη θέση των "έξυπνων συμβολαίων" (smart contracts) μέσα στην αλυσίδα, υπογραμμίζοντας τη ροή από transactions → contract logic → state storage → block header και state root. (Πηγή: Nguyen, 2022)

Κενά στην υπάρχουσα βιβλιογραφία:

Πρώτον, η υπάρχουσα βιβλιογραφία εστιάζει κυρίως σε τεχνικές βελτιστοποίησης του *consensus* (π.χ. Proof of Stake, sharding) και λιγότερο στη διαχείριση του *state I/O*, που αποτελεί τον πραγματικό παράγοντα καθυστέρησης στην εκτέλεση των συναλλαγών (Buterin et al., 2021). Τα υπάρχοντα έργα κατά κύριο λόγο αναδεικνύουν τη σημασία του *storage backend* στη συνολική καθυστέρηση των Ethereum clients, ωστόσο προτείνουν κυρίως λύσεις σε επίπεδο hardware (π.χ. χρήση NVMe SSDs ή database tuning), χωρίς να εξετάζουν προγνωστικές πολιτικές προφόρτωσης.

Δεύτερον, οι περισσότερες υπάρχουσες τεχνικές caching στον χώρο των blockchain είναι ad-hoc, με στατικά όρια μεγέθους και χωρίς προσαρμοστικότητα στα μοτίβα πρόσβασης. Οι

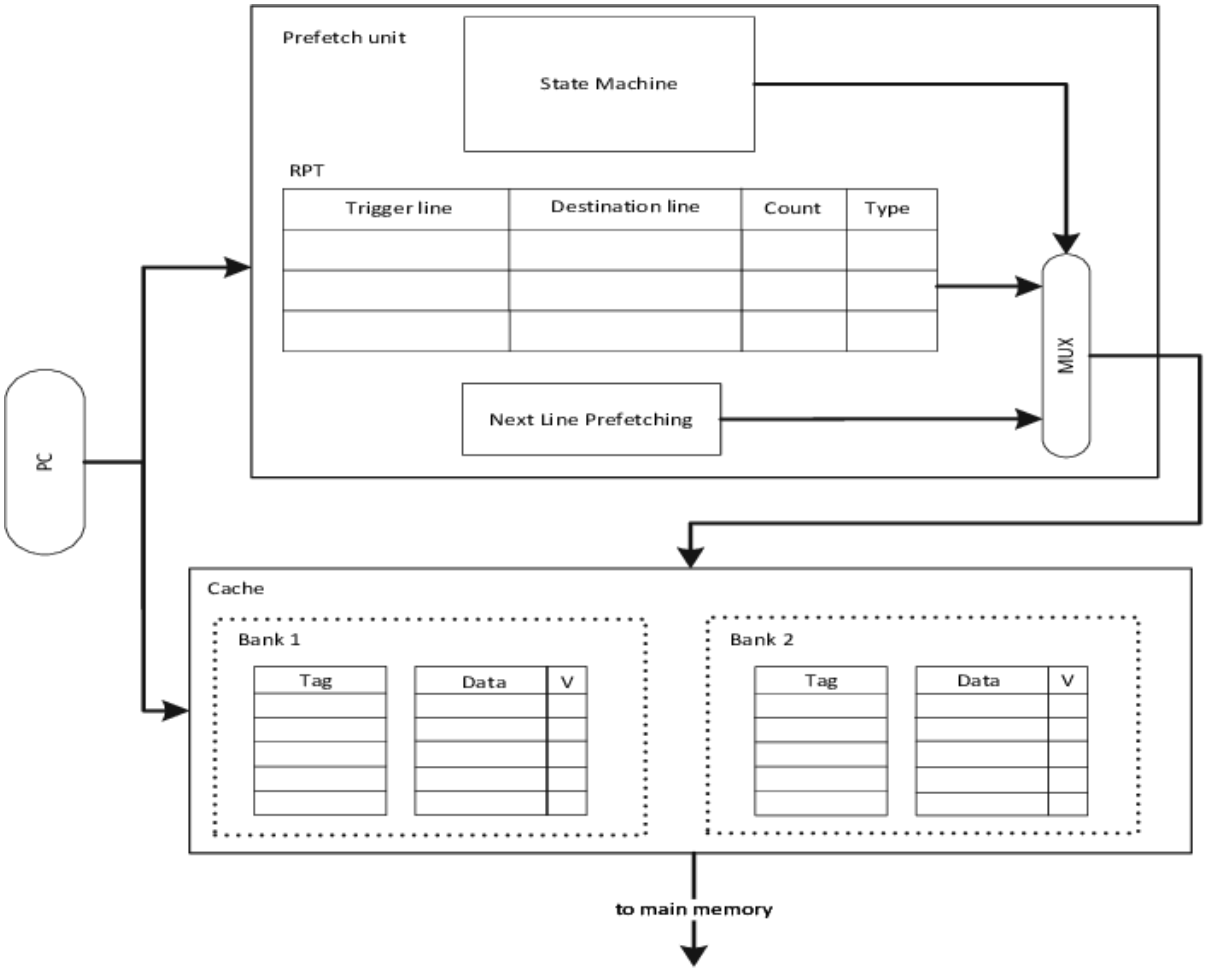
clients όπως ο Geth και ο Nethermind εφαρμόζουν στοιχειώδεις LRU caches ή *snapshot caching* μηχανισμούς, οι οποίοι δεν διαθέτουν προληπτικό χαρακτήρα (Ethereum Foundation, 2023). Αυτό σημαίνει ότι η απόδοση τους εξαρτάται αποκλειστικά από το παρελθόν (πρόσφατα χρησιμοποιημένα δεδομένα), χωρίς αξιοποίηση στατιστικών ή χρονικών προτύπων για μελλοντικές προσπελάσεις (Bai et al., 2024).

Τρίτον, η πλειονότητα των μελετών για prefetching προέρχεται από το πεδίο των λειτουργικών συστημάτων και των αρχιτεκτονικών επεξεργαστών, όπου το πρόβλημα της πρόβλεψης επόμενων προσπελάσεων βασίζεται σε σειριακές ακολουθίες δεδομένων (Li et al., 2021). Αντίθετα, στο Ethereum οι ακολουθίες αυτές είναι *μη γραμμικές* και εξαρτώνται από τη λογική των έξυπνων συμβολαίων. Η δυσκολία πρόβλεψης των μελλοντικών *storage slots* ή *accounts* περιορίζει τη χρησιμότητα των κλασικών prefetchers, όπως των sequential ή stride-based μοντέλων (Zheng et al., 2017).

Τέλος, υπάρχει έλλειψη σε συστηματικές μελέτες που να συνδυάζουν πραγματικά δεδομένα εκτέλεσης από το Ethereum mainnet με πειραματική αξιολόγηση πολιτικών prefetching. Οι περισσότερες προσεγγίσεις βασίζονται σε θεωρητικά μοντέλα ή σε απλουστευμένες υποθέσεις για το workload, χωρίς εμπειρική επιβεβαίωση των αποτελεσμάτων (Jung et al., 2023). Επομένως, η έλλειψη ενός ενιαίου, αναπαραγώγιμου πλαισίου αξιολόγησης δημιουργεί ένα ουσιαστικό επιστημονικό κενό.

1.4 Επισκόπηση προτεινόμενων πολιτικών (ReadAhead, ContractTopK)

Η βελτίωση της απόδοσης πρόσβασης στο *state* του Ethereum προϋποθέτει την ύπαρξη μηχανισμών που να μπορούν να προβλέψουν και να προφορτώσουν (prefetch) τα δεδομένα που πρόκειται να ζητηθούν στο άμεσο μέλλον, μειώνοντας έτσι τη συχνότητα των ακριβών *misses* στη μνήμη ή στο δίσκο. Η παρούσα εργασία εισάγει και συγκρίνει δύο πολιτικές προφόρτωσης: τη ReadAhead και τη Per-Contract Top-K. Οι πολιτικές αυτές σχεδιάστηκαν ώστε να είναι απλές, επεξηγήσιμες και επεκτάσιμες, ώστε να μπορούν να ενσωματωθούν εύκολα σε υπάρχοντες blockchain clients (Kurisaka et al., 2025). **Η Εικόνα 3** παρουσιάζει ένα τυπικό μοντέλο prefetch-cache, με μονάδα prefetch που τροφοδοτεί την κύρια cache μέσω επεξεργασίας προβλεπτικών γραμμών (RPT → next-line).



Εικόνα 3. Αρχιτεκτονικό διάγραμμα ενός prefetch-cache μηχανισμού, όπου η μονάδα prefetch (με πίνακα RPT — Reference Prediction Table) οδηγεί σε προφόρτωση γραμμών cache πριν από τη χρήση τους (next-line prefetching). (Πηγή: Cilku & Puschner, 2015)

1.4.1.Η πολιτική ReadAhead

Η πολιτική ReadAhead βασίζεται σε μια κλασική αρχή των συστημάτων αρχείων και λειτουργικών συστημάτων: ότι οι επερχόμενες προσπελάσεις τείνουν να ακολουθούν μια *σειριακή* ή *ημι-σειριακή* ακολουθία (Li et al., 2021). Εφαρμόζοντας αυτήν την ιδέα στο περιβάλλον του Ethereum, η πολιτική προφορτώνει δεδομένα από ένα ή περισσότερα επόμενα blocks, πριν αυτά εκτελεστούν. Συγκεκριμένα, για κάθε τρέχον block B, η ReadAhead εξετάζει

τα blocks *B+1, B+2, …, B+n* (όπου *n* το παραμετροποιήσιμο *look-ahead distance*) και προφορτώνει όλους τους λογαριασμούς (*accounts*) και τα κλειδιά αποθήκευσης (*storage slots*) που πρόκειται να χρησιμοποιηθούν.

Με αυτόν τον τρόπο, οι πιθανές καθυστερήσεις ανάγνωσης (miss latencies) μετατρέπονται σε προκαταβολικά προγραμματισμένες προαναγνώσεις με χαμηλότερο κόστος (prefetch latency). Η κύρια αδυναμία της ReadAhead είναι η εξάρτησή της από τη χρονική προβλεψιμότητα. Σε workloads όπου οι συναλλαγές δεν παρουσιάζουν επαναληψιμότητα, όπως σε *DeFi* πρωτόκολλα ή *NFT marketplaces* με ασύμμετρα μοτίβα, το look-ahead μπορεί να οδηγήσει σε αυξημένο *waste* και περιορισμένο όφελος. Ωστόσο, για συμβατικά workloads (π.χ. επαναλαμβανόμενες κλήσεις στα ίδια smart contracts), η ReadAhead παραμένει μια αποδοτική και σταθερή λύση (Ergen et al., 2024).

1.4.2.Η πολιτική Per-Contract Top-K

Η δεύτερη προτεινόμενη προσέγγιση, η Per-Contract Top-K, υλοποιεί μια *στατιστική στρατηγική προφόρτωσης*. Αντί να κοιτάζει το επόμενο block, η πολιτική αυτή αναλύει εκ των προτέρων (offline) τα ιστορικά δεδομένα εκτέλεσης και υπολογίζει, για κάθε έξυπνο συμβόλαιο, ποια *storage slots* προσπελαύνονται συχνότερα. Ο μηχανισμός αυτός στηρίζεται σε ένα εργαλείο ανάλυσης (*analyze.py*) που δημιουργεί ένα αρχείο JSON με τα “top-K” slots ανά συμβόλαιο, το οποίο στη συνέχεια χρησιμοποιείται κατά την εκτέλεση.

Κατά την προσομοίωση, όταν εντοπίζεται ότι ένα συγκεκριμένο συμβόλαιο θα εμφανιστεί στο επόμενο block, η πολιτική προφορτώνει αυτόματα τα *K* πιο συχνά slots του, μαζί με τον λογαριασμό του. Με αυτόν τον τρόπο, η Per-Contract Top-K επιχειρεί να “μαντέψει” τις επόμενες προσπελάσεις βάσει συχνότητας και όχι χρονικής ακολουθίας, μια προσέγγιση πιο ανθεκτική σε μη προβλέψιμα workloads (Bai et al., 2024).

Η βασική καινοτομία της πολιτικής έγκειται στην εκμετάλλευση της επαναληψιμότητας σε επίπεδο συμβολαίων. Πολλά smart contracts, όπως τα ERC-20 tokens ή τα decentralized exchanges, προσπελαύνουν επανειλημμένα τα ίδια storage slots (π.χ. balances, allowances). Η Top-K πολιτική αξιοποιεί αυτό το μοτίβο, μειώνοντας το πλήθος των τυχαίων αναγνώσεων.

1.4.3.Συνολική θεώρηση

Οι δύο πολιτικές ακολουθούν διαφορετικές στρατηγικές πρόβλεψης:

- Η ReadAhead στοχεύει στη χρονική προβλεψιμότητα·
- Η Top-K στη στατιστική επαναληψιμότητα

Η συμβολή της παρούσας εργασίας είναι ότι υλοποιεί και συγκρίνει αυτές τις στρατηγικές σε κοινό προσομοιωτή, με πραγματικά και συνθετικά δεδομένα, επιτρέποντας τη συστηματική αξιολόγηση της αποδοτικότητάς τους στο πλαίσιο του Ethereum. Μέσα από τις μετρικές *hit rate*, *coverage*, *waste* και *speedup*, τεκμηριώνεται η βελτίωση της αποδοτικότητας, καθώς και τα όρια κάθε πολιτικής υπό διαφορετικά workloads.

1.5 Δομή της εργασίας

Το Κεφάλαιο 1, *Εισαγωγή*, παρουσιάζει το κίνητρο και το πρόβλημα που οδήγησαν στη διερεύνηση της προφόρτωσης δεδομένων (prefetching) στους Ethereum clients, τους στόχους, καθώς και τα κενά της βιβλιογραφίας που επιχειρεί να καλύψει η εργασία. Επιπλέον, παρέχεται μια επισκόπηση των προτεινόμενων πολιτικών προφόρτωσης (ReadAhead, ContractTopK). Το κεφάλαιο ολοκληρώνεται με τη συνολική περιγραφή της δομής της πτυχιακής, προσφέροντας έναν χάρτη κατανόησης του έργου που ακολουθεί.

Το Κεφάλαιο 2, *Θεωρητικό Υπόβαθρο και Συναφής Έρευνα*, εστιάζει στην ανάλυση του execution layer του Ethereum, στο μοντέλο *state* (accounts, storage slots) και στη σημασία της τοπικότητας δεδομένων. Παρουσιάζονται αναλυτικά οι τεχνικές caching και prefetching που χρησιμοποιούνται σε λειτουργικά συστήματα, βάσεις δεδομένων και κατανεμημένα συστήματα, καθώς και οι ερευνητικές εργασίες που σχετίζονται με τη βελτιστοποίηση της πρόσβασης στο blockchain state.

Το Κεφάλαιο 3, *Μεθοδολογία και Σχεδίαση του Προσομοιωτή*, περιγράφει αναλυτικά τη δομή και τη λειτουργία του προσομοιωτή caching που αναπτύχθηκε και παρουσιάζει τις μετρικές αξιολόγησης (hit rate, coverage, waste, speedup).

Το Κεφάλαιο 4, *Υλοποίηση*, εστιάζει στην τεχνική πλευρά της εργασίας. Περιγράφεται η δομή του κώδικα και των φακέλων (configs, data, src, results), τα βασικά scripts (π.χ. simulator.py, analyze.py, run_experiment.py) και τα εργαλεία που υποστηρίζουν τη λειτουργία του προσομοιωτή.

Το Κεφάλαιο 5, *Πειραματική Αξιολόγηση* περιγράφει τα datasets που χρησιμοποιήθηκαν (συνθετικά workloads και πραγματικά Ethereum traces μέσω BigQuery). Παρουσιάζονται συγκριτικά αποτελέσματα για τις πολιτικές ReadAhead και ContractTopK. Μέσα από γραφήματα και πίνακες συνοψίζονται οι μετρήσεις *hit rate*, *coverage*, *waste* και *speedup*, τεκμηριώνοντας ποσοτικά τη βελτίωση απόδοσης που επιφέρουν οι πολιτικές.

Το Κεφάλαιο 6, *Συμπεράσματα και Μελλοντική Εργασία*, συνοψίζει τα βασικά πορίσματα της έρευνας και επισημαίνει τις δυνατότητες επέκτασης του έργου.

Κεφάλαιο 2 – Θεωρητικό Υπόβαθρο και Συναφής Έρευνα

2.1 Το Execution Layer του Ethereum: λογαριασμοί, storage slots και state transitions

Το **Ethereum** αποτελεί ένα από τα πιο διαδεδομένα και τεχνολογικά εξελιγμένα blockchain συστήματα, με θεμελιώδη διαφορά από το Bitcoin στο γεγονός ότι υποστηρίζει *γενικευμένο υπολογισμό* μέσω έξυπνων συμβολαίων (*smart contracts*) (Buterin et al., 2021). Η λειτουργία αυτών των συμβολαίων βασίζεται στο λεγόμενο **execution layer**, το οποίο είναι υπεύθυνο για την εκτέλεση των συναλλαγών, την ενημέρωση του *state* του δικτύου και τη διασφάλιση της συνέπειας και της εγκυρότητας κάθε υπολογισμού. Η κατανόηση της δομής και της δυναμικής του execution layer αποτελεί απαραίτητη προϋπόθεση για οποιαδήποτε μελέτη που αποσκοπεί στη βελτίωση της απόδοσης των clients και ειδικά της πρόσβασης στο *state data*, όπως η παρούσα εργασία.

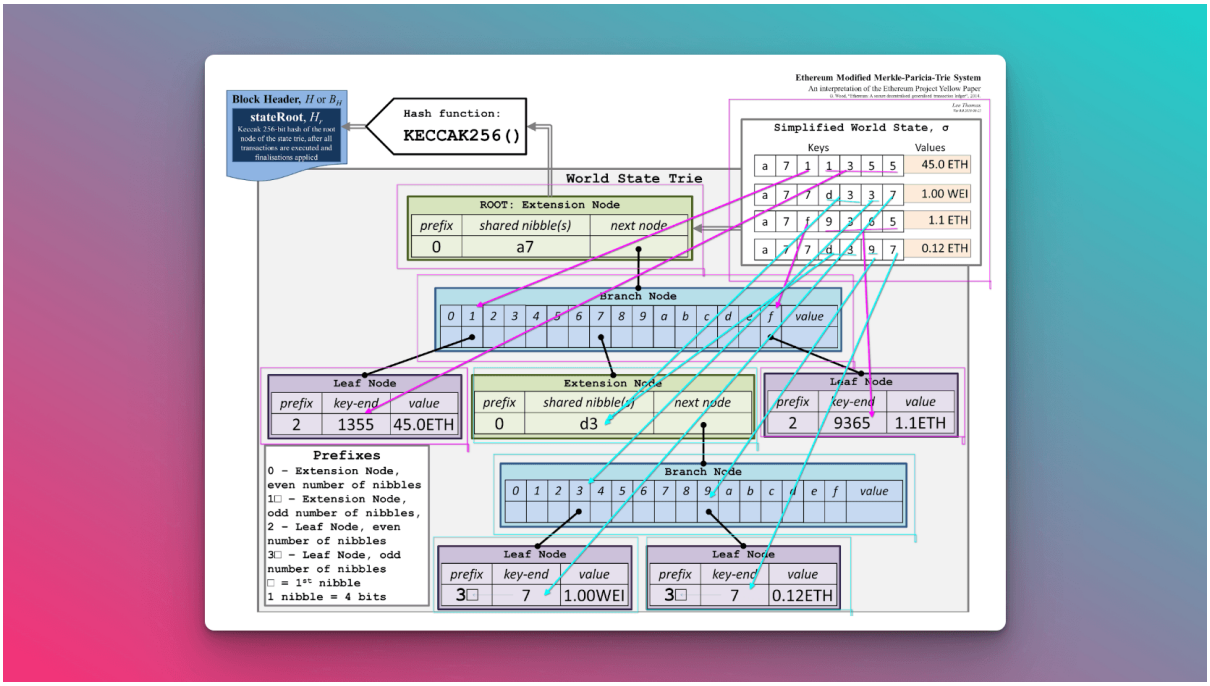
Η έννοια του state στο Ethereum

Το state του Ethereum είναι ένα μεγάλης κλίμακας, μεταβαλλόμενο σύνολο δεδομένων που περιλαμβάνει όλα τα στοιχεία ταυτότητας και κατάστασης του δικτύου: *λογαριασμούς (accounts)*, *υπόλοιπα (balances)*, *κώδικα συμβολαίων (contract code)* και *δεδομένα αποθήκευσης (storage)* (Wood, 2014). Κάθε κόμβος του δικτύου διατηρεί ένα πλήρες αντίγραφο του state, το οποίο ενημερώνεται σε κάθε νέο block μέσω της εκτέλεσης των συναλλαγών που περιέχει. Η αναπαράσταση του state βασίζεται σε μια Merkle Patricia Trie (MPT), όπως φαίνεται και στην Εικόνα 4, μια παραλλαγή δομής δέντρου με κρυπτογραφική δέσμευση (hashing), που εξασφαλίζει την ακεραιότητα και την αποτελεσματική επικύρωση των δεδομένων (Antonopoulos & Wood, 2018).

Στο MPT, κάθε κόμβος (node) αντιστοιχεί είτε σε λογαριασμό είτε σε *storage slot*. Οι λογαριασμοί διακρίνονται σε δύο κατηγορίες:

1. Externally Owned Accounts (EOAs), που ελέγχονται από χρήστες μέσω ιδιωτικών κλειδιών και περιέχουν μόνο ένα balance, και
2. Contract Accounts, που συνδέονται με ένα smart contract και περιλαμβάνουν επιπλέον *storage area* (τοπική μνήμη).

Η εσωτερική μνήμη κάθε συμβολαίου οργανώνεται σε storage slots, 32-byte θέσεις με δείκτες (keys) που αποθηκεύουν τιμές (values) στον χώρο αποθήκευσης του συμβολαίου. Αυτά τα slots αποτελούν το βασικό αντικείμενο προσπέλασης κατά την εκτέλεση εντολών SLOAD (ανάγνωση) και SSTORE (εγγραφή) στη *Virtual Machine* του Ethereum (EVM) (Ergen et al., 2024).



Εικόνα 4. Απεικόνιση του Ethereum World State Trie (Merkle–Patricia Trie), που δείχνει τη σύνδεση μεταξύ block header (stateRoot), hash function (KECCAK256) και των κόμβων του trie (branch, extension, leaf). Το διάγραμμα παρουσιάζει πώς τα κλειδιά και οι τιμές του απλοποιημένου world state αντιστοιχίζονται σε hashed paths μέσω των prefixes και nibbles. (Understanding Merkle Patricia Tries in Ethereum | Guides | GoldRush, n.d.)

Λειτουργία του Execution Layer

Το execution layer είναι υπεύθυνο για τη μετάφραση των συναλλαγών σε μεταβολές του state, μέσα από μια ακολουθία βημάτων που περιλαμβάνει *state transitions*. Κάθε *transaction* που εκτελείται δημιουργεί έναν νέο υπολογιστικό κύκλο, ο οποίος διαβάζει, τροποποιεί και γράφει δεδομένα στο state. Αυτή η διαδικασία περιγράφεται τυπικά ως συνάρτηση μετάβασης:

$$\sigma_{t+1} = \Upsilon(\sigma_t, T)$$

όπου σ_t είναι το state πριν την εκτέλεση, Υη συνάρτηση εκτέλεσης και T η συναλλαγή (Wood, 2014).

Η EVM (Ethereum Virtual Machine) ερμηνεύει bytecode εντολές (π.χ. ADD, CALL, SLOAD, SSTORE) και ενημερώνει το state σύμφωνα με το *gas cost model*. Οι περισσότερες εντολές που αφορούν SLOAD/SSTORE είναι I/O-bound, καθώς απαιτούν ανάγνωση ή εγγραφή από το persistent storage, συνήθως αποθηκευμένο σε LevelDB ή RocksDB στο filesystem του κόμβου (Kurisaka et al., 2025). Εδώ εντοπίζεται το πραγματικό σημείο συμφόρησης (bottleneck) στην απόδοση των Ethereum clients: η διαδικασία ανάκτησης ενός slot από το δίσκο μπορεί να είναι χιλιάδες φορές πιο αργή από μια λειτουργία στη μνήμη (Hias et al., 2024).

Για κάθε συναλλαγή, η EVM πρέπει να αποκτήσει πρόσβαση στα δεδομένα του λογαριασμού που εκτελεί τη λειτουργία, καθώς και σε οποιαδήποτε storage slots αναφέρονται μέσα στον bytecode. Οι προσπελάσεις αυτές δεν είναι απαραίτητα διαδοχικές, αλλά μπορούν να αφορούν διαφορετικά συμβόλαια, τυχαία κλειδιά και περιοχές μνήμης. Αυτό σημαίνει ότι το execution layer παράγει *μη σειριακά access patterns*, τα οποία είναι δύσκολο να προβλεφθούν και να αποθηκευτούν αποδοτικά (Bai et al., 2024).

Η διαχείριση δεδομένων από τους Ethereum clients

Οι δημοφιλέστεροι clients του Ethereum, όπως ο Geth, ο Nethermind και ο Besu, χρησιμοποιούν *persistent key-value databases* (π.χ. LevelDB, RocksDB) για να αποθηκεύουν το state. Κάθε φορά που ένας κόμβος εκτελεί μια συναλλαγή, πρέπει να ανακτήσει από τη βάση τα hashes που αντιστοιχούν στα απαιτούμενα accounts ή storage slots. Επειδή τα δεδομένα αυτά δεν βρίσκονται πάντοτε στη μνήμη, απαιτείται ανάγνωση από τον δίσκο, διαδικασία που εξαρτάται άμεσα από τον τύπο του αποθηκευτικού μέσου (NVMe ή SATA SSDs) (Ethereum Foundation, 2023).

Για να μειωθεί η καθυστέρηση, οι clients εφαρμόζουν απλές πολιτικές caching, κυρίως με LRU (Least Recently Used) στρατηγική. Ωστόσο, αυτές λειτουργούν αντιδραστικά αποθηκεύοντας ό,τι χρησιμοποιήθηκε πρόσφατα, χωρίς να προλαμβάνουν μελλοντικές προσπελάσεις (Li et al., 2021). Επειδή οι προσπελάσεις του EVM είναι δυναμικές και εξαρτώνται από το περιεχόμενο των συναλλαγών, το caching συχνά αποτυγχάνει να προσφέρει υψηλό hit rate, ιδίως σε blocks με μεγάλο αριθμό διαφορετικών συμβολαίων.

Η απουσία προληπτικής προφόρτωσης (prefetching) σημαίνει ότι ο client συχνά αναγκάζεται να περιμένει για την ολοκλήρωση των I/O operations, δημιουργώντας καθυστερήσεις που περιορίζουν τη συνολική ταχύτητα επεξεργασίας block. Στο Ethereum mainnet, ένα block πρέπει να εκτελεστεί και να επικυρωθεί μέσα σε ~12 δευτερόλεπτα· επομένως, ακόμη και μικρές καθυστερήσεις ανάγνωσης μπορούν να οδηγήσουν σε υποβάθμιση της throughput απόδοσης του κόμβου και να επηρεάσουν τη συμμετοχή του στο δίκτυο (Jung et al., 2023).

Οι προκλήσεις του state access

Το πρόβλημα του *state access* στο Ethereum συνδέεται με τρεις βασικούς παράγοντες:

1. Τυχαία πρόσβαση (random access): Οι συναλλαγές δεν είναι προβλέψιμες ως προς τα storage slots που θα προσπελάσουν.
2. Έλλειψη locality: Οι αποθηκευμένες τιμές κατανέμονται ανομοιόμορφα στη Merkle Trie και στα υποκείμενα databases, μειώνοντας τη χωρική τοπικότητα.

3. Υψηλό latency I/O: Η πρόσβαση σε storage nodes απαιτεί πολλαπλά database lookups και hashing επαληθεύσεις, οι οποίες επιβαρύνουν σημαντικά το χρόνο εκτέλεσης (Kurisaka et al., 2025).

Αυτοί οι παράγοντες έχουν οδηγήσει την κοινότητα ανάπτυξης του Ethereum να αναζητά λύσεις που βελτιώνουν την *data locality* και τη διαχείριση του state, όπως τα *Verkle trees*, το *state expiry* και οι *stateless clients* (Buterin et al., 2021). Ωστόσο, οι προσεγγίσεις αυτές στοχεύουν κυρίως στη μείωση του μεγέθους του state και όχι στην επιτάχυνση της πρόσβασης. Το execution layer του Ethereum αποτελεί το πιο απαιτητικό και ταυτόχρονα το πιο κρίσιμο τμήμα του συστήματος, καθώς συγκεντρώνει τις περισσότερες λειτουργίες που επηρεάζουν την απόδοση. Η δομή του state (λογαριασμοί, storage slots) και η λειτουργία της EVM δημιουργούν ένα προβληματικό προφίλ προσπελάσεων, όπου η έλλειψη προβλεψιμότητας και το υψηλό I/O latency περιορίζουν την αποδοτικότητα των clients.

Η ανάλυση του execution layer επομένως παρέχει το τεχνολογικό υπόβαθρο πάνω στο οποίο στηρίζεται η παρούσα ερευνητική συμβολή, δηλαδή η ανάπτυξη, υλοποίηση και πειραματική σύγκριση prefetching πολιτικών που επιχειρούν να γεφυρώσουν το χάσμα μεταξύ της ταχύτητας εκτέλεσης και της αποθήκευσης στο blockchain (Hias et al., 2024; Kurisaka et al., 2025; Jung et al., 2023).

2.2 Cache behavior σε Ethereum clients (π.χ. Geth, Nethermind) και ανάγκη για locality optimization

Η απόδοση ενός Ethereum client εξαρτάται σε μεγάλο βαθμό από τον τρόπο με τον οποίο αποθηκεύει και ανακτά δεδομένα από το *state* κατά την εκτέλεση συναλλαγών. Καθώς κάθε block περιέχει εκατοντάδες ή και χιλιάδες συναλλαγές, που με τη σειρά τους περιλαμβάνουν πολλαπλές εντολές *SLOAD* και *SSTORE*, η απόδοση του caching μηχανισμού καθορίζει σε μεγάλο βαθμό τη συνολική ταχύτητα του execution layer. Ωστόσο, η φύση των blockchain workloads (έντονα *I/O-bound*, μη σειριακή και εξαιρετικά διαφοροποιημένη) καθιστά δύσκολη τη βελτιστοποίηση της *data locality* και την αποδοτική χρήση των caches (Kurisaka et al., 2025).

2.2.1 Δομή και λειτουργία cache στους Ethereum clients

Οι περισσότεροι Ethereum clients, όπως ο Geth, ο Nethermind και ο Besu, χρησιμοποιούν πολυεπίπεδους μηχανισμούς προσωρινής αποθήκευσης (multi-layer caching). Οι μηχανισμοί αυτοί περιλαμβάνουν συνήθως:

- In-memory caches, που αποθηκεύουν προσωρινά τα πιο πρόσφατα ή συχνά χρησιμοποιημένα state entries.
- Database-level caches, που βασίζονται στα εσωτερικά caching layers του LevelDB ή RocksDB (συνήθως LRU ή Bloom-filter-based buffers).
- Snapshot caching, όπου προδημιουργούνται προσωρινά στιγμιότυπα του state για συγκεκριμένα blocks, ώστε να επιταχύνεται η εκτέλεση διαδοχικών συναλλαγών (Ethereum Foundation, 2023).

Στον Geth, για παράδειγμα, κάθε φορά που εκτελείται μια συναλλαγή, το σύστημα επιχειρεί πρώτα να ανακτήσει τα δεδομένα του λογαριασμού ή των *storage slots* από την in-memory state cache. Αν αυτά δεν βρεθούν, πραγματοποιείται *lookup* στο state database, που βρίσκεται συνήθως σε αποθηκευτικό μέσο SSD. Ο χρόνος ανάγνωσης ενός slot από τη βάση μπορεί να κυμαίνεται από μερικές εκατοντάδες μικροδευτερόλεπτα (NVMe SSDs) έως αρκετά milliseconds (SATA drives), κάτι που είναι ιδιαίτερα επιβαρυντικό όταν επαναλαμβάνεται χιλιάδες φορές ανά block (Hias et al., 2024).
Η επιτυχία του caching εξαρτάται από το *hit rate*, δηλαδή το ποσοστό των αναζητήσεων που ικανοποιούνται από τη μνήμη χωρίς πρόσβαση στη βάση. Όσο υψηλότερο είναι το hit rate, τόσο μικρότερη η ανάγκη για ακριβές αναγνώσεις από το storage.

2.2.2 Το πρόβλημα της τοπικότητας (locality) στα blockchain workloads

Η τοπικότητα πρόσβασης αποτελεί θεμελιώδη έννοια στη θεωρία των υπολογιστικών συστημάτων. Διακρίνεται σε δύο βασικές μορφές:

- Χρονική τοπικότητα (temporal locality), όταν ένα αντικείμενο που χρησιμοποιήθηκε πρόσφατα είναι πιθανό να χρησιμοποιηθεί ξανά σύντομα.
- Χωρική τοπικότητα (spatial locality), όταν αντικείμενα που βρίσκονται κοντά στη μνήμη είναι πιθανό να προσπελαστούν διαδοχικά (Li et al., 2021).

Στο Ethereum, όμως, η ύπαρξη τέτοιας τοπικότητας δεν είναι εγγυημένη. Οι συναλλαγές μπορεί να αφορούν εντελώς διαφορετικά smart contracts, με αποτέλεσμα να μην υπάρχει επαναληψιμότητα ή φυσική εγγύτητα μεταξύ των δεδομένων. Για παράδειγμα, δύο διαδοχικές συναλλαγές σε ένα block μπορεί να τροποποιούν τα balances διαφορετικών ERC-20 tokens ή να αλληλεπιδρούν με contracts που δεν έχουν κοινό state. Αυτό οδηγεί σε μη προβλέψιμα access patterns και σε χαμηλή απόδοση των LRU caches (Bai et al., 2024).
Επιπλέον, η δομή των δεδομένων στο Ethereum (συγκεκριμένα η Merkle Patricia Trie) επιδεινώνει το πρόβλημα. Τα κλειδιά (keys) των accounts και slots διασκορπίζονται

ομοιόμορφα μέσω κρυπτογραφικού hashing, γεγονός που εξαλείφει τη χωρική τοπικότητα. Αυτό σημαίνει ότι ακόμη και γειτονικές προσπελάσεις σε συμβόλαια μπορεί να απαιτούν εντελώς διαφορετικά I/O operations στο δίσκο. Οι clients δεν μπορούν επομένως να επωφεληθούν από τις τυπικές τεχνικές caching που στηρίζονται στην εγγύτητα των δεδομένων (Antonopoulos & Wood, 2018).

Η χρονική τοπικότητα παραμένει επίσης περιορισμένη. Παρότι ορισμένα smart contracts, όπως τα DEX pools ή τα token contracts, εμφανίζουν επαναλαμβανόμενες λειτουργίες, η αναλογία αυτών των "θερμών" περιοχών του state παραμένει μικρή σε σχέση με το συνολικό πλήθος των storage entries. Αυτό σημαίνει ότι μόνο ένα μικρό υποσύνολο του state επαναχρησιμοποιείται σε σύντομο χρονικό διάστημα (Hias et al., 2024).

2.2.3 Επιδράσεις της έλλειψης locality στην απόδοση

Η απουσία προβλεψιμότητας στις προσπελάσεις οδηγεί σε χαμηλή απόδοση cache και αυξημένο αριθμό I/O requests ανά block. Όπως έχει τεκμηριωθεί από μετρήσεις πραγματικών κόμβων, κάθε block του Ethereum μπορεί να προκαλέσει δεκάδες χιλιάδες *key-value lookups* σε βάσεις LevelDB, εκ των οποίων το μεγαλύτερο ποσοστό είναι τυχαίες αναγνώσεις (Kurisaka et al., 2025). Το αποτέλεσμα είναι ότι ο client δαπανά έως και 60–70% του συνολικού χρόνου εκτέλεσης σε λειτουργίες ανάγνωσης state, αντί σε καθαρούς υπολογισμούς EVM.

Αυτό έχει σημαντικές συνέπειες όχι μόνο για την ταχύτητα επεξεργασίας blocks, αλλά και για τη βιωσιμότητα του δικτύου. Οι κόμβοι που εκτελούν πιο αργά τα blocks είναι πιθανό να καθυστερούν στη διάδοση νέων blocks, μειώνοντας τις πιθανότητες συμμετοχής τους στη συναίνεση και, κατά συνέπεια, τα πιθανά rewards (Jung et al., 2023). Επίσης, η αυξημένη I/O δραστηριότητα οδηγεί σε υψηλότερη κατανάλωση ενέργειας και φθορά των αποθηκευτικών μέσων, αυξάνοντας το λειτουργικό κόστος ενός full node (Buterin et al., 2021).

Οι υπάρχουσες λύσεις, όπως η αύξηση της cache capacity ή η χρήση ταχύτερων NVMe δίσκων, προσφέρουν μερική μόνο βελτίωση. Η απόδοση δεν αυξάνεται γραμμικά με το μέγεθος της cache, καθώς τα δεδομένα που προσπελαύνονται συχνά αλλάζουν διαρκώς και η τοπικότητα παραμένει περιορισμένη (Ergen et al., 2024). Επομένως, η βελτίωση της απόδοσης δεν μπορεί να βασιστεί αποκλειστικά στο hardware, αλλά απαιτεί έξυπνη διαχείριση της πρόσβασης, μέσω προγνωστικών ή στατιστικών μηχανισμών.

2.2.4 Η ανάγκη για locality optimization

Η locality optimization στο πλαίσιο του Ethereum αναφέρεται στην προσπάθεια αναγνώρισης και εκμετάλλευσης των μοτίβων πρόσβασης στο state, με σκοπό να αυξηθεί το *hit rate* και να μειωθούν οι καθυστερήσεις ανάγνωσης. Οι τεχνικές αυτές μπορούν να εφαρμοστούν σε διάφορα επίπεδα:

1. Execution-level optimization, με προφόρτωση δεδομένων (prefetching) από blocks που αναμένεται να εκτελεστούν σύντομα.
2. Contract-level optimization, με στατιστική ανάλυση της συχνότητας πρόσβασης ανά smart contract και προληπτική φόρτωση των πιο δημοφιλών storage slots.
3. Database-level optimization, με αναδιοργάνωση των key-value pairs ώστε να αποθηκεύονται σε πιο εγγύς περιοχές του δίσκου, ελαχιστοποιώντας τυχαίες προσπελάσεις (Zheng et al., 2017).

Η παρούσα εργασία επικεντρώνεται στις δύο πρώτες κατηγορίες, καθώς είναι ανεξάρτητες από το hardware και μπορούν να υλοποιηθούν σε επίπεδο λογισμικού χωρίς τροποποιήσεις στη βάση δεδομένων. Οι πολιτικές ReadAhead και Per-Contract Top-K αποτελούν δύο διαφορετικές προσεγγίσεις του locality optimization: η πρώτη επιχειρεί να αξιοποιήσει τη χρονική τοπικότητα, ενώ η δεύτερη να αναδείξει τη στατιστική επαναληψιμότητα σε επίπεδο συμβολαίων.

Η ανάγκη για τέτοιες πολιτικές είναι προφανής: το Ethereum έχει πλέον φτάσει σε τέτοια κλίμακα, όπου η πλήρης αποθήκευση του state υπερβαίνει τα 600 GB (Ethereum Foundation, 2023). Με το υπάρχον μοντέλο πρόσβασης, κάθε βελτίωση ακόμη και 10–15% στο hit rate μπορεί να οδηγήσει σε πολλαπλάσια αύξηση του throughput και σε μείωση του ενεργειακού αποτυπώματος των clients. Η υιοθέτηση πολιτικών προφόρτωσης επομένως δεν είναι απλώς τεχνική βελτιστοποίηση, αλλά κρίσιμη στρατηγική για τη μακροπρόθεσμη βιωσιμότητα και αποκέντρωση του δικτύου (Jung et al., 2023; Hias et al., 2024).

Η συμπεριφορά των cache μηχανισμών στους Ethereum clients αποκαλύπτει ένα σημαντικό παράδοξο: ενώ η τεχνολογία αποθήκευσης έχει προοδεύσει, η αποτελεσματικότητα πρόσβασης στα δεδομένα παραμένει χαμηλή λόγω της απουσίας τοπικότητας και προβλεψιμότητας. Οι τυπικές πολιτικές caching που λειτουργούν επαρκώς σε παραδοσιακά συστήματα αποτυγχάνουν σε περιβάλλοντα blockchain, όπου τα access patterns είναι ασύμμετρα και οι καθυστερήσεις I/O συχνά κυριαρχούν στο συνολικό χρόνο εκτέλεσης.

Η αναζήτηση λύσεων βασισμένων στην *locality optimization*, μέσω προφόρτωσης δεδομένων, στατιστικής πρόβλεψης ή προσαρμοστικών πολιτικών caching, αποτελεί επομένως το επόμενο

βήμα για τη βελτίωση της αποδοτικότητας του execution layer. Αυτή η προσέγγιση αποτελεί το κεντρικό αντικείμενο της παρούσας εργασίας.

2.3 Θεμελιώδεις αρχές του prefetching

Το prefetching είναι μια προληπτική τεχνική που επιδιώκει να μειώσει την *καθυστέρηση ανάγνωσης* (read latency) προβλέποντας ποια δεδομένα θα ζητηθούν στο μέλλον και φορτώνοντάς τα εκ των προτέρων στη μνήμη. Ο μηχανισμός αυτός έχει εφαρμοστεί εκτενώς σε λειτουργικά συστήματα και επεξεργαστές, αλλά οι αρχές του μπορούν να επεκταθούν και σε επίπεδο blockchain execution layer (Li et al., 2021).

Η αποτελεσματικότητα του prefetching μετράται με βάση τρεις κύριες μετρικές:

1. Coverage: το ποσοστό των μελλοντικών αιτημάτων που προφορτώθηκαν επιτυχώς.
2. Accuracy (ή *usefulness*): το ποσοστό των προφορτωμένων δεδομένων που πράγματι χρησιμοποιήθηκαν.
3. Timeliness: το πόσο νωρίς ή αργά προφορτώθηκαν τα δεδομένα σε σχέση με την ανάγκη τους (Hasan et al., 2020).

Η βασική πρόκληση είναι η ισορροπία μεταξύ coverage και accuracy. Ένα prefetcher με υψηλό coverage αλλά χαμηλή accuracy καταναλώνει πόρους χωρίς ουσιαστικό όφελος, ενώ ένας με χαμηλό coverage παραμένει αναποτελεσματικός. Η επίτευξη αυτής της ισορροπίας αποτελεί αντικείμενο ενεργούς έρευνας (Bai et al., 2024).

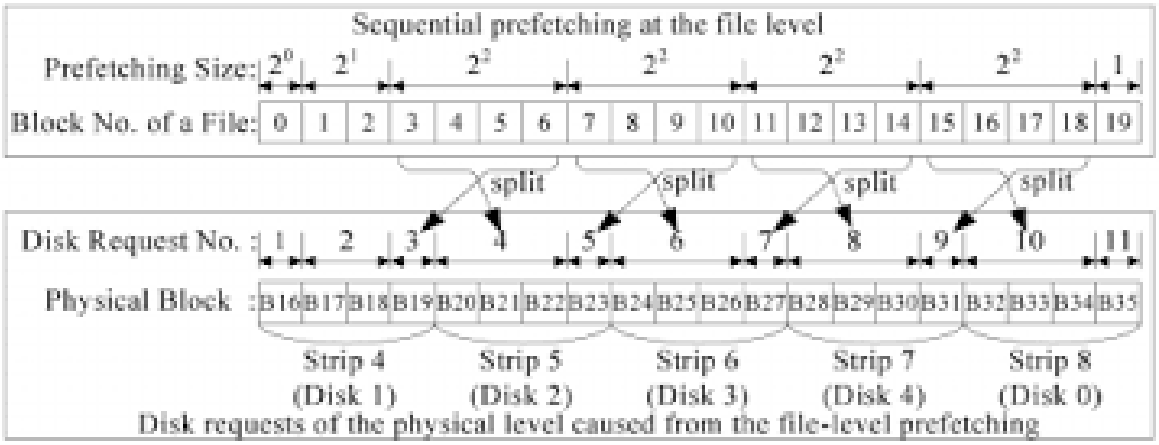
2.3.1 Sequential και look-ahead prefetching

Η απλούστερη μορφή προφόρτωσης είναι το sequential prefetching, που χρησιμοποιείται παραδοσιακά σε file systems (όπως ext4 ή NTFS) και σε block-based storage. Το σύστημα υποθέτει ότι, αν ζητήθηκε το block i, τότε πιθανόν να ζητηθούν σύντομα και τα $i+1$, $i+2$, κ.ο.κ. (Zheng et al., 2017). Αυτή η στρατηγική αξιοποιεί την *χωρική τοπικότητα* και επιτυγχάνει υψηλό hit rate σε σειριακά workloads, αλλά αποτυγχάνει όταν η πρόσβαση είναι μη γραμμική.

Η look-ahead prefetching αποτελεί πιο εξελιγμένη εκδοχή, όπου το σύστημα χρησιμοποιεί παραμετροποιήσιμη *απόσταση πρόβλεψης* (look-ahead distance, d). Στο πλαίσιο του Ethereum, μια τέτοια προσέγγιση αντιστοιχεί στην πολιτική ReadAhead, η οποία προφορτώνει δεδομένα από τα επόμενα blocks ($B+1$, $B+2$, …, $B+d$), προσπαθώντας να καλύψει χρονική τοπικότητα μεταξύ blocks (Ergen et al., 2024).

Αυτή η τεχνική είναι απλή και αποδοτική όταν το workload έχει επαναληψιμότητα ή προβλεψιμότητα. Ωστόσο, η επιλογή του d είναι κρίσιμη: πολύ μικρή τιμή οδηγεί σε χαμηλό coverage, ενώ πολύ μεγάλη μειώνει το accuracy. Επομένως, η απόδοση του look-ahead

prefetching εξαρτάται από την *προσαρμοστικότητα* της πολιτικής στα χαρακτηριστικά του workload (Kurisaka et al., 2025).



Εικόνα 5. Σχηματική απεικόνιση της διαδοχικής προφόρτωσης σε επίπεδο αρχείου (sequential prefetching at the file level) και του τρόπου με τον οποίο τα αιτήματα ανάγνωσης (disk requests) καταμερίζονται στους φυσικούς δίσκους (striped storage). Η αύξηση του prefetching size (2^0, 2^1, 2^2, ...) προκαλεί "split" αιτήματα που επιβαρύνουν το φυσικό επίπεδο αποθήκευσης, αναδεικνύοντας τη σημασία της ισορροπίας μεταξύ λογικής και φυσικής προφόρτωσης. (Baek & Park, 2009)

2.3.2 Predictive και statistical prefetching

Πέρα από τις σειριακές πολιτικές, η έρευνα έχει εστιάσει σε προβλεπτικά (predictive) prefetchers, τα οποία χρησιμοποιούν στατιστικά ή πιθανοτικά μοντέλα για να προβλέψουν τα επόμενα δεδομένα. Οι τεχνικές αυτές εκμεταλλεύονται τη συχνότητα και τη συσχέτιση των προηγούμενων προσπελάσεων, με στόχο να εντοπίσουν επαναλαμβανόμενα μοτίβα (Yang et al., 2025).

Μια δημοφιλής μορφή είναι οι Markov-based prefetchers, όπου κάθε αντικείμενο συνδέεται με πιθανές επόμενες προσπελάσεις, με βάση τις συχνότητες εμφάνισης στα traces. Έτσι, η πρόβλεψη βασίζεται σε πιθανότητες μετάβασης (transition probabilities). Αυτή η ιδέα μπορεί να προσαρμοστεί σε περιβάλλοντα blockchain, όπου κάθε smart contract έχει πιθανότητα να καλέσει συγκεκριμένα storage slots (Jung et al., 2023).

Αντίστοιχα, τα correlation-based prefetchers μαθαίνουν σχέσεις μεταξύ μη διαδοχικών αντικειμένων: αν το A προσπελαύνεται, τότε υπάρχει πιθανότητα *p* να προσπελαστεί και το B (Hasan et al., 2020). Αυτή η προσέγγιση θα μπορούσε να μοντελοποιήσει τη συσχέτιση μεταξύ λειτουργιών εντός ενός contract (π.χ. balances και allowances σε ERC-20 tokens).

Στην παρούσα εργασία, η πολιτική Per-Contract Top-K εντάσσεται σε αυτή την κατηγορία: στηρίζεται σε στατιστική ανάλυση συχνότητας ανά συμβόλαιο, προφορτώνοντας τα πιο πιθανά slots που θα χρησιμοποιηθούν, χωρίς να βασίζεται στη χρονική σειρά των blocks (Hias et al., 2024).

2.3.3 Learning-based prefetching

Τα τελευταία χρόνια, η έρευνα στο πεδίο έχει στραφεί προς machine-learning-based prefetchers, που επιχειρούν να μάθουν δυναμικά τα μοτίβα πρόσβασης μέσω εποπτευόμενης ή ενισχυτικής μάθησης (Yang et al., 2025). Τέτοιες προσεγγίσεις χρησιμοποιούνται ήδη σε distributed storage συστήματα και data centers, επιτυγχάνοντας σημαντική βελτίωση σε προβλέψιμα workloads.

- Τα supervised learning prefetchers εκπαιδεύονται σε ιστορικά traces για να αναγνωρίζουν πρότυπα προσπέλασης. Συνήθως χρησιμοποιούν decision trees, gradient boosting ή νευρωνικά δίκτυα για να προβλέψουν το επόμενο request (Hasan et al., 2020).
- Τα reinforcement learning prefetchers προσαρμόζονται σε πραγματικό χρόνο, επιλέγοντας δράσεις (π.χ. ποια δεδομένα να προφορτώσουν) που μεγιστοποιούν ένα reward function, όπως το hit rate ή η μείωση του latency (Jung et al., 2023).

Αν και τέτοια μοντέλα δεν έχουν ακόμη εφαρμοστεί εκτενώς στο blockchain, αποτελούν προοπτική μελλοντικής έρευνας.

2.4 Αρχή λειτουργίας του read-ahead

Το read-ahead είναι μια *προληπτική στρατηγική ανάγνωσης* που βασίζεται στην πρόβλεψη ότι οι επόμενες αναγνώσεις θα αφορούν κοντινά δεδομένα σε σχέση με το τρέχον σημείο πρόσβασης. Κατά την εκτέλεση ενός αιτήματος ανάγνωσης (*read request*) από το δίσκο, το σύστημα όχι μόνο ανακτά το ζητούμενο block, αλλά και ένα σύνολο από τα επόμενα διαδοχικά blocks, τα οποία αποθηκεύει προσωρινά στη μνήμη. Έτσι, όταν οι επόμενες αναγνώσεις ζητηθούν, τα δεδομένα βρίσκονται ήδη διαθέσιμα στο cache, μειώνοντας δραστικά το *I/O latency* (Denning, 1980; Ruemmler & Wilkes, 1994).

Η τεχνική αυτή εκμεταλλεύεται κυρίως τη χωρική τοπικότητα (spatial locality), δηλαδή την τάση των διεργασιών να προσπελαύνουν συνεχόμενες περιοχές δεδομένων. Σε περιβάλλοντα όπου τα δεδομένα αποθηκεύονται σε σειρές (π.χ. σε αρχεία, πίνακες ή blocks), η χωρική τοπικότητα μπορεί να αυξήσει το hit rate της cache κατά τάξεις μεγέθους.

2.4.1 Εφαρμογές σε λειτουργικά συστήματα

Τα λειτουργικά συστήματα υλοποιούν το read-ahead εδώ και δεκαετίες ως τμήμα του *Virtual File System (VFS)*. Για παράδειγμα, το Linux kernel διαθέτει τον μηχανισμό readahead(), ο οποίος παρακολουθεί το μοτίβο πρόσβασης σε αρχεία και προσαρμόζει δυναμικά το παράθυρο προφόρτωσης ανάλογα με τη συμπεριφορά του προγράμματος (Love, 2010).

Όταν ανιχνευθεί σειριακή ανάγνωση, ο kernel ξεκινά με ένα μικρό prefetch window (π.χ. 128 KB) και το αυξάνει προοδευτικά αν συνεχιστεί η σειριακή πρόσβαση, έως ένα μέγιστο όριο. Αν όμως εντοπιστεί τυχαία ή μη προβλέψιμη πρόσβαση, το read-ahead απενεργοποιείται προσωρινά για να αποφευχθεί η σπατάλη πόρων.

Αυτή η προσαρμοστικότητα είναι το κλειδί της επιτυχίας του read-ahead σε λειτουργικά συστήματα: ενεργοποιείται μόνο όταν υπάρχει επαρκής ένδειξη τοπικότητας. Αντίθετα, σε περιβάλλοντα όπως το Ethereum, όπου η τοπικότητα είναι αδύνατο να ανιχνευθεί αυτόματα χωρίς ανάλυση των συμβολαίων, απαιτείται *εξωτερικός προγνωστικός μηχανισμός*, δηλαδή μια πολιτική prefetching προσαρμοσμένη στο execution layer.

2.4.2 Read-ahead σε block storage και databases

Σε επίπεδο block storage, το read-ahead υλοποιείται κυρίως από τον I/O scheduler και τους device drivers, οι οποίοι προφορτώνουν συνεχόμενα blocks δεδομένων στον buffer cache του λειτουργικού. Οι συσκευές SSD και NVMe μπορούν να επωφεληθούν περιορισμένα από τέτοια πολιτική, λόγω του χαμηλού access time, αλλά η τεχνική παραμένει κρίσιμη σε workloads με υψηλή σειριακότητα (Ruemmler & Wilkes, 1994).

Σε βάσεις δεδομένων, η τεχνική επεκτείνεται υπό τη μορφή sequential scan prefetching ή index prefetching. Συστήματα όπως το PostgreSQL και το RocksDB χρησιμοποιούν read-ahead buffers ώστε να επιταχύνουν μεγάλα range queries ή compactions. Ειδικά το RocksDB, το οποίο χρησιμοποιείται από τους Ethereum clients (όπως Nethermind), ενσωματώνει μηχανισμό table-level readahead για τη φόρτωση των SST files, μειώνοντας τα random I/O operations (Kurisaka et al., 2025).

Αυτές οι υλοποιήσεις όμως βασίζονται στην υπόθεση ότι τα δεδομένα που αναζητούνται είναι διαδοχικά στο δίσκο. Στο Ethereum, τα state objects (accounts και slots) είναι κρυπτογραφικά κατακερματισμένα (hashed), με αποτέλεσμα η φυσική τους θέση στο storage να μην σχετίζεται με τη λογική τους γειτνίαση. Επομένως, η κλασική read-ahead τεχνική δεν μπορεί να αποδώσει χωρίς κάποια *λογική ομαδοποίηση* ή *προγνωστική πληροφορία* (Hias et al., 2024).

2.4.3 Περιορισμοί του παραδοσιακού read-ahead

Παρά τα αποδεδειγμένα οφέλη του, το read-ahead παρουσιάζει ορισμένους σημαντικούς περιορισμούς:

1. Ανεπαρκής προσαρμοστικότητα: Οι περισσότερες υλοποιήσεις προϋποθέτουν σταθερά μοτίβα πρόσβασης. Όταν το workload αλλάζει δυναμικά, η απόδοση πέφτει.

2. Έλλειψη προβλεπτικότητας σε random workloads: Το prefetch μπορεί να προκαλέσει σπατάλη (waste) όταν τα δεδομένα που προφορτώθηκαν δεν χρησιμοποιούνται ποτέ.
3. Μη αξιοποίηση στατιστικής πληροφορίας: Οι περισσότερες παραδοσιακές πολιτικές βασίζονται σε σειριακότητα, όχι σε ιστορική συχνότητα ή συσχετίσεις δεδομένων (Hasan et al., 2020).

Προκειμένου να προσαρμοστεί η ιδέα του read-ahead στα ιδιαίτερα χαρακτηριστικά του Ethereum, η πολιτική ReadAhead που υλοποιείται στο πλαίσιο της παρούσας εργασίας λειτουργεί ως *logical readahead*, εστιάζοντας στη χρονική ακολουθία των blocks και όχι στην τοπολογική διάταξη των δεδομένων στο δίσκο (Ergen et al., 2024; Kurisaka et al., 2025).

2.4.4 Η προσαρμογή της αρχής Read-Ahead στο Ethereum

Η εφαρμογή της τεχνικής read-ahead στο Ethereum προϋποθέτει την αναγνώριση μοτίβων πρόσβασης σε block-level. Επειδή κάθε block περιλαμβάνει γνωστό εκ των προτέρων σύνολο συναλλαγών, ένας Ethereum client μπορεί να προβλέψει ποια storage slots θα απαιτηθούν στο άμεσο μέλλον. Αυτό επιτρέπει την προφόρτωση αυτών των δεδομένων πριν από την έναρξη της εκτέλεσης του επόμενου block. (Hias et al., 2024).

Η πολιτική ReadAhead της παρούσας εργασίας βασίζεται σε αυτήν την αρχή. Αντί να προφορτώνει συνεχόμενα bytes ή blocks, προφορτώνει state entries (accounts και storage slots) από τα επόμενα N blocks. Με αυτόν τον τρόπο, αξιοποιεί τη χρονική τοπικότητα μεταξύ blocks. Ειδικότερα, κάθε φορά που εκτελείται ένα block B, το σύστημα διαβάζει το *access trace* των επόμενων blocks $B+1$ ... $B+N$ και προφορτώνει τα accounts και storage slots που πρόκειται να ζητηθούν.

2.5 Συναφείς εργασίες για blockchain state management, prefetch policies και προγνωστικά μοντέλα

Η έρευνα γύρω από τη βελτιστοποίηση της πρόσβασης στο state των blockchain συστημάτων έχει γνωρίσει σημαντική πρόοδο τα τελευταία χρόνια, καθώς το μέγεθος και η πολυπλοκότητα του Ethereum έχουν αυξηθεί εκθετικά. Το πρόβλημα του *state bloat* και της χαμηλής αποδοτικότητας των clients έχει αναγνωριστεί ως κεντρικό εμπόδιο για την επεκτασιμότητα και τη βιωσιμότητα του δικτύου (Buterin et al., 2021; Kurisaka et al., 2025). Ωστόσο, οι περισσότερες προσπάθειες εστιάζουν σε λύσεις επιπέδου πρωτοκόλλου (όπως *state expiry*, *Verkle trees* ή *stateless clients*), αφήνοντας σχετικά ανεξερεύνητη τη διάσταση της βελτιστοποίησης πρόσβασης μέσω caching και prefetching.

Η παρούσα ενότητα παρουσιάζει τις βασικές κατηγορίες συναφών ερευνητικών εργασιών που συνθέτουν το πλαίσιο στο οποίο εντάσσεται η παρούσα μελέτη.

2.5.1 Έρευνες για το blockchain state management

Η διαχείριση του *state* στο Ethereum έχει αποτελέσει αντικείμενο εντατικής έρευνας, κυρίως λόγω του φαινομένου του state growth, δηλαδή της συνεχούς αύξησης του συνολικού όγκου των δεδομένων που διατηρεί κάθε κόμβος. Σύμφωνα με τους Buterin et al. (2021), το πλήρες state του Ethereum ξεπερνά τα 600 GB, ενώ αυξάνεται με ρυθμό ~15 GB/μήνα. Αυτή η αύξηση καθιστά όλο και πιο δαπανηρή τη λειτουργία ενός *full node*, οδηγώντας σε συγκέντρωση ισχύος στους λίγους τεχνικά ικανούς παρόχους υποδομών (Hias et al., 2024).

Οι κύριες προτάσεις που έχουν εξεταστεί για την αντιμετώπιση του προβλήματος είναι:

- State pruning: περιοδική διαγραφή παλαιών ή μη χρησιμοποιούμενων καταστάσεων (Ergen et al., 2024).
- State expiry: εφαρμογή χρονικού ορίου ζωής (TTL) για τα state entries.
- Verkle trees: αντικατάσταση της Merkle Patricia Trie με πιο συμπαγή δομή που επιτρέπει αποτελεσματικότερα αποδεικτικά στοιχεία (Buterin et al., 2021).
- Stateless clients: μεταφορά της ευθύνης για την αποθήκευση state δεδομένων από τους κόμβους στους χρήστες μέσω *witness data* (Ethereum Foundation, 2023).

Παρότι οι λύσεις αυτές μειώνουν το αποθηκευτικό βάρος και επιταχύνουν το synchronization, δεν αντιμετωπίζουν άμεσα το πρόβλημα της πρόσβασης σε δεδομένα κατά την εκτέλεση συναλλαγών. Η απόδοση του execution layer παραμένει περιορισμένη από το *latency* των I/O λειτουργιών, κάτι που καθιστά αναγκαία την έρευνα σε τεχνικές caching και prefetching σε επίπεδο client (Jung et al., 2023).

2.5.2 Ερευνητικές προσεγγίσεις σε caching και prefetching στο blockchain

Οι μελέτες που επικεντρώνονται σε μηχανισμούς caching στα blockchain συστήματα είναι ελάχιστες σε σύγκριση με εκείνες που αφορούν την αρχιτεκτονική των πρωτοκόλλων.

Οι Li et al. (2021) μελέτησαν την απόδοση *storage caches* σε blockchain κόμβους, καταδεικνύοντας ότι οι πολιτικές LRU και LFU αποτυγχάνουν να αξιοποιήσουν την τοπικότητα, καθώς τα access patterns είναι μη στατικά.

Αν και υπάρχουν πειραματικές προσπάθειες βελτίωσης μέσω adaptive caching, οι οποίες προσαρμόζουν το μέγεθος ή τη στρατηγική cache βάσει μετρήσεων runtime (π.χ. Adaptive Replacement Cache, Megiddo & Modha, 2003), η προσέγγιση αυτή παραμένει *αντιδραστική*: ανταποκρίνεται στις προηγούμενες προσπελάσεις χωρίς να προβλέπει τις επόμενες.

Η παρούσα εργασία εισάγει δυναμικές πολιτικές προφόρτωσης, οι οποίες προσαρμόζονται είτε χρονικά (ReadAhead) είτε στατιστικά (Per-Contract Top-K) στα χαρακτηριστικά των συναλλαγών και των συμβολαίων.

2.5.3 Προγνωστικά μοντέλα και στατιστικές πολιτικές prefetching

Σημαντική πρόοδος έχει επιτευχθεί διεθνώς στις τεχνικές προγνωστικής προφόρτωσης (predictive prefetching), οι οποίες χρησιμοποιούν στατιστικά ή machine learning μοντέλα για να αναγνωρίσουν τα μοτίβα πρόσβασης σε δεδομένα.

Η βασική ιδέα είναι η ανάλυση των access traces ώστε να εντοπιστούν επαναλαμβανόμενες ακολουθίες (patterns) και να δημιουργηθούν πιθανότητες μετάβασης μεταξύ αντικειμένων. Οι προγνωστικές προσεγγίσεις ταξινομούνται σε τέσσερις κύριες κατηγορίες:

1. Frequency-based models, όπου προφορτώνονται τα πιο συχνά χρησιμοποιούμενα δεδομένα (όπως η Top-K πολιτική της παρούσας εργασίας).
2. Markov-based models, όπου κάθε πρόσβαση προβλέπει την επόμενη με βάση τις πιθανότητες μετάβασης.
3. Correlation-based models, που εντοπίζουν μη γραμμικές σχέσεις μεταξύ αντικειμένων.
4. Machine-learning models, που εκπαιδεύονται σε μεγάλα ιστορικά traces (Yang et al., 2025).

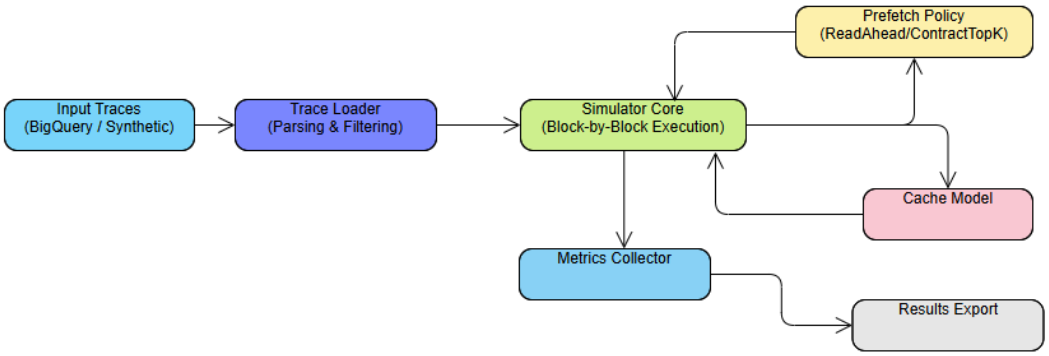
Στο blockchain πλαίσιο, η εφαρμογή τέτοιων τεχνικών βρίσκεται ακόμη σε πρώιμο στάδιο. Οι Xu et al. (2022) πειραματίστηκαν με έναν *Markov-chain prefetcher* για το Ethereum, αποδεικνύοντας ότι η πρόβλεψη σε επίπεδο contract-level access sequence μπορεί να αυξήσει το hit rate κατά 25% σε σχέση με το baseline caching. Ωστόσο, η πολυπλοκότητα και ο υπολογιστικός φόρτος καθιστούν τέτοιες μεθόδους δύσκολες για ενσωμάτωση σε πραγματικούς clients.

Η πολιτική Per-Contract Top-K που παρουσιάζεται στην παρούσα εργασία αποτελεί πρακτική, ελαφριά εκδοχή ενός predictive prefetcher, καθώς εκμεταλλεύεται τα στατιστικά ανά συμβόλαιο χωρίς να απαιτεί μεγάλες computational δαπάνες. Ενσωματώνει έτσι τα πλεονεκτήματα της απλότητας των στατικών μεθόδων με μέρος της ευφυΐας των προγνωστικών μοντέλων.

Κεφάλαιο 3 – Μεθοδολογία και Σχεδίαση του Προσομοιωτή

3.1 Γενική αρχιτεκτονική συστήματος και ροή δεδομένων

Η αρχιτεκτονική του προσομοιωτή σχεδιάστηκε με σκοπό να μετρά με ακρίβεια την απόδοση διαφόρων πολιτικών προανάκτησης (prefetch) στο **state** του Ethereum. Η φιλοσοφία του συστήματος βασίζεται στη λογική της *αρθρωτής σχεδίασης* (modular architecture), ώστε κάθε υποσύστημα, όπως ο μηχανισμός φόρτωσης δεδομένων, το μοντέλο cache, οι πολιτικές προφόρτωσης και οι μηχανισμοί μετρήσεων, να μπορεί να αναπτυχθεί, να αντικατασταθεί ή να επεκταθεί ανεξάρτητα από τα υπόλοιπα. Με αυτόν τον τρόπο, η προσομοίωση παραμένει εύκολα παραμετροποιήσιμη, προσαρμόσιμη σε διαφορετικά workloads και απολύτως επαναλήψιμη, γεγονός κρίσιμο για την επιστημονική αξιοπιστία των αποτελεσμάτων.



Εικόνα 6. Υψηλού επιπέδου αρχιτεκτονική του προσομοιωτή και ροή δεδομένων μεταξύ Loader, Simulator, Policies, Cache και Metrics Collector.

Στην καρδιά του συστήματος βρίσκεται ο **πυρήνας της προσομοίωσης**, ο οποίος εκτελεί κάθε block του Ethereum βήμα προς βήμα, προσομοιώνοντας τις λειτουργίες ανάγνωσης και εγγραφής του state (SLOAD/SSTORE). Για κάθε block, ο προσομοιωτής ανακτά μια λίστα προσπελάσεων από το trace, η οποία περιλαμβάνει τα πεδία {ts, block_number, tx_id, op, account, slot}. Αυτή η λίστα τροφοδοτείται στον μηχανισμό cache, ο οποίος ελέγχει αν το ζητούμενο αντικείμενο βρίσκεται ήδη στη μνήμη. Αν το αντικείμενο υπάρχει, χαρακτηρίζεται ως *hit* και επιβαρύνει τον συνολικό χρόνο εκτέλεσης μόνο με τον καθορισμένο χρόνο ανάγνωσης (hit latency). Αν δεν υπάρχει, προκαλείται *miss*, το οποίο συνεπάγεται ανάγνωση από τον αποθηκευτικό χώρο και επιπλέον καθυστέρηση (miss latency).

Η **πολιτική προφόρτωσης** (ReadAhead, ContractTopK) ενεργοποιείται στην αρχή κάθε block. Ο προσομοιωτής καλεί την πολιτική και της επιτρέπει να προτείνει ποια αντικείμενα θα προφορτωθούν εκ των προτέρων. Τα προφορτωμένα αντικείμενα τοποθετούνται προσωρινά στη μνήμη cache, επιβαρύνοντας τον συνολικό χρόνο με ένα χαμηλότερο *prefetch latency*,

καθώς θεωρείται ότι η ανάγνωσή τους γίνεται ασύγχρονα και με μειωμένο κόστος σε σχέση με τα κανονικά misses.

Η ροή των δεδομένων ακολουθεί μία προκαθορισμένη ακολουθία βημάτων. Αρχικά, ο **Loader** φορτώνει τα traces που προέρχονται είτε από πραγματικά δεδομένα του Ethereum μέσω BigQuery, είτε από συνθετικά workloads που δημιουργήθηκαν για τις ανάγκες της μελέτης. Κατόπιν, ο **Simulator** αναλαμβάνει να επεξεργαστεί τα blocks ένα προς ένα. Πριν ξεκινήσει η εκτέλεση κάθε block, καλεί την ενεργή πολιτική προφόρτωσης, η οποία αποφασίζει ποια δεδομένα αξίζει να φορτωθούν προκαταβολικά. Ο Simulator ελέγχει το προτεινόμενο σύνολο δεδομένων και κατόπιν εκτελεί την προφόρτωση μέσω του μηχανισμού cache. Κατά την εκτέλεση των συναλλαγών του block, κάθε πρόσβαση στο state ελέγχεται εάν βρίσκεται ήδη στη μνήμη (hit) ή αν απαιτείται ανάγνωση από το storage (miss).

Μετά την ολοκλήρωση ενός block, ο προσομοιωτής υπολογίζει τις κύριες μετρικές απόδοσης. Η **κάλυψη (coverage)** ορίζεται ως το ποσοστό των προφορτωμένων αντικειμένων που τελικά χρησιμοποιήθηκαν, ενώ το **waste** αντιπροσωπεύει το ποσοστό των προφορτωμένων δεδομένων που δεν χρησιμοποιήθηκαν ποτέ. Ο συνδυασμός αυτών των δύο μεγεθών, μαζί με τον **ρυθμό επιτυχιών (hit rate)** και το συνολικό **speedup** σε σχέση με τη βασική γραμμή αναφοράς (baseline), παρέχει μια πλήρη εικόνα της αποτελεσματικότητας κάθε πολιτικής.

Η **μοντελοποίηση των καθυστερήσεων (latency model)** επιτρέπει την προσαρμογή της προσομοίωσης σε διαφορετικά περιβάλλοντα υλικού. Δύο βασικά προφίλ χρησιμοποιούνται: το *NVMe-like*, που προσομοιώνει γρήγορες αποθηκευτικές μονάδες με μικρούς χρόνους πρόσβασης, και το *SATA-like*, που προσομοιώνει παραδοσιακούς σκληρούς δίσκους με μεγαλύτερα latencies.

Η επικοινωνία μεταξύ των επιμέρους υποσυστημάτων πραγματοποιείται μέσω σαφώς ορισμένων διεπαφών (interfaces). Οι πολιτικές προφόρτωσης υλοποιούν κοινές συναρτήσεις οι οποίες επιτρέπουν στον προσομοιωτή να λειτουργεί ανεξάρτητα από τον εσωτερικό μηχανισμό κάθε πολιτικής. Ο μηχανισμός cache παρέχει συναρτήσεις διαχείρισης της cache, ενώ το υποσύστημα των μετρικών αναλαμβάνει την καταγραφή των γεγονότων (hits, misses, prefetches) και την εξαγωγή τους σε μορφή CSV. Στο τέλος κάθε εκτέλεσης, ο orchestrator συλλέγει όλα τα αποτελέσματα και τα αποθηκεύει σε αρχεία CSV και plots για περαιτέρω ανάλυση.

Η συνολική ροή δεδομένων μπορεί να συνοψιστεί ως εξής: τα δεδομένα εισέρχονται μέσω του Loader, μετατρέπονται σε access lists ανά block, εκτελούνται στον Simulator με την εκάστοτε πολιτική προφόρτωσης, αποθηκεύονται οι μετρήσεις απόδοσης και, τέλος, παράγονται γραφήματα και συγκεντρωτικά αποτελέσματα. Με τον τρόπο αυτό, η προσομοίωση παραμένει

πλήρως ελεγχόμενη, επιτρέποντας την ακριβή μέτρηση του οφέλους που προκύπτει από κάθε πολιτική.

Η σχεδίαση της αρχιτεκτονικής του συστήματος βασίστηκε σε δύο αρχές: **αναπαραγωγιμότητα** και **επεκτασιμότητα**. Η αναπαραγωγιμότητα επιτυγχάνεται με σταθερά configuration αρχεία και καταγραφή όλων των παραμέτρων σε αρχεία configs/*.yaml και results/*.json. Η επεκτασιμότητα επιτυγχάνεται με τον διαχωρισμό των modules, ώστε μελλοντικά να μπορούν να ενσωματωθούν πιο σύνθετες τεχνικές.

3.2 Δομή modules: loader, cache, simulator, metrics, plots

Βάση της αρχιτεκτονικής είναι το **module του προσομοιωτή (simulator)**, το οποίο αποτελεί τον εκτελεστικό πυρήνα του συστήματος. Ο simulator είναι υπεύθυνος για την αλληλεπίδραση όλων των άλλων υποσυστημάτων: διαβάζει τα δεδομένα εισόδου, ενεργοποιεί τις πολιτικές προφόρτωσης, ελέγχει τις προσπελάσεις στο state, ενημερώνει τις μετρικές και αποθηκεύει τα αποτελέσματα. Η λειτουργία του είναι πλήρως παραμετροποιήσιμη μέσω configuration αρχείων YAML, τα οποία καθορίζουν το workload, το latency profile και τη χωρητικότητα της cache. Ο simulator εκτελεί το πείραμα block προς block, δημιουργώντας ένα πιστό χρονικό μοντέλο του τρόπου με τον οποίο ένας Ethereum client θα προσπελάσει το state του κατά την εκτέλεση συναλλαγών.

Το module **loader** αποτελεί το σημείο εισόδου των δεδομένων. Ο ρόλος του είναι να διαβάζει και να προεπεξεργάζεται τα traces που χρησιμοποιούνται στην προσομοίωση. Αυτά τα traces προέρχονται είτε από πραγματικά δεδομένα του Ethereum mainnet, εξαγόμενα μέσω Google BigQuery, είτε από συνθετικά datasets που έχουν δημιουργηθεί για πειραματικούς σκοπούς.

Για την απόκτηση πραγματικών δεδομένων χρησιμοποιήθηκε η πλατφόρμα **Google BigQuery**, η οποία παρέχει δημόσια datasets του **Ethereum Mainnet**. Η διαδικασία άντλησης ξεκίνησε με τη σύνταξη ερωτημάτων που στοχεύουν στους πίνακες bigquery-public-data.crypto_ethereum.traces και transactions. Τα queries φιλτράρουν τις εγγραφές βάσει τύπου κλήσης (call_type), hash συναλλαγής και block number, ώστε να παραληφθούν μόνο οι εγγραφές που αντιστοιχούν σε state accesses.

Τα αποτελέσματα αυτών των ερωτημάτων εξήχθησαν σε μορφή JSON Lines (.jsonl), όπου κάθε γραμμή περιέχει τα πεδία block_number, transaction_index, from_address, to_address, και type. Το παραγόμενο αρχείο φορτώθηκε απευθείας στον loader, ο οποίος φροντίζει να μετατρέψει τα δεδομένα σε ενιαία access lists ανά block και να τα ταξινομήσει χρονολογικά, διασφαλίζοντας ότι η σειρά εκτέλεσης αντικατοπτρίζει τη ροή των πραγματικών συναλλαγών

του δικτύου. Παράλληλα, εφαρμόζεται μηχανισμός καθαρισμού δεδομένων (data sanitization), που απορρίπτει διπλές εγγραφές, άκυρα slots ή ατελείς transactions.

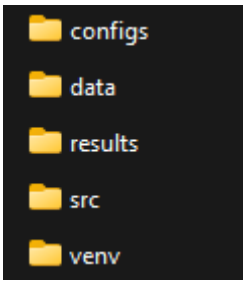
Επιπλέον, κρίθηκε απαραίτητη η δημιουργία **συνθετικών datasets (synthetic workloads)**, ώστε να μπορούν να εκτελεστούν ελεγχόμενα πειράματα με συγκεκριμένα μοτίβα πρόσβασης. Για τον σκοπό αυτόν αναπτύχθηκε ειδικός generator εντός του προσομοιωτή, ο οποίος δημιουργεί τεχνητά traces με διαφορετικά χαρακτηριστικά τοπικότητας και πολυπλοκότητας. Τα synthetic data οργανώνονται σε τρεις κατηγορίες: *storage-heavy* (με συχνές προσπελάσεις σε SLOAD/SSTORE operations), *storage-light* (με αραιές προσπελάσεις) και *generic workloads* (συνδυασμός των δύο).

Η ύπαρξη και των δύο τύπων δεδομένων (πραγματικών και συνθετικών) επιτρέπει την ακριβή αξιολόγηση της αποδοτικότητας των πολιτικών σε περιβάλλοντα με διαφορετικά επίπεδα προβλεψιμότητας και φορτίου. Τα δεδομένα του BigQuery παρέχουν το ρεαλιστικό, μη ντετερμινιστικό προφίλ του Ethereum mainnet, ενώ τα synthetic traces επιτρέπουν την απομόνωση παραμέτρων και τη μελέτη συγκεκριμένων φαινομένων.

Ο **μηχανισμός cache (cache module)** είναι υπεύθυνος για τη διαχείριση της προσωρινής αποθήκευσης δεδομένων και την προσομοίωση του τρόπου που λειτουργεί η μνήμη σε έναν πραγματικό Ethereum client. Υλοποιεί μια παραλλαγή της πολιτικής LRU (Least Recently Used) και μπορεί να διαμορφωθεί με διαφορετική χωρητικότητα, latency profile και prefetch strategy. Κάθε φορά που ο simulator ζητά πρόσβαση σε κάποιο στοιχείο του state, το cache module ελέγχει αν αυτό βρίσκεται ήδη στη μνήμη. Αν ναι, καταγράφεται ένα hit και η καθυστέρηση πρόσβασης είναι μικρή. Αν όχι, προκαλείται miss, το οποίο αντιστοιχεί σε ανάγνωση από το storage και συνεπάγεται υψηλότερο latency. Επιπλέον, το cache module διαχειρίζεται τις λειτουργίες προφόρτωσης που προτείνουν οι πολιτικές prefetch, τοποθετώντας εκ των προτέρων δεδομένα στη μνήμη με μικρότερο κόστος πρόσβασης. Με αυτόν τον τρόπο επιτρέπει τη μέτρηση του coverage και του waste για κάθε πείραμα.

Το **module των μετρικών (metrics)** συγκεντρώνει και αναλύει τα στατιστικά που προκύπτουν κατά τη διάρκεια της προσομοίωσης. Καταγράφει το πλήθος των hits, misses, prefetches, τις καθυστερήσεις πρόσβασης και τον συνολικό χρόνο εκτέλεσης. Με βάση αυτά τα δεδομένα, υπολογίζει δείκτες απόδοσης όπως το hit rate (ποσοστό επιτυχιών), το coverage (ποσοστό επιτυχών προφορτώσεων), το waste (ποσοστό άχρηστων προφορτώσεων) και το speedup σε σχέση με το baseline. Στο τέλος κάθε εκτέλεσης, το module εξάγει τα αποτελέσματα σε μορφή CSV, τα οποία μπορούν να χρησιμοποιηθούν από εργαλεία ανάλυσης και απεικόνισης δεδομένων.

Τέλος, το **module plots** αναλαμβάνει τη γραφική απεικόνιση των αποτελεσμάτων. Χρησιμοποιεί βιβλιοθήκες όπως το matplotlib και το pandas για να δημιουργήσει αυτόματα γραφήματα και διαγράμματα από τα αρχεία CSV που παράγει το metrics module.



Εικόνα 7. Δομή του project directory του προσομοιωτή. Εμφανίζονται οι βασικοί φάκελοι που απαρτίζουν το σύστημα: ο φάκελος configs/ περιλαμβάνει τα αρχεία ρυθμίσεων σε μορφή YAML, ο φάκελος data/ περιέχει τα traces εισόδου, ενώ ο φάκελος results/ αποθηκεύει τα αρχεία εξόδου και τα συγκεντρωτικά αποτελέσματα των πειραμάτων.

3.3 Μοντέλο cache, παραμετροποίηση latencies (NVMe-like, SATA-like) και χωρητικότητας

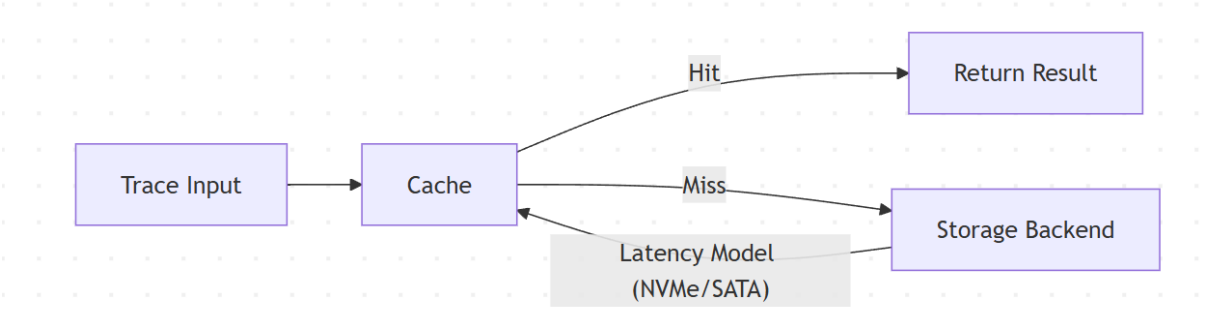
Το μοντέλο cache είναι υπεύθυνο για την προσωρινή αποθήκευση των δεδομένων που ανακτώνται από το blockchain state και για την αποτίμηση της απόδοσης των πολιτικών προφόρτωσης (prefetching policies). Η σχεδίαση του μοντέλου ακολουθεί μια αφαιρετική προσέγγιση (abstraction model), η οποία επιτρέπει την προσομοίωση διαφορετικών τύπων αποθηκευτικών συστημάτων χωρίς την ανάγκη πρόσβασης σε πραγματικό υλικό. Αντί να μετράται ο φυσικός χρόνος ανάγνωσης/εγγραφής, εφαρμόζεται ένα **παραμετροποιήσιμο latency model**, το οποίο αποδίδει καθυστερήσεις ανάλογες με τα προφίλ επιδόσεων δίσκων τύπου **NVMe** και **SATA**, προσαρμοσμένα στα χαρακτηριστικά των Ethereum clients.

Η λογική λειτουργίας της cache βασίζεται στη στρατηγική **Least Recently Used (LRU)**, που επιλέχθηκε για την απλότητά της και τη συνάφειά της με τις περισσότερες πραγματικές υλοποιήσεις caching επιπέδου συστήματος. Κάθε προσπέλαση (SLOAD ή SSTORE) ελέγχει αν το ζητούμενο state entry βρίσκεται ήδη στη cache· αν όχι, καταγράφεται ως *miss* και η ανάγνωση γίνεται από το “storage backend” (προσομοιωμένο ως key-value store). Κατόπιν, το στοιχείο εισάγεται στην cache και το latency της πράξης προστίθεται στο συνολικό χρόνο εκτέλεσης. Το μέγεθος της cache καθορίζεται δυναμικά μέσω των παραμέτρων των YAML αρχείων ρυθμίσεων.

Η παραμετροποίηση των **latencies** υλοποιείται μέσω συντελεστών που αναπαριστούν τον μέσο χρόνο προσπέλασης δεδομένων για κάθε προφίλ αποθήκευσης. Για παράδειγμα, στο προφίλ *NVMe-like* ορίζεται miss-latency ίσο με 40 μs, ώστε να προσομοιωθεί η συμπεριφορά μοντέρνων SSDs υψηλής ταχύτητας. Αντίστοιχα, στο *SATA-like* προφίλ χρησιμοποιείται miss-latency 400 μs, που αντιστοιχεί σε πιο συμβατικές μονάδες αποθήκευσης, όπως SATA SSDs.

Επιπλέον, η **χωρητικότητα της cache** μπορεί να τροποποιηθεί ανάλογα με το είδος του workload. Στα *storage-heavy* σενάρια, όπου τα smart contracts εκτελούν συνεχείς προσπελάσεις σε πλήθος storage slots, η αύξηση της cache επιτρέπει την κατακράτηση μεγαλύτερου αριθμού πρόσφατων entries. Αντίθετα, στα *storage-light* workloads, η υπερβολική αύξηση της χωρητικότητας δεν προσφέρει ουσιαστικό κέρδος, καθώς οι προσπελάσεις είναι σποραδικές και μη επαναλαμβανόμενες.

Τέλος, η σχεδίαση του cache module υποστηρίζει παραμετρική επέκταση, επιτρέποντας τη δοκιμή νέων τύπων αποθήκευσης (π.χ. persistent memory, hybrid SSD-HDD systems).



Εικόνα 8. Σχηματική απεικόνιση του μοντέλου cache και της ροής δεδομένων. Οι καθυστερήσεις προσομοιώνονται μέσω latency profiles, ενώ οι προσπελάσεις ταξινομούνται ως hits ή misses.

3.4 Πολιτικές προανάκτησης: ορισμοί και παράμετροι

Οι πολιτικές προανάκτησης (*prefetching policies*) καθορίζουν τον τρόπο με τον οποίο το σύστημα προβλέπει και προφορτώνει δεδομένα από το blockchain state. Σκοπός τους είναι η μείωση των καθυστερήσεων ανάγνωσης που προκαλούνται από τις συχνές προσπελάσεις στο storage των smart contracts. Η σχεδίασή τους βασίστηκε στη λογική ότι η πρόσβαση στο state του Ethereum δεν είναι τυχαία αλλά παρουσιάζει επαναλαμβανόμενα μοτίβα, τόσο σε διαδοχικά blocks όσο και στο εσωτερικό των ίδιων των συμβολαίων.

Η πολιτική **ReadAhead** στηρίζεται στην κλασική αρχή της χρονικής τοπικότητας (*temporal locality*). Ο αλγόριθμος λοιπόν προφορτώνει έναν αριθμό από επόμενα blocks (π.χ. *blocks_ahead = 1 ή 2*) και εισάγει στη cache τα storage entries που αναμένονται να χρειαστούν. Με τον τρόπο αυτό, η πολιτική επιτυγχάνει μείωση του χρόνου εκτέλεσης.

Αντίθετα, η πολιτική **ContractTopK** υιοθετεί μια στατιστική, *data-driven* προσέγγιση, βασισμένη στη συχνότητα πρόσβασης των storage slots ανά συμβόλαιο. Για κάθε smart contract, κατασκευάζεται ένα προφίλ προσπελάσεων (π.χ. data/topk_slots.json) που αποτυπώνει ποια slots χρησιμοποιούνται συχνότερα. Ο μηχανισμός αναλύει αυτά τα προφίλ offline και, κατά την εκτέλεση της προσομοίωσης, προφορτώνει τα K πιο πιθανά slots που θα χρειαστούν. Η τιμή του **K** (π.χ. *K=5*) αντιπροσωπεύει το μέγεθος του prefetch set και αποτελεί κρίσιμη παράμετρο, καθώς ελέγχει την ισορροπία μεταξύ επιτυχούς κάλυψης (*coverage*) και σπατάλης πόρων (*waste*). Αυτή η πολιτική αποδεικνύεται πιο αποτελεσματική σε περιβάλλοντα με προβλέψιμες συμπεριφορές συμβολαίων, όπως DEXs ή staking contracts, όπου τα ίδια πεδία ενημερώνονται επανειλημμένα.

Η εφαρμογή των πολιτικών γίνεται δυναμικά μέσα από το αρχείο ρυθμίσεων, όπου κάθε πολιτική δηλώνεται με το όνομά της και τα αντίστοιχα *params*. Η παρακάτω γραμμή από το αρχείο YAML:

Κώδικας 1. Εφαρμογή πολιτικής ReadAhead δυναμικά μέσα από το αρχείο των ρυθμίσεων

```
- name: ReadAhead
  params:
    blocks_ahead: 1
    max_items_per_block:
      accounts: 200
      storage: 400
```

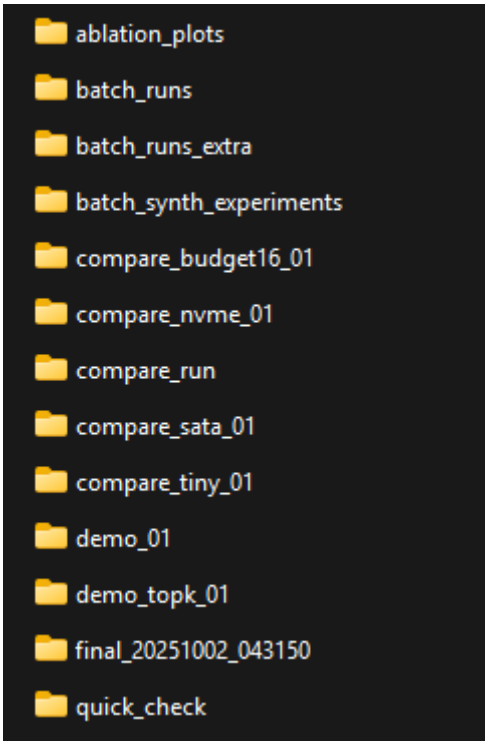
αντιστοιχεί στην ενεργοποίηση του ReadAhead prefetcher με προκαθορισμένα όρια ανά block. Αντίστοιχα, η δήλωση:

Κώδικας 2. Εφαρμογή πολιτικής ContractTopK δυναμικά μέσα από το αρχείο των ρυθμίσεων

```
- name: ContractTopK
  params:
    stats_path: data/topk_slots.json
    k: 5
```

ενεργοποιεί τη στατιστική πολιτική με βάση τα 5 πιο συχνά χρησιμοποιούμενα slots. Ο προσομοιωτής διαβάζει αυτές τις παραμέτρους, δημιουργεί τα κατάλληλα αντικείμενα πολιτικών και τις εφαρμόζει σε κάθε κύκλο εκτέλεσης του trace.

Οι μετρικές που προκύπτουν από την εκτέλεση των πολιτικών καταγράφονται αυτόματα στο αρχείο *metrics.csv* και αποτυπώνονται γραφικά μέσω του module *plots*.



Εικόνα 9. Δομή του φακέλου results/ που περιέχει τα παραγόμενα πειραματικά δεδομένα από τις εκτελέσεις του προσομοιωτή. Κάθε υποφάκελος αντιστοιχεί σε διαφορετικό σενάριο.

3.5 Ορισμός και υπολογισμός μετρικών αξιολόγησης (hit rate, coverage, waste, speedup)

Στην παρούσα εργασία χρησιμοποιούνται τέσσερις βασικές μετρικές: **hit rate**, **coverage**, **waste** και **speedup**.

Η μετρική **hit rate** εκφράζει το ποσοστό των αιτημάτων ανάγνωσης που ικανοποιούνται απευθείας από την cache χωρίς να απαιτείται πρόσβαση στο υποκείμενο αποθηκευτικό υπόστρωμα. Μαθηματικά, ορίζεται ως ο λόγος του πλήθους των επιτυχημένων αναζητήσεων (*hits*) προς το συνολικό πλήθος προσπελάσεων (*hits* + *misses*). Ένας υψηλός hit rate υποδηλώνει ότι το προανακτημένο ή ήδη αποθηκευμένο περιεχόμενο είναι επαρκώς αντιπροσωπευτικό των επικείμενων αιτημάτων, μειώνοντας την ανάγκη για δαπανηρές λειτουργίες I/O. Η συγκεκριμένη μετρική έχει ιδιαίτερη σημασία στα blockchain περιβάλλοντα, όπου οι καθυστερήσεις ανάγνωσης από τη βάση δεδομένων (π.χ. LevelDB, RocksDB) επηρεάζουν άμεσα τον χρόνο εκτέλεσης των συναλλαγών.

Η **coverage** αναφέρεται στο ποσοστό των επιτυχημένων προφορτώσεων που χρησιμοποιήθηκαν πράγματι κατά την εκτέλεση. Εκφράζει δηλαδή τη χρησιμότητα της πολιτικής προφόρτωσης, δείχνοντας ποιο ποσοστό των δεδομένων που προφορτώθηκαν αξιοποιήθηκε τελικά. Υπολογίζεται ως ο λόγος των επιτυχημένων προανακτήσεων που χρησιμοποιήθηκαν (*useful prefetches*) προς το συνολικό πλήθος των προφορτώσεων.

Η **waste** αποτυπώνει το ποσοστό των προφορτωμένων δεδομένων που δεν χρησιμοποιήθηκαν ποτέ. Εκφράζεται ως ο λόγος των άχρηστων προφορτώσεων (*useless prefetches*) προς το σύνολο των προφορτώσεων. Αν και ένα μικρό ποσοστό waste είναι αναμενόμενο λόγω της προληπτικής φύσης της προφόρτωσης, υψηλές τιμές δηλώνουν μη αποδοτική αξιοποίηση των πόρων, ιδιαίτερα κρίσιμη παράμετρος για τα αποκεντρωμένα συστήματα, όπου η υπερφόρτωση μνήμης ή δικτύου μπορεί να επιφέρει ενεργειακό και υπολογιστικό κόστος.

Τέλος, η μετρική **speedup** αποτελεί τη συνολική απόδοση της πολιτικής σε σχέση με το baseline (δηλαδή την εκτέλεση χωρίς prefetching). Υπολογίζεται ως ο λόγος του συνολικού χρόνου εκτέλεσης της baseline περίπτωσης προς τον χρόνο εκτέλεσης με προφόρτωση. Τιμές μεγαλύτερες του 1 δηλώνουν επιτάχυνση, ενώ τιμές μικρότερες του 1 υποδηλώνουν επιβράδυνση.

Στην παρούσα εργασία, όλες οι μετρικές υπολογίζονται αυτόματα κατά τη διάρκεια της προσομοίωσης, μέσω του module **metrics.py**, το οποίο συλλέγει δεδομένα από τα υποσυστήματα της cache και του προσομοιωτή, δημιουργώντας συγκεντρωτικούς πίνακες αποτελεσμάτων σε μορφή CSV. Τα αρχεία αυτά αποθηκεύονται στον φάκελο *results/*, παρέχοντας δυνατότητα μεταγενέστερης ανάλυσης ή οπτικοποίησης.

3.6 Πειραματικό πρωτόκολλο

Αρχικά, καθορίστηκε ένα ενιαίο **workload** το οποίο εφαρμόστηκε σε όλες τις πολιτικές. Το συγκεκριμένο workload προέρχεται είτε από πραγματικά traces του Ethereum mainnet (εξαγόμενα μέσω Google BigQuery), είτε από συνθετικά datasets που αναπαράγουν χαρακτηριστικά προτύπων πρόσβασης σε smart contracts. Η χρήση κοινού συνόλου δεδομένων διασφαλίζει ότι κάθε πολιτική δέχεται τις ίδιες ακολουθίες εντολών, γεγονός που επιτρέπει τη σύγκριση με όρους ίσης πολυπλοκότητας και χρονικού κόστους. Επίσης, για κάθε πείραμα χρησιμοποιήθηκαν τα ίδια latency profiles.

Κατά τη φάση εκτέλεσης, όλες οι πολιτικές τρέχουν σειριακά μέσω του *orchestrator script* (*run_experiment.py* ή *batch_run.py*), χρησιμοποιώντας τις ίδιες παραμέτρους εισόδου και κοινό αρχείο *trace*. Οι εκτελέσεις καταγράφονται σε μορφή *CSV* με τις ίδιες στήλες μετρικών (*hit_rate, coverage, waste, speedup*), γεγονός που επιτρέπει τη στατιστική σύγκριση των αποτελεσμάτων με ακρίβεια.

Κεφάλαιο 4 – Υλοποίηση

4.1 Δομή κώδικα (project layout, φάκελοι configs/data/src/results)

Το έργο αναπτύχθηκε σε περιβάλλον Python (>=3.10) με χρήση εικονικού περιβάλλοντος (*venv*), γεγονός που επιτρέπει την απομόνωση των βιβλιοθηκών και την αποφυγή συγκρούσεων εξαρτήσεων. Η βασική δομή του project περιλαμβάνει μια διακριτή ταξινόμηση των φακέλων σε τέσσερις κύριες κατηγορίες: *configs*, *data*, *src* και *results*.

Στον φάκελο configs βρίσκονται τα αρχεία παραμετροποίησης σε μορφή YAML. Κάθε αρχείο περιγράφει πλήρως τις συνθήκες εκτέλεσης ενός πειράματος, όπως τη χωρητικότητα των cache layers, τα latency profiles και τις παραμέτρους κάθε πολιτικής προφόρτωσης. Η ύπαρξη αυτών των αρχείων καθιστά δυνατή τη γρήγορη αναπαραγωγή οποιασδήποτε εκτέλεσης.

Ο φάκελος data φιλοξενεί τα αρχεία εισόδου (*traces*) που χρησιμοποιούνται από το module *loader* κατά την έναρξη της προσομοίωσης. Περιλαμβάνει δεδομένα πραγματικής προέλευσης από το Ethereum mainnet, τα οποία έχουν εξαχθεί μέσω Google BigQuery και μετατραπεί σε JSONL μορφή για ευκολότερη επεξεργασία, καθώς και συνθετικά datasets που δημιουργήθηκαν για ελεγχόμενα σενάρια αξιολόγησης.

Ο φάκελος src περιλαμβάνει τον πυρήνα της υλοποίησης. Περιέχει τα βασικά modules που συναποτελούν τη λειτουργικότητα του προσομοιωτή, όπως *cache.py* (μηχανισμός cache και

counters), *simulator.py* (επεξεργασία block-by-block και εκτέλεση συναλλαγών), *policies/* (υλοποίηση διαφορετικών prefetching στρατηγικών), *metrics.py* (υπολογισμός και καταγραφή των μετρικών αξιολόγησης) και *plots.py* (οπτικοποίηση αποτελεσμάτων). Η αρχιτεκτονική ακολουθεί την αρχή της modular decomposition, όπου κάθε αρχείο επιτελεί μία σαφώς καθορισμένη λειτουργία, διευκολύνοντας τη μελλοντική αντικατάσταση ή προσθήκη νέων πολιτικών χωρίς την ανάγκη ανασχεδιασμού του πυρήνα.

Ο φάκελος results αποτελεί το τελικό σημείο συγκέντρωσης όλων των εκτελέσεων. Περιέχει τα εξαγόμενα αρχεία *CSV* με τις μετρικές (*metrics.csv*), τα παραγόμενα γραφήματα, καθώς και τους υποφακέλους για κάθε πείραμα.

4.2 Κύρια scripts

4.2.1. simulator.py – προσομοίωση block-by-block

Κώδικας 3. Κώδικας simulator.py

```
from typing import Dict, Optional

from .schema import CacheConfig
from .cache import StateCache
from .metrics import Metrics
from .policies.baseline import BaselinePolicy
from .policies.readahead import ReadAheadPolicy
from .policies.contract_topk import ContractTopKPolicy


def run_simulation(
    policy_name: str,
    blocks: Dict[int, list],
    cache_cfg: CacheConfig,
    policy_params: Optional[dict] = None,
) -> Metrics:
    """
    Εκτελεί την προσομοίωση για ένα σύνολο blocks με βάση την επιλεγμένη πολιτική.
    - Προανάκτηση (accounts + storage) ανά block σύμφωνα με το plan της πολιτικής
    - Πρόσβαση accounts/SLOAD/SSTORE με χρέωση latency ανά hit/miss
    - Συγκεντρώνει counters και επιστρέφει Metrics
    """
    policy = _make_policy(policy_name, policy_params or {})
    cache = StateCache(cache_cfg)

    # Τρέχουμε με φυσική σειρά block id
    for block_id in sorted(blocks.keys()):
        accesses = blocks[block_id]

        # Prefetch για το block
        plan = policy.plan_for_block(block_id, blocks)
        cache.prefetch_accounts(plan.get("accounts", []))
```

```
            cache.prefetch_storage(plan.get("storage", []))

            # Εκτέλεση προσπελάσεων
            for a in accesses:
                if getattr(a, "kind", None) in ("SLOAD", "SSTORE"):
                    cache.access_storage(a.address, a.slot)
                else:
                    cache.access_account(a.address)

        # Τερματισμός/λογισμός waste
        cache.finalize()
        c = cache.counters

        return Metrics(
            policy=getattr(policy, "name", policy_name),
            blocks=len(blocks),
            hits=c.hits,
            misses=c.misses,
            prefetch_items=c.prefetch_items,
            prefetch_used=c.prefetch_used,
            prefetch_wasted=c.prefetch_wasted,
            t_hits_us=c.t_hits_us,
            t_misses_us=c.t_misses_us,
            t_prefetch_us=c.t_prefetch_us,
        )


    def _make_policy(name: str, params: dict):
        """
        Factory για πολιτικές. Υποστηρίζει:
        - Baseline
        - ReadAhead (με optional prefetch_budget_items)
        - ContractTopK (με optional prefetch_budget_items)
        """
        name_low = (name or "").lower()

        if name_low == "baseline":
            return BaselinePolicy()

        if name_low == "readahead":
            return ReadAheadPolicy(
                blocks_ahead=int(params.get("blocks_ahead", 1)),
                max_items_per_block=params.get("max_items_per_block", 1000),
                prefetch_budget_items=params.get("prefetch_budget_items"),
            )

        if name_low in ("contracttopk", "contract_topk", "topk"):
            return ContractTopKPolicy(
                stats_path=params.get("stats_path", "data/topk_slots.json"),
                k=int(params.get("k", 8)),
                blocks_ahead=int(params.get("blocks_ahead", 1)),
                include_accounts=bool(params.get("include_accounts", True)),
                prefetch_budget_items=params.get("prefetch_budget_items"),
            )

        raise ValueError("Unknown policy: %s" % name)
```

Ο κώδικας αυτός αποτελεί τον πυρήνα του προσομοιωτή εκτέλεσης για την αξιολόγηση των πολιτικών προφόρτωσης στο επίπεδο του **Ethereum execution layer**. Ενσωματώνει τόσο τη λογική εκτέλεσης των προσπελάσεων στο state, όσο και την καταγραφή των μετρικών που αξιολογούν την απόδοση κάθε πολιτικής. Η λειτουργία του οργανώνεται γύρω από τη συνάρτηση **run_simulation**, η οποία αποτελεί το κεντρικό σημείο εκκίνησης κάθε πειράματος, και τη βοηθητική **_make_policy**, η οποία δρα ως εργοστάσιο δημιουργίας (factory) για τις διαθέσιμες πολιτικές προφόρτωσης.

Η συνάρτηση **run_simulation** δέχεται τέσσερις κύριες παραμέτρους: το όνομα της πολιτικής (policy_name), το σύνολο των blocks προς προσομοίωση (blocks), τη διαμόρφωση της cache (cache_cfg) και, προαιρετικά, τις επιμέρους παραμέτρους της πολιτικής (policy_params). Κατά την εκκίνηση, η συνάρτηση δημιουργεί αντικείμενο πολιτικής μέσω της _make_policy, το οποίο μπορεί να είναι μία από τις **BaselinePolicy**, **ReadAheadPolicy** ή **ContractTopKPolicy**, ανάλογα με το όρισμα. Στη συνέχεια, δημιουργείται ένα στιγμιότυπο του **StateCache**, το οποίο περιλαμβάνει τους μετρητές πρόσβασης και τα καθορισμένα προφίλ καθυστέρησης (latencies).

Η διαδικασία εκτέλεσης ακολουθεί φυσική σειρά αύξοντος **block id**, γεγονός που εξασφαλίζει ρεαλιστική αναπαράσταση της εκτέλεσης blockchain συναλλαγών. Για κάθε block, η πολιτική υπολογίζει ένα **prefetch plan**, δηλαδή το σύνολο των δεδομένων που πρέπει να προφορτωθούν εκ των προτέρων (accounts και storage slots). Το σχέδιο αυτό καθορίζεται από τη μέθοδο **plan_for_block** της κάθε πολιτικής και μπορεί να διαφέρει δραστικά: στη **ReadAheadPolicy** το σχέδιο βασίζεται στα επόμενα blocks, ενώ στην **ContractTopKPolicy** στατιστικά δεδομένα καθορίζουν ποια στοιχεία θα προφορτωθούν. Το προφορτωμένο περιεχόμενο εγγράφεται προσωρινά στην cache μέσω των μεθόδων **prefetch_accounts** και **prefetch_storage**.

Κατά τη διάρκεια της εκτέλεσης των προσπελάσεων, κάθε αντικείμενο που περιέχεται στο trace προσπελαύνεται μέσω των αντίστοιχων συναρτήσεων της cache. Αν η αναζήτηση αφορά αποθηκευμένα δεδομένα λογαριασμού, χρησιμοποιείται η **access_account**, ενώ για προσβάσεις σε storage slots καλείται η **access_storage**. Ο μηχανισμός αυτός καταγράφει για κάθε πρόσβαση αν πρόκειται για *hit* (επιτυχία) ή *miss* (αποτυχία), καθώς και τους αντίστοιχους χρόνους καθυστέρησης, οι οποίοι εξαρτώνται από το latency profile της cache. Η λογική αυτή επιτρέπει την προσομοίωση πραγματικών συνθηκών αποθήκευσης, όπου η ανάγνωση από cache έχει πολύ μικρότερο κόστος από μια πρόσβαση στο υποκείμενο αποθηκευτικό μέσο.

Μετά την ολοκλήρωση όλων των blocks, η μέθοδος **finalize()** της cache εκτελεί έναν τελικό υπολογισμό για τα prefetches που δεν χρησιμοποιήθηκαν, προσδιορίζοντας το ποσοστό **waste**. Στο τέλος της διαδικασίας, η συνάρτηση δημιουργεί ένα αντικείμενο **Metrics**, το οποίο

περιέχει συγκεντρωτικά δεδομένα για τον αριθμό των *hits*, *misses*, *prefetch_used*, *prefetch_wasted* και τα συνολικά χρονικά κόστη. Οι τιμές αυτές αποτελούν τη βάση για τον υπολογισμό των μετρικών **hit rate**, **coverage**, **waste** και **speedup**, που αναλύονται στο επόμενο στάδιο αξιολόγησης.

Η βοηθητική συνάρτηση **_make_policy** λειτουργεί ως **factory pattern** για τη δημιουργία αντικειμένων πολιτικών με δυναμικό τρόπο. Με βάση το όνομα που δίνεται ως είσοδο, επιστρέφει το κατάλληλο αντικείμενο πολιτικής και περνάει τις σχετικές παραμέτρους παραμετροποίησης. Αν δοθεί όνομα που δεν αντιστοιχεί σε γνωστή πολιτική, η συνάρτηση εγείρει εξαίρεση (ValueError), εξασφαλίζοντας την ορθότητα των εισόδων.

4.2.2. cache.py – μηχανισμός LRU cache και counters

Κώδικας 4. Κώδικας cache.py

```
from collections import OrderedDict
from dataclasses import dataclass
from typing import Set, List, Tuple
from .schema import CacheConfig


@dataclass
class CacheCounters:
    hits: int = 0
    misses: int = 0
    prefetch_items: int = 0
    prefetch_used: int = 0
    prefetch_wasted: int = 0
    t_hits_us: int = 0
    t_misses_us: int = 0
    t_prefetch_us: int = 0


class LRUSet:
    """Απλό LRU για keys (strings)."""
    def __init__(self, capacity: int):
        self.capacity = max(1, capacity)
        self._od = OrderedDict()  # type: OrderedDict[str, bool]

    def __contains__(self, key: str) -> bool:
        return key in self._od

    def touch(self, key: str):
        if key in self._od:
            self._od.move_to_end(key)
        else:
            if len(self._od) >= self.capacity:
                self._od.popitem(last=False)
            self._od[key] = True

    def remove(self, key: str):
        self._od.pop(key, None)
```

```
def _stk(address: str, slot: str) -> str:
    """Φτιάχνει string-key για storage: addr|slot."""
    return f"{address}|{slot}"


class StateCache:
    """
    Δύο επίπεδα: accounts & storage. Συνδυαστικοί μετρητές (total).
    """
    def __init__(self, cfg: CacheConfig):
        self.cfg = cfg
        self.accounts = LRUSet(cfg.account_capacity)
        self.storage = LRUSet(cfg.storage_capacity)

        self._prefetched_pending_accounts: Set[str] = set()
        self._prefetched_pending_storage: Set[str] = set()
        self.counters = CacheCounters()

    # --------- Prefetch ----------
    def prefetch_accounts(self, addrs: List[str]) -> int:
        time_us = 0
        for addr in addrs:
            if addr not in self.accounts:
                self.accounts.touch(addr)
                self._prefetched_pending_accounts.add(addr)
                self.counters.prefetch_items += 1
                time_us += self.cfg.t_prefetch_item_us
        self.counters.t_prefetch_us += time_us
        return time_us

    def prefetch_storage(self, keys: List[Tuple[str, str]]) -> int:
        time_us = 0
        for addr, slot in keys:
            if slot is None:
                continue
            k = _stk(addr, slot)
            if k not in self.storage:
                self.storage.touch(k)
                self._prefetched_pending_storage.add(k)
                self.counters.prefetch_items += 1
                time_us += self.cfg.t_prefetch_item_us
        self.counters.t_prefetch_us += time_us
        return time_us

    # --------- Access ----------
    def access_account(self, addr: str) -> int:
        if addr in self.accounts:
            self.accounts.touch(addr)
            self.counters.hits += 1
            self.counters.t_hits_us += self.cfg.t_hit_account_us
            if addr in self._prefetched_pending_accounts:
                self._prefetched_pending_accounts.remove(addr)
                self.counters.prefetch_used += 1
            return self.cfg.t_hit_account_us
        else:
```

```
            self.accounts.touch(addr)
            self.counters.misses += 1
            self.counters.t_misses_us += self.cfg.t_miss_account_us
            return self.cfg.t_miss_account_us

    def access_storage(self, addr: str, slot: str) -> int:
        if slot is None:
            return self.access_account(addr)
        k = _stk(addr, slot)
        if k in self.storage:
            self.storage.touch(k)
            self.counters.hits += 1
            self.counters.t_hits_us += self.cfg.t_hit_storage_us
            if k in self._prefetched_pending_storage:
                self._prefetched_pending_storage.remove(k)
                self.counters.prefetch_used += 1
            return self.cfg.t_hit_storage_us
        else:
            self.storage.touch(k)
            self.counters.misses += 1
            self.counters.t_misses_us += self.cfg.t_miss_storage_us
            return self.cfg.t_miss_storage_us

    # --------- Finalize ----------
    def finalize(self):
        self.counters.prefetch_wasted +=
len(self._prefetched_pending_accounts)
        self.counters.prefetch_wasted +=
len(self._prefetched_pending_storage)
        self._prefetched_pending_accounts.clear()
        self._prefetched_pending_storage.clear()
```

Ο κώδικας υλοποιεί ένα μοντέλο cache για λογαριασμούς και storage slots, με LRU πολιτική αντικατάστασης και καταμέτρηση χρονομετρικών και ποσοτικών μετρικών. Στο ανώτερο επίπεδο βρίσκεται η κλάση StateCache, η οποία ενσαρκώνει δύο ανεξάρτητες LRU δομές (μία για accounts και μία για storage) και διατηρεί έναν συγκεντρωτικό μετρητή τύπου CacheCounters με πλήθος hits, misses, αντικειμένων που προφορτώθηκαν, πόσα από αυτά χρησιμοποιήθηκαν και πόσα σπαταλήθηκαν, καθώς και αθροιστικούς χρόνους σε μικροδευτερόλεπτα για hits, misses και prefetch. Οι παράμετροι καθυστέρησης και χωρητικότητας παρέχονται μέσω του CacheConfig.

Οι μετρητές ορίζονται στο CacheCounters ως dataclass, ώστε η συλλογή τους να είναι τυπική και αποδοτική. Η εσωτερική LRU υλοποιείται στην κλάση LRUSet χρησιμοποιώντας OrderedDict: το κλειδί είναι το αντικείμενο της cache (π.χ. διεύθυνση λογαριασμού ή «διεύθυνση|slot» για storage) και η τιμή είναι αδιάφορη· η σειρά εισαγωγής/αναφοράς κωδικοποιεί την πρόσφατη χρήση. Η μέθοδος touch μετακινεί το κλειδί στο τέλος όταν προσπελαστεί ξανά, ενώ στην εισαγωγή νέου κλειδιού ελέγχει τη χωρητικότητα και αποβάλλει

το παλαιότερο στοιχείο (LRU eviction) όταν απαιτείται. Η __contains__ δίνει O(1) έλεγχο ύπαρξης, ενώ η remove επιτρέπει ρητή διαγραφή, παρότι εδώ δεν χρησιμοποιείται από το ανώτερο επίπεδο.

Για τα storage κλειδιά ορίζεται η βοηθητική _stk(address, slot) που σχηματίζει ένα σταθερό string αναγνωριστικό με το πρότυπο addr|slot. Αυτή η κανονικοποίηση αποφεύγει περίπλοκες δομές σύνθετων κλειδιών και κάνει την LRU ανεξάρτητη από τύπους.

Η StateCache αρχικοποιείται με δύο LRUSet βάσει χωρητικοτήτων που προκύπτουν από το CacheConfig. Επιπλέον, διατηρεί δύο σύνολα (_prefetched_pending_accounts και _prefetched_pending_storage) για να παρακολουθεί τις προφορτωμένες αλλά «μη ακόμη χρησιμοποιημένες» εγγραφές. Αυτά τα σύνολα είναι ο μηχανισμός που επιτρέπει αργότερα να αποδοθεί ακριβώς ποια προφόρτωση ήταν ωφέλιμη και ποια σπαταλήθηκε. Με άλλα λόγια, όταν προφορτώνεται κάτι, σημειώνεται εδώ ως «pending» και όταν αργότερα το ίδιο αντικείμενο σε μια πρόσβαση καταλήξει σε hit, αφαιρείται από το αντίστοιχο pending set και αυξάνεται ο μετρητής prefetch_used. Ό,τι απομείνει pending μέχρι το τέλος, θα χαρακτηριστεί prefetch_wasted.

Οι μέθοδοι prefetch_accounts και prefetch_storage υλοποιούν την πράξη της προφόρτωσης. Για κάθε υποψήφιο στοιχείο ελέγχεται πρώτα αν είναι ήδη στην αντίστοιχη LRU, αν όχι εισάγεται με touch, καταγράφεται στο pending set, αυξάνεται ο μετρητής prefetch_items και προστίθεται στο t_prefetch_us ο σταθερός χρόνος προφόρτωσης ανά στοιχείο (cfg.t_prefetch_item_us). Η μέθοδος επιστρέφει τον συνολικό χρόνο που «χρεώθηκε» για την προφόρτωση, επιτρέποντας στον προσομοιωτή να συμπεριλάβει αυτό το κόστος στον συνολικό εκτελεστικό χρόνο.

Οι μέθοδοι προσπέλασης access_account και access_storage μοντελοποιούν την εκτέλεση SLOAD/SSTORE ή account-level reads και writes. Σε account access, αν η διεύθυνση υπάρχει στο LRU, θεωρείται hit: γίνεται touch για ανανέωση της LRU σειράς, αυξάνεται ο hits, προστίθεται ο αντίστοιχος χρόνος t_hit_account_us στον αθροιστή t_hits_us, και, αν ο λογαριασμός ήταν προφορτωμένος και ακόμη pending, αφαιρείται από το pending set και αυξάνεται το prefetch_used. Αν δεν υπάρχει, καταγράφεται miss: εισαγωγή με touch, αύξηση misses και προσθήκη του t_miss_account_us στον t_misses_us. Η access_storage λειτουργεί ανάλογα για storage κλειδιά. Όταν δεν υπάρχει slot, η μέθοδος ορθά εκτρέπει την πράξη σε access_account, διότι χωρίς slot η πρόσβαση ουσιαστικά είναι account-level. Στο φυσιολογικό storage access, σχηματίζεται το κλειδί με _stk; αν βρεθεί στο LRU, είναι hit με χρέωση t_hit_storage_us και, εφόσον ήταν pending, χαρακτηρίζεται used, αλλιώς είναι miss με χρέωση t_miss_storage_us. Οι χρόνοι hit/miss διαχωρίζονται μεταξύ account και storage γιατί έχουν

διαφορετικά προφίλ καθυστέρησης και αυτό επιτρέπει ρεαλιστική μοντελοποίηση. Στο τέλος ενός πειράματος καλείται η finalize, η οποία μετατρέπει ό,τι παραμένει στα pending sets σε prefetch_wasted. Μετά, τα pending sets καθαρίζονται ώστε η cache να βρίσκεται σε συνεπή τελική κατάσταση.

4.2.3. policies/ – Baseline, ReadAhead, ContractTopK

Κώδικας 5. Κώδικας Baseline Policy

```
from .base import PrefetchPolicy

class BaselinePolicy(PrefetchPolicy):
    name = "Baseline"

    def plan_for_block(self, current_block: int, blocks: dict) -> dict:
        return {"accounts": []}
```

Κώδικας 6.Κώδικας ReadAhead.py policy

```
from typing import Dict, List, Set, Tuple, Optional
from .base import PrefetchPolicy

def _limits(mp):
    # Δέχεται είτε int, είτε dict με accounts/storage
    if isinstance(mp, dict):
        return int(mp.get("accounts", 1000)), int(mp.get("storage", 4000))
    return int(mp), 0  # μόνο accounts


class ReadAheadPolicy(PrefetchPolicy):
    name = "ReadAhead"

    def __init__(
        self,
        blocks_ahead: int = 1,
        max_items_per_block=1000,
        prefetch_budget_items: Optional[int] = None,
    ):
        self.blocks_ahead = max(0, int(blocks_ahead))
        self.max_items_per_block = max_items_per_block
        self.prefetch_budget_items = (
            int(prefetch_budget_items) if prefetch_budget_items is not None
else None
        )

    def plan_for_block(self, current_block: int, blocks: Dict[int, list]) -
> dict:
```

```
        if self.blocks_ahead <= 0:
            return {"accounts": [], "storage": []}

        all_blocks = sorted(blocks.keys())
        try:
            idx = all_blocks.index(current_block)
        except ValueError:
            return {"accounts": [], "storage": []}

        max_accs, max_stor = _limits(self.max_items_per_block)
        budget = self.prefetch_budget_items

        accs: List[str] = []
        stor: List[Tuple[str, str]] = []
        seen_a: Set[str] = set()
        seen_s: Set[str] = set()

        def budget_full() -> bool:
            return budget is not None and (len(accs) + len(stor)) >= budget

        for step in range(1, self.blocks_ahead + 1):
            if idx + step >= len(all_blocks):
                break
            bnext = all_blocks[idx + step]
            for access in blocks[bnext]:
                if access.kind in ("SLOAD", "SSTORE") and max_stor > 0:
                    k = f"{access.address}|{access.slot}"
                    if k not in seen_s and len(stor) < max_stor:
                        seen_s.add(k)
                        stor.append((access.address, access.slot))
                else:
                    if access.address not in seen_a and len(accs) <
max_accs:
                        seen_a.add(access.address)
                        accs.append(access.address)

                if budget_full():
                    break

            if budget_full() or ((len(accs) >= max_accs) and (len(stor) >=
max_stor)):
                break

        return {"accounts": accs, "storage": stor}
```

Κώδικας 7. Κώδικας contract_topK.py policy

```
import json
from typing import Dict, List, Tuple, Set, Optional
from .base import PrefetchPolicy

class ContractTopKPolicy(PrefetchPolicy):
    name = "ContractTopK"

    def __init__(
        self,
        stats_path: str,
```

```
        k: int = 8,
        blocks_ahead: int = 1,
        include_accounts: bool = True,
        prefetch_budget_items: Optional[int] = None,
    ):
        self.blocks_ahead = max(0, int(blocks_ahead))
        self.k = int(k)
        self.include_accounts = bool(include_accounts)
        self.prefetch_budget_items = (
            int(prefetch_budget_items) if prefetch_budget_items is not None
else None
        )

        with open(stats_path, "r", encoding="utf-8") as f:
            # {addr: [slot1, slot2, ...]}
            self.topk: Dict[str, List[str]] = json.load(f)

    def plan_for_block(self, current_block: int, blocks: Dict[int, list]) -
> dict:
        accs: List[str] = []
        stor: List[Tuple[str, str]] = []

        all_blocks = sorted(blocks.keys())
        try:
            idx = all_blocks.index(current_block)
        except ValueError:
            return {"accounts": [], "storage": []}

        seen_c: Set[str] = set()
        for step in range(1, self.blocks_ahead + 1):
            if idx + step >= len(all_blocks):
                break
            bnext = all_blocks[idx + step]
            for a in blocks[bnext]:
                if a.kind in ("SLOAD", "SSTORE"):
                    seen_c.add(a.address)

        # Υποψήφιοι για prefetch
        cand_accs: List[str] = []
        cand_stor: List[Tuple[str, str]] = []

        for caddr in sorted(seen_c):
            if self.include_accounts:
                cand_accs.append(caddr)

        for caddr in sorted(seen_c):
            slots = self.topk.get(caddr, [])[: self.k]
            for s in slots:
                cand_stor.append((caddr, s))

        # Εφαρμογή budget: πρώτα accounts (αν ζητήθηκαν), μετά stor μέχρι
να φτάσουμε το όριο
        if self.prefetch_budget_items is not None:
            total = self.prefetch_budget_items
            take_a = min(len(cand_accs), total)
            accs = cand_accs[:take_a]
            remaining = total - take_a
```

```
            stor = cand_stor[: max(0, remaining)]
        else:
            accs = cand_accs
            stor = cand_stor

        return {"accounts": accs, "storage": stor}
```

Οι τρεις ενότητες περιγράφουν τις πολιτικές **Baseline**, **ReadAhead** και **ContractTopK** που εφαρμόζονται στον προσομοιωτή. Κάθε πολιτική υλοποιείται ως υποκλάση της αφηρημένης κλάσης PrefetchPolicy, η οποία ορίζει τη βασική διεπαφή που πρέπει να ακολουθεί κάθε πολιτική, κυρίως μέσω της μεθόδου plan_for_block(). Η κοινή αυτή διεπαφή επιτρέπει στον προσομοιωτή να εκτελεί πειράματα με διαφορετικές πολιτικές χωρίς να χρειάζεται να γνωρίζει τον εσωτερικό τους μηχανισμό.

Η **BaselinePolicy** λειτουργεί ως ουδέτερη γραμμή αναφοράς (control group). Η κλάση δεν πραγματοποιεί καμία προφόρτωση και επιστρέφει πάντα ένα κενό σχέδιο ({"accounts": []}). Αυτό σημαίνει ότι κάθε πρόσβαση θα εκτελεστεί κανονικά μέσω της cache, χωρίς καμία προληπτική ανάκτηση δεδομένων. Ουσιαστικά, ο ρόλος της είναι να μετρήσει τη φυσική συμπεριφορά του συστήματος χωρίς βοήθεια από προγνωστικούς μηχανισμούς, παρέχοντας τη «γραμμή βάσης» από την οποία θα υπολογιστεί η σχετική επιτάχυνση.

Η **ReadAheadPolicy** αποτελεί την πρώτη «ενεργή» πολιτική προφόρτωσης και βασίζεται στην αρχή της χρονικής τοπικότητας. Η πολιτική δέχεται τρεις βασικές παραμέτρους: blocks_ahead, που καθορίζει πόσα επόμενα blocks θα εξεταστούν για προφόρτωση, max_items_per_block, που ορίζει το μέγιστο πλήθος αντικειμένων (accounts και storage slots) που μπορούν να προφορτωθούν ανά block και prefetch_budget_items, που προαιρετικά περιορίζει το συνολικό αριθμό στοιχείων που μπορούν να προφορτωθούν. Εσωτερικά, η πολιτική αναζητά τη θέση του τρέχοντος block στη λίστα των διαθέσιμων blocks και επαναλαμβάνει τη διαδικασία προφόρτωσης για όσα επόμενα βρίσκονται εντός του παραθύρου που ορίζεται από το blocks_ahead. Για κάθε επόμενο block, συλλέγονται τα accounts και τα storage slots που θα προσπελαστούν. Εφόσον αυτά δεν έχουν ήδη προστεθεί στο σύνολο seen_a ή seen_s, καταγράφονται στα αντίστοιχα σύνολα accs και stor. Ο μηχανισμός αυτός εξασφαλίζει ότι δεν θα υπάρξουν διπλές προφορτώσεις για τα ίδια αντικείμενα.

Παράλληλα, εφαρμόζεται έλεγχος ορίου (μέσω της εσωτερικής συνάρτησης budget_full()), ώστε η πολιτική να σταματά όταν το πλήθος των προφορτωμένων αντικειμένων υπερβεί το καθορισμένο budget, αν αυτό υπάρχει. Επιστρέφεται τελικά ένα λεξικό με δύο λίστες: μία για accounts και μία για storage slots, οι οποίες αντιπροσωπεύουν το σχέδιο προφόρτωσης του επόμενου block.

Η **ContractTopKPolicy** ακολουθεί μια διαφορετική, στατιστική προσέγγιση προφόρτωσης. Η πολιτική στηρίζεται σε ένα εξωτερικό αρχείο στατιστικών (stats_path), το οποίο περιέχει για κάθε διεύθυνση συμβολαίου, μια λίστα με τα πιο συχνά προσπελαζόμενα slots. Αυτά τα δεδομένα έχουν παραχθεί σε προγενέστερο στάδιο με τη βοήθεια του εργαλείου analyze.py, το οποίο επεξεργάζεται traces “offline” και υπολογίζει τα *top-K* slots ανά συμβόλαιο.

Κατά την αρχικοποίηση της πολιτικής, το αρχείο αυτό διαβάζεται σε μορφή JSON και αποθηκεύεται ως λεξικό self.topk, όπου κάθε κλειδί είναι διεύθυνση συμβολαίου και η τιμή του είναι η λίστα των συχνότερων slots. Στη φάση εκτέλεσης, η πολιτική εξετάζει τα επόμενα blocks (blocks_ahead) και καταγράφει ποιες διευθύνσεις συμβολαίων θα εμφανιστούν σε αυτές τις συναλλαγές. Για καθεμία από αυτές, αν το flag include_accounts είναι ενεργό, προσθέτει και τη διεύθυνση του λογαριασμού στη λίστα υποψηφίων accounts (cand_accs). Στη συνέχεια, για κάθε διεύθυνση, ανακτά από το λεξικό self.topk τα πρώτα *K* slots και τα προσθέτει στη λίστα υποψηφίων storage slots (cand_stor).

Η πολιτική μπορεί επίσης να περιοριστεί μέσω του παραμέτρου prefetch_budget_items, που λειτουργεί ως ανώτατο όριο για το πλήθος των προφορτωμένων αντικειμένων. Αν υπάρχει όριο, προφορτώνονται πρώτα οι λογαριασμοί (accounts) και κατόπιν τα storage slots μέχρι να συμπληρωθεί το διαθέσιμο budget. Αν δεν υπάρχει περιορισμός, προφορτώνονται όλα τα υποψήφια δεδομένα που προέκυψαν από την ανάλυση. Τελικά, η πολιτική επιστρέφει ένα λεξικό με τις λίστες accounts και storage, το οποίο καθορίζει τα αντικείμενα που θα περάσουν στο στάδιο προφόρτωσης.

Σε αντιδιαστολή με τη ReadAhead, η ContractTopKPolicy δεν στηρίζεται στη χρονική διαδοχή των blocks αλλά σε **στατιστικά προφίλ επαναληψιμότητας** ανά συμβόλαιο. Επομένως, αποδίδει καλύτερα σε περιβάλλοντα όπου οι ίδιες συναρτήσεις smart contracts εκτελούνται επανειλημμένα, όπως συμβαίνει σε DeFi πρωτόκολλα ή token transfers. Η πολιτική αυτή, παρότι πιο σταθερή σε ακανόνιστα workloads, εξαρτάται από την ακρίβεια των στατιστικών και απαιτεί πρόσθετο στάδιο offline ανάλυσης.

4.2.4.analyze.py – offline στατιστική ανάλυση slots per contract

Κώδικας 8. Κώδικας analyze.py

```
import json
import argparse
from collections import Counter, defaultdict
from typing import Dict, List, Tuple
from .io_loader import load_jsonl

def build_topk_stats(workload_path: str, k: int = 8) -> Dict[str,
List[Tuple[str, int]]]:
```

```
    """
    Μετρά συχνότητες slots ανά contract από SLOAD/SSTORE και επιστρέφει
    {contract_addr: [(slot, count), ...] με φθίνουσα σειρά και μέχρι k
στοιχεία}.
    """
    accesses = load_jsonl(workload_path)
    counts: Dict[str, Counter] = defaultdict(Counter)
    for a in accesses:
        if a.kind in ("SLOAD", "SSTORE") and a.slot is not None:
            counts[a.address][a.slot] += 1

    out: Dict[str, List[Tuple[str, int]]] = {}
    for addr, counter in counts.items():
        out[addr] = counter.most_common(k)
    return out

def save_topk_json(path_out: str, stats: Dict[str, List[Tuple[str, int]]])
-> None:
    # σώζουμε ως {addr: [slot1, slot2, ...]}
    slim = {addr: [slot for slot, _c in lst] for addr, lst in
stats.items()}
    with open(path_out, "w", encoding="utf-8") as f:
        json.dump(slim, f, ensure_ascii=False, indent=2)

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("workload", help="path σε JSONL workload")
    ap.add_argument("--k", type=int, default=8)
    ap.add_argument("--out", type=str, default="data/topk_slots.json")
    args = ap.parse_args()

    stats = build_topk_stats(args.workload, args.k)
    save_topk_json(args.out, stats)
    print(f"Saved Top-K stats → {args.out} (k={args.k})")

if __name__ == "__main__":
    main()
```

Ο συγκεκριμένος κώδικας αποτελεί ένα βοηθητικό εργαλείο ανάλυσης, το οποίο χρησιμοποιείται για την παραγωγή στατιστικών συχνοτήτων πρόσβασης στα **storage slots** των έξυπνων συμβολαίων. Το αρχείο αυτό λειτουργεί ως **offline preprocessing module** του συστήματος και τροφοδοτεί με δεδομένα την πολιτική προφόρτωσης ContractTopKPolicy. Στόχος του είναι να αναλύσει workloads (traces) από συναλλαγές του Ethereum, να εντοπίσει για κάθε contract ποια slots προσπελαύνονται συχνότερα και να δημιουργήσει ένα συνοπτικό JSON αρχείο με τις πιο σημαντικές καταχωρήσεις (top-K), που θα χρησιμοποιηθούν στη φάση της προσομοίωσης.

Η συνάρτηση **build_topk_stats** δέχεται ως είσοδο τη διαδρομή του αρχείου trace (workload_path) και την παράμετρο **k**, η οποία καθορίζει πόσες κορυφαίες προσπελάσεις ανά συμβόλαιο θα κρατηθούν. Το αρχείο εισόδου είναι σε μορφή **JSONL (JSON Lines)**, δηλαδή

κάθε γραμμή περιέχει ένα ανεξάρτητο JSON αντικείμενο που αντιπροσωπεύει μια πρόσβαση (π.χ. SLOAD, SSTORE ή account access). Η ανάγνωση του αρχείου πραγματοποιείται μέσω της βοηθητικής συνάρτησης **load_jsonl**, η οποία επιστρέφει λίστα από αντικείμενα πρόσβασης.

Για κάθε εγγραφή, ο κώδικας ελέγχει αν η ενέργεια ανήκει στις κατηγορίες *SLOAD* ή *SSTORE* (δηλαδή αν αφορά ανάγνωση ή εγγραφή σε storage slot) και αν το πεδίο slot δεν είναι κενό. Αν οι συνθήκες ικανοποιούνται, αυξάνεται ο μετρητής της αντίστοιχης διεύθυνσης (a.address) για το συγκεκριμένο slot. Η μεταβλητή **counts** είναι ένα λεξικό τύπου defaultdict(Counter). Κάθε contract διεύθυνση αντιστοιχίζεται σε έναν μετρητή (Counter) που παρακολουθεί τη συχνότητα εμφάνισης των slots. Μετά τη συλλογή όλων των προσπελάσεων, για κάθε συμβόλαιο δημιουργείται μια λίστα από τα **k συχνότερα slots**, ταξινομημένα κατά φθίνουσα συχνότητα, μέσω της μεθόδου most_common(k) του Counter. Το αποτέλεσμα αποθηκεύεται στο λεξικό **out**, όπου κάθε κλειδί είναι μια διεύθυνση συμβολαίου και κάθε τιμή είναι μια λίστα από ζεύγη (slot, count).

Η επόμενη συνάρτηση, **save_topk_json**, αναλαμβάνει να μετατρέψει τη δομή των αποτελεσμάτων σε μορφή κατάλληλη για χρήση από τον προσομοιωτή. Αντί να κρατήσει και τις τιμές των μετρήσεων, δημιουργεί ένα "slim" λεξικό που περιέχει μόνο τις διευθύνσεις και τις λίστες των slots, δηλαδή {addr: [slot1, slot2, ...]}. Αυτό το αρχείο είναι πολύ ελαφρύτερο και ευκολότερο στην ανάγνωση από τις πολιτικές, αφού δεν απαιτούνται οι απόλυτοι αριθμοί προσπελάσεων, παρά μόνο η κατάταξη των slots. Η εγγραφή στο δίσκο πραγματοποιείται με χρήση της json.dump, με παραμέτρους που διασφαλίζουν ευανάγνωστη μορφοποίηση (indentation και UTF-8 encoding). Το παραγόμενο αρχείο αποθηκεύεται, στο φάκελο data/ .

Η συνάρτηση **main()** καθιστά το script εκτελέσιμο από τη γραμμή εντολών, προσδίδοντάς του λειτουργικότητα **stand-alone εργαλείου**. Μέσω της βιβλιοθήκης argparse, ορίζονται τα απαραίτητα ορίσματα: η διαδρομή του workload (workload), η τιμή του k (με προεπιλογή 8) και η διαδρομή εξόδου (--out), που προεπιλεγμένα είναι data/topk_slots.json. Αφού αναλυθούν τα ορίσματα, το script καλεί τη build_topk_stats για την παραγωγή των στατιστικών και τη save_topk_json για την αποθήκευσή τους. Στο τέλος, εκτυπώνεται μήνυμα επιτυχούς ολοκλήρωσης, ενημερώνοντας τον χρήστη για το πού αποθηκεύτηκαν τα αποτελέσματα και ποια τιμή του *k* χρησιμοποιήθηκε.

4.2.5.run_experiment.py / batch_run.py – orchestrator και automation

*Κώδικας 9. Κώδικας **run_experiment.py***

```
# -*- coding: utf-8 -*-
```

```
"""
Single-run driver για τον simulator.

- Διαβάζει YAML
- Δέχεται CLI overrides (και σε dotted και σε underscored μορφή)
- Bridge ονομάτων: distance->blocks_ahead, slots_path/topk_file-
>stats_path, size_kb->capacities
- Τρέχει baseline / readahead / contract_topk
- Γράφει metrics.csv, και φτιάχνει plots (baseline vs κάθε policy) + multi
plot
- Σώζει effective_config/keys για έλεγχο
"""

import sys
import csv
import json
import yaml
import argparse
import datetime
from pathlib import Path
from typing import List, Dict, Any

from .schema import CacheConfig
from .io_loader import load_jsonl, group_by_block
from .simulator import run_simulation
from .plots import save_basic_plots, save_multi_plots


# ---------------- Helpers ----------------

def _as_int(x, default=None):
    try:
        return int(x)
    except Exception:
        return default

def _ensure_dir(p: Path):
    p.mkdir(parents=True, exist_ok=True)
    return p

def _now_tag() -> str:
    return datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

def _metrics_row(m, tag: str, speedup: float) -> Dict[str, Any]:
    return {
        "policy": tag,
        "blocks": m.blocks,
        "hits": m.hits,
        "misses": m.misses,
        "hit_rate": f"{m.hit_rate:.6f}",
        "prefetch_items": m.prefetch_items,
        "prefetch_used": m.prefetch_used,
        "prefetch_wasted": m.prefetch_wasted,
        "coverage": f"{m.coverage:.6f}",
        "waste": f"{m.waste:.6f}",
        "t_hits_us": m.t_hits_us,
        "t_misses_us": m.t_misses_us,
```

```
        "t_prefetch_us": m.t_prefetch_us,
        "total_time_us": m.total_time_us,
        "speedup_vs_baseline": f"{speedup:.6f}",
    }


# --------------- Load & overrides ---------------

def load_yaml_config(path: Path) -> Dict[str, Any]:
    with open(path, "r", encoding="utf-8") as f:
        return yaml.safe_load(f) or {}


def apply_cli_overrides(cfg: Dict[str, Any], argv: List[str]) -> Dict[str,
Any]:
    """
    Υποστηρίζει dotted και underscored args, χαρτογραφημένα στο ίδιο dest.
    """
    p = argparse.ArgumentParser(add_help=False)

    # workload
    p.add_argument("--config.workload_path",
dest="config_workload_path")
    p.add_argument("--config_workload_path",
dest="config_workload_path")

    # prefetch
    p.add_argument("--config.prefetch.distance",
dest="config_prefetch_distance")
    p.add_argument("--config_prefetch_distance",
dest="config_prefetch_distance")
    p.add_argument("--config.prefetch.blocks_ahead",
dest="config_prefetch_blocks_ahead")
    p.add_argument("--config_prefetch_blocks_ahead",
dest="config_prefetch_blocks_ahead")

    # topk
    p.add_argument("--config.topk.k",               dest="config_topk_k")
    p.add_argument("--config_topk_k",               dest="config_topk_k")
    p.add_argument("--config.topk.stats_path",
dest="config_topk_stats_path")
    p.add_argument("--config_topk_stats_path",
dest="config_topk_stats_path")
    p.add_argument("--config.topk.slots_path",
dest="config_topk_slots_path")
    p.add_argument("--config_topk_slots_path",
dest="config_topk_slots_path")

    # cache
    p.add_argument("--config.cache.size_kb",
dest="config_cache_size_kb")
    p.add_argument("--config_cache_size_kb",
dest="config_cache_size_kb")
    p.add_argument("--config.cache.account_capacity",
dest="config_cache_account_capacity")
    p.add_argument("--config_cache_account_capacity",
dest="config_cache_account_capacity")
```

```
    p.add_argument("--config.cache.storage_capacity",
dest="config_cache_storage_capacity")
    p.add_argument("--config_cache_storage_capacity",
dest="config_cache_storage_capacity")

    # results dir
    p.add_argument("--results.dir",                 dest="results_dir")
    p.add_argument("--results_dir",                 dest="results_dir")

    known, _ = p.parse_known_args(argv)

    # Εφαρμογή
    if getattr(known, "config_workload_path", None):
        cfg.setdefault("workload", {})["path"] = known.config_workload_path

    if getattr(known, "config_prefetch_blocks_ahead", None):
        cfg.setdefault("prefetch", {})["blocks_ahead"] =
_as_int(known.config_prefetch_blocks_ahead)
    if getattr(known, "config_prefetch_distance", None):
        cfg.setdefault("prefetch", {})["blocks_ahead"] =
_as_int(known.config_prefetch_distance)

    if getattr(known, "config_topk_k", None):
        cfg.setdefault("topk", {})["k"] = _as_int(known.config_topk_k)
    if getattr(known, "config_topk_stats_path", None):
        cfg.setdefault("topk", {})["stats_path"] =
known.config_topk_stats_path
    if getattr(known, "config_topk_slots_path", None):
        cfg.setdefault("topk", {})["stats_path"] =
known.config_topk_slots_path  # bridge

    cache = cfg.setdefault("cache", {})
    if getattr(known, "config_cache_size_kb", None):
        size_kb = _as_int(known.config_cache_size_kb)
        if size_kb:
            half = max(1, (size_kb * 1024) // 2)
            cache.setdefault("account_capacity", half)
            cache.setdefault("storage_capacity", half)
    if getattr(known, "config_cache_account_capacity", None):
        cache["account_capacity"] =
_as_int(known.config_cache_account_capacity)
    if getattr(known, "config_cache_storage_capacity", None):
        cache["storage_capacity"] =
_as_int(known.config_cache_storage_capacity)

    if getattr(known, "results_dir", None):
        cfg.setdefault("results", {})["dir"] = known.results_dir

    return cfg


# --------------------- Main ---------------------

def main():
    ap = argparse.ArgumentParser(description="Run single experiment")
    ap.add_argument("config", type=str, help="Path to final_experiment.yaml
(ή άλλο YAML)")
```

```
    args, rest = ap.parse_known_args()

    cfg_path = Path(args.config)
    cfg = load_yaml_config(cfg_path)
    cfg = apply_cli_overrides(cfg, rest)

    # Bridge από YAML αν υπάρχουν παλιά ονόματα
    if "prefetch" in cfg:
        pf = cfg["prefetch"]
        if "blocks_ahead" not in pf and "distance" in pf:
            pf["blocks_ahead"] = _as_int(pf["distance"])
    if "topk" in cfg:
        tk = cfg["topk"]
        if "stats_path" not in tk:
            for alt in ("slots_path", "topk_file", "file"):
                if alt in tk:
                    tk["stats_path"] = tk[alt]
                    break
    if "cache" in cfg:
        cc = cfg["cache"]
        if "account_capacity" not in cc or "storage_capacity" not in cc:
            size_kb = _as_int(cc.get("size_kb"))
            if size_kb:
                half = max(1, (size_kb * 1024) // 2)
                cc.setdefault("account_capacity", half)
                cc.setdefault("storage_capacity", half)

    # Out dir
    results_dir = Path(cfg.get("results", {}).get("dir",
"results/compare_run")).resolve()
    _ensure_dir(results_dir)

    # Debug dumps
    ts_tag = _now_tag()
    with open(results_dir / f"effective_config_{ts_tag}.json", "w",
encoding="utf-8") as f:
        json.dump(cfg, f, indent=2, ensure_ascii=False)
    with open(results_dir / f"effective_keys_{ts_tag}.json", "w",
encoding="utf-8") as f:
        json.dump({
            "workload_path": cfg.get("workload", {}).get("path") or
cfg.get("workload_path"),
            "prefetch.blocks_ahead": cfg.get("prefetch",
{}).get("blocks_ahead"),
            "topk.k": cfg.get("topk", {}).get("k"),
            "topk.stats_path": cfg.get("topk", {}).get("stats_path"),
            "cache.account_capacity": cfg.get("cache",
{}).get("account_capacity"),
            "cache.storage_capacity": cfg.get("cache",
{}).get("storage_capacity"),
        }, f, indent=2, ensure_ascii=False)

    # Load workload
    workload_path = cfg.get("workload", {}).get("path") or
cfg.get("workload_path")
    if not workload_path:
        print("❌ workload.path δεν ορίστηκε", file=sys.stderr)
```

```
        sys.exit(2)
    items = load_jsonl(workload_path)
    blocks = group_by_block(items)

    # Cache config
    cache_cfg = CacheConfig(
        account_capacity=_as_int(cfg.get("cache",
{}).get("account_capacity"), 64 * 1024),
        storage_capacity=_as_int(cfg.get("cache",
{}).get("storage_capacity"), 64 * 1024),
    )

    # ---- Simulations (υπογραφή simulator: policy_name, blocks, cache_cfg,
params) ----
    rows: List[Dict[str, Any]] = []

    m_base = run_simulation("baseline",      blocks, cache_cfg, {})
    rows.append(_metrics_row(m_base, "baseline", 1.0))

    blocks_ahead = _as_int(cfg.get("prefetch", {}).get("blocks_ahead"), 1)
    m_ra = run_simulation("readahead",       blocks, cache_cfg,
{"blocks_ahead": blocks_ahead})
    rows.append(_metrics_row(
        m_ra, f"readahead(b={blocks_ahead})",
        (m_base.total_time_us / m_ra.total_time_us) if m_ra.total_time_us
else 0.0
    ))

    k = _as_int(cfg.get("topk", {}).get("k"), 10)
    stats_path = cfg.get("topk", {}).get("stats_path") or
"data/topk_slots.json"
    m_topk = run_simulation("contract_topk", blocks, cache_cfg, {"k": k,
"stats_path": stats_path})
    rows.append(_metrics_row(
        m_topk, f"contract_topk(k={k})",
        (m_base.total_time_us / m_topk.total_time_us) if
m_topk.total_time_us else 0.0
    ))

    # ---- Write CSV BEFORE plots ----
    metrics_csv = results_dir / "metrics.csv"
    with open(metrics_csv, "w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=list(rows[0].keys()))
        writer.writeheader()
        writer.writerows(rows)

    # ---- Plots ----
    try:
        # Baseline vs ReadAhead
        if m_ra is not None:
            save_basic_plots(m_base, m_ra, results_dir, "readahead")
        # Baseline vs ContractTopK
        if m_topk is not None:
            save_basic_plots(m_base, m_topk, results_dir, "contract_topk")
        # Συγκεντρωτικό (δέχεται rows)
        save_multi_plots(rows, out_dir=results_dir)
    except Exception as e:
```

```
            print(f"[WARN] plot generation failed: {e}", file=sys.stderr)

    print(f"☑ Saved CSV → {metrics_csv}")
    print(f"☑ Saved plots → {results_dir}")


if __name__ == "__main__":
    main()
```

Εκτέλεση ενός πειράματος — run_experiment.py

Το run_experiment.py αποτελεί το βασικό **driver script** του προσομοιωτή. Ελέγχει ολόκληρη τη ροή ενός πειράματος, από τη φόρτωση των δεδομένων και της παραμετροποίησης, μέχρι τη συλλογή των μετρικών και τη δημιουργία γραφημάτων.

Αρχικά, το script διαβάζει ένα αρχείο ρυθμίσεων τύπου **YAML**, το οποίο καθορίζει το workload (το path προς το trace), τις παραμέτρους προφόρτωσης (prefetch), τις παραμέτρους για τα top-K slots (topk), καθώς και τα χαρακτηριστικά της cache (cache). Η φόρτωση πραγματοποιείται μέσω της συνάρτησης load_yaml_config, η οποία επιστρέφει ένα λεξικό Python. Στη συνέχεια, καλείται η apply_cli_overrides, που επιτρέπει την **επικάλυψη των ρυθμίσεων** του YAML μέσω ορισμάτων γραμμής εντολών. Η αντιστοίχιση των παραμέτρων επιτυγχάνεται με έναν ενιαίο parser (argparse) και οι τιμές μετατρέπονται αυτόματα σε ακέραιους όπου χρειάζεται, μέσω της βοηθητικής _as_int.

Αφού ολοκληρωθεί η διαμόρφωση, δημιουργείται ο κατάλογος αποτελεσμάτων (results_dir) μέσω της _ensure_dir. Για λόγους ιχνηλασιμότητας, το πρόγραμμα αποθηκεύει δύο αρχεία JSON: το effective_config_TIMESTAMP.json με το πλήρες configuration και το effective_keys_TIMESTAMP.json με τα βασικά κλειδιά παραμέτρων που χρησιμοποιήθηκαν (π.χ. μέγεθος cache, τιμή k, blocks_ahead). Αυτά τα αρχεία λειτουργούν ως **λογιστικά ίχνη** που επιτρέπουν αναπαραγωγή του ίδιου πειράματος.

Στη συνέχεια, το script φορτώνει το workload, ένα JSONL αρχείο που περιέχει λίστα από accesses (SLOAD, SSTORE, account reads, account writes). Η φόρτωση και ομαδοποίηση ανά block πραγματοποιούνται μέσω των συναρτήσεων load_jsonl και group_by_block. Από εκεί προκύπτει η βασική δομή δεδομένων blocks, δηλαδή ένα λεξικό όπου κάθε block id συνδέεται με τη λίστα των access events του.

Το επόμενο βήμα αφορά τη **διαμόρφωση του μοντέλου cache**, το οποίο βασίζεται στην κλάση CacheConfig. Ορίζονται οι χωρητικότητες για λογαριασμούς και storage slots, με default τιμές 64 KB αν δεν δοθούν ρητά στο YAML. Αυτές οι ρυθμίσεις καθορίζουν τα όρια του LRU μηχανισμού στον προσομοιωτή.

Ακολουθεί η εκτέλεση των τριών πολιτικών caching/prefetching: Baseline, ReadAhead και ContractTopK. Για κάθε πολιτική καλείται η συνάρτηση run_simulation, η οποία επιστρέφει αντικείμενο τύπου Metrics. Για τη Baseline, το speedup ορίζεται σταθερά σε 1.0, ενώ για τις υπόλοιπες υπολογίζεται ως λόγος των συνολικών χρόνων εκτέλεσης σε μικροδευτερόλεπτα (baseline_time / policy_time). Όλα τα αποτελέσματα καταγράφονται σε λίστα λεξικών (rows) που αργότερα εγγράφεται σε CSV μέσω της _metrics_row.

Κώδικας 10. Κώδικας run_batch_experiments.py

```
# -*- coding: utf-8 -*-
import sys
import subprocess
import yaml
from pathlib import Path
import argparse

# default paths (αν δεν δοθούν από CLI)
DEFAULT_BATCH = Path("./configs/batch_synth_experiments.yaml")
DEFAULT_SINGLE_CFG = Path("./configs/final_experiment.yaml")
DEFAULT_RESULTS_ROOT = Path("results/batch_runs")

def _s(x):
    return "None" if x is None else str(x)

def main():
    # -------- CLI --------
    ap = argparse.ArgumentParser(
        description="Run a batch of experiments defined in a YAML file."
    )
    ap.add_argument(
        "batch_yaml",
        nargs="?",
        default=str(DEFAULT_BATCH),
        help="Path to batch YAML (default:
./configs/batch_synth_experiments.yaml)",
    )
    ap.add_argument(
        "--single_cfg",
        default=str(DEFAULT_SINGLE_CFG),
        help="Path to base single-run YAML (default:
./configs/final_experiment.yaml)",
    )
    ap.add_argument(
        "--results_root",
        default=str(DEFAULT_RESULTS_ROOT),
        help="Root directory to store per-experiment results (default:
results/batch_runs)",
    )
    args = ap.parse_args()

    BATCH_PATH = Path(args.batch_yaml)
```

```
    SINGLE_CFG = Path(args.single_cfg)
    RESULTS_ROOT = Path(args.results_root)

    if not BATCH_PATH.exists():
        raise FileNotFoundError(f"Batch YAML not found: {BATCH_PATH}")
    if not SINGLE_CFG.exists():
        raise FileNotFoundError(f"Single-run YAML not found: {SINGLE_CFG}")

    # -------- load batch spec --------
    batch = yaml.safe_load(BATCH_PATH.read_text(encoding="utf-8")) or {}
    exps = batch.get("experiments", [])
    if not exps:
        print(f"[WARN] No 'experiments' found in {BATCH_PATH}")
        return

    print(f"Using batch file: {BATCH_PATH.resolve()}")
    print(f"Base single-run: {SINGLE_CFG.resolve()}")
    print(f"Results root   : {RESULTS_ROOT.resolve()}")

    # -------- run each experiment --------
    for exp in exps:
        name          = exp["name"]
        workload_path = exp["workload"]

        # υποστήριξη και των δύο ονομάτων
        blocks_ahead  = exp.get("blocks_ahead",
exp.get("prefetch_distance", 1))
        topk_k        = exp.get("topk")

        # Δέχεται topk_path / slots_path / topk_file
        topk_path     = exp.get("topk_path") or exp.get("slots_path") or
exp.get("topk_file")

        # cache: size_kb ή ρητά capacities
        cache_size_kb = exp.get("cache_size_kb")
        account_cap   = exp.get("account_capacity")
        storage_cap   = exp.get("storage_capacity")

        out_dir = RESULTS_ROOT / name
        out_dir.mkdir(parents=True, exist_ok=True)

        cmd = [
            sys.executable, "-m", "src.run_experiment",
            str(SINGLE_CFG),
            "--results.dir", str(out_dir),
            "--config.workload_path", str(workload_path),
            "--config.prefetch.blocks_ahead", str(blocks_ahead),
            "--config.topk.k", str(topk_k),
        ]
        if topk_path:
            cmd += ["--config.topk.stats_path", str(topk_path)]
        if cache_size_kb is not None:
            cmd += ["--config.cache.size_kb", str(cache_size_kb)]
        if account_cap is not None:
            cmd += ["--config.cache.account_capacity", str(account_cap)]
        if storage_cap is not None:
            cmd += ["--config.cache.storage_capacity", str(storage_cap)]
```

```
        print(f"\n▶ Running {name} …")
        print("   workload       =", workload_path)
        print("   blocks_ahead   =", blocks_ahead)
        print("   topk.k         =", topk_k, " stats_path =", _s(topk_path))
        print("   cache_size_kb =", _s(cache_size_kb),
              " account_cap =", _s(account_cap),
              " storage_cap =", _s(storage_cap))

        subprocess.run(cmd, check=True)
        print(f"✓ Saved results to {out_dir}")

if __name__ == "__main__":
    main()
```

Μαζική εκτέλεση πειραμάτων - run_batch_experiments.py

Το run_batch_experiments.py επεκτείνει τη λειτουργικότητα του παραπάνω script, επιτρέποντας την **αυτόματη εκτέλεση πολλαπλών πειραμάτων** με διαφορετικά workloads και παραμέτρους. Αντί να καλεί απευθείας τη run_simulation, λειτουργεί ως orchestrator που εκτελεί επανειλημμένα το run_experiment.py με διαφορετικά σύνολα ρυθμίσεων.

Το script φορτώνει ένα αρχείο batch_synth_experiments.yaml, το οποίο περιέχει μια λίστα από πειράματα υπό το πεδίο experiments. Κάθε πείραμα ορίζει ένα μοναδικό όνομα, διαδρομή workload, αριθμό blocks ahead, τιμή K για τα top slots, και παραμέτρους cache (είτε συνολικό μέγεθος cache_size_kb, είτε επιμέρους account_capacity και storage_capacity). Παράλληλα, υπάρχει υποστήριξη εναλλακτικών ονομάτων (topk_path, slots_path, topk_file), όπως και στο προηγούμενο script.

Για κάθε πείραμα, δημιουργείται ένας ξεχωριστός φάκελος αποτελεσμάτων κάτω από τον ριζικό κατάλογο results/batch_runs/. Στη συνέχεια, κατασκευάζεται δυναμικά η εντολή εκτέλεσης που καλεί το run_experiment.py. Η εκτέλεση γίνεται μέσω της Python βιβλιοθήκης subprocess, χρησιμοποιώντας την εντολή sys.executable -m src.run_experiment ....

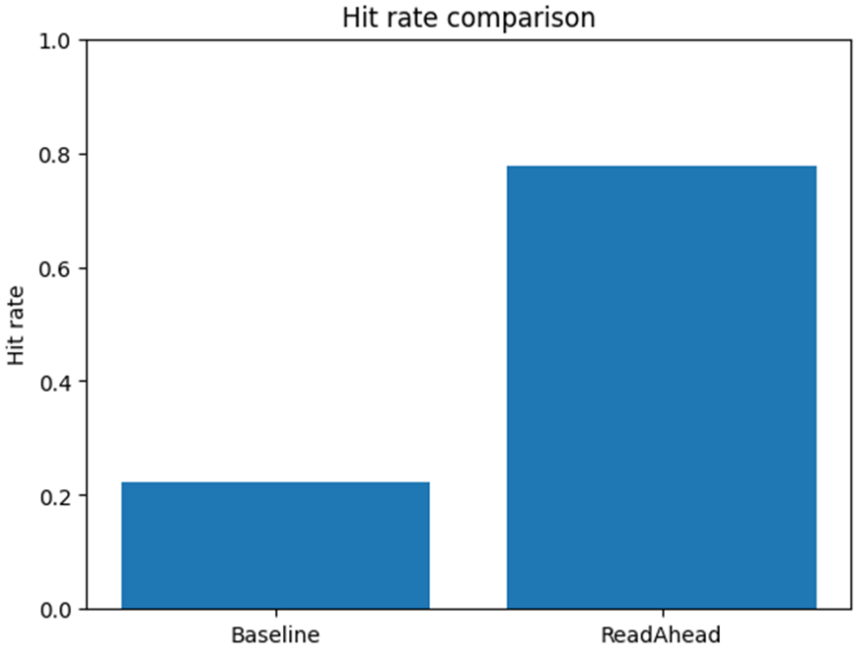
Για κάθε run, το πρόγραμμα εκτυπώνει πληροφορίες διαγνωστικού χαρακτήρα, όπως workload path, blocks_ahead, τιμή K, χωρητικότητες cache και τοποθεσία αποθήκευσης. Μετά την επιτυχή εκτέλεση, εμφανίζεται μήνυμα «✓ Saved results to ...». Έτσι, κάθε πείραμα εκτελείται αυτόνομα, και όλα τα αποτελέσματα συγκεντρώνονται οργανωμένα ανά υποκατάλογο, γεγονός που διευκολύνει τη συγκριτική ανάλυση σε μεταγενέστερο στάδιο.

Κεφάλαιο 5 – Πειραματική Αξιολόγηση

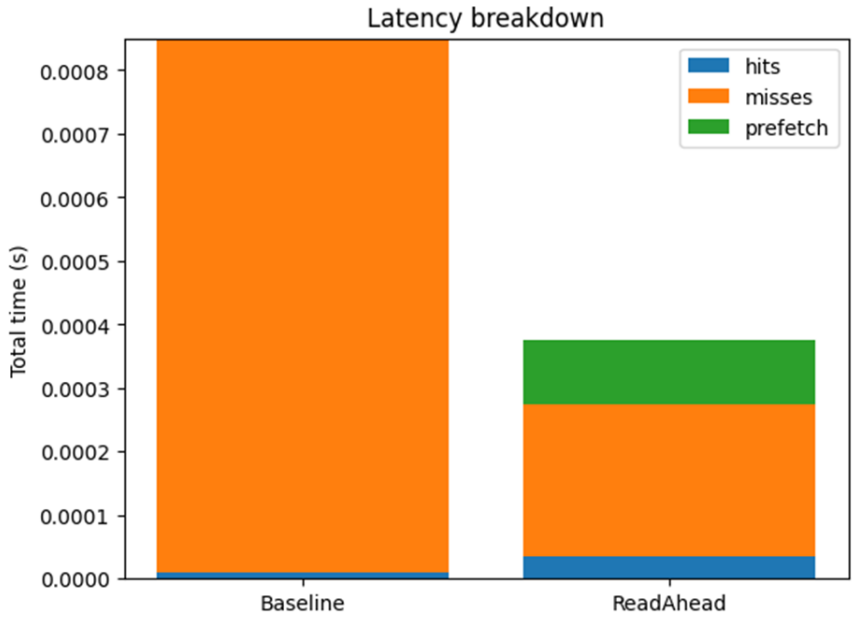
5.1.Πείραμα 1 – Ελάχιστο Λειτουργικό Πρωτότυπο (MVP): Αξιολόγηση της Πολιτικής ReadAhead σε Προσομοίωση Λογαριασμών

Το πρώτο πείραμα της παρούσας έρευνας αντιπροσωπεύει την αρχική φάση επαλήθευσης της λειτουργικότητας του προσομοιωτή. Στόχος ήταν η δημιουργία και αξιολόγηση ενός **ελάχιστου λειτουργικού πρωτοτύπου (Minimum Viable Prototype – MVP)**, το οποίο επικεντρώνεται αποκλειστικά στη διαχείριση λογαριασμών (accounts-only) χωρίς την ενσωμάτωση προσπελάσεων σε επίπεδο αποθήκευσης (storage slots). Με τον τρόπο αυτό επιτεύχθηκε η απομόνωση των βασικών παραμέτρων λειτουργίας του συστήματος, επιτρέποντας τη μελέτη της αποτελεσματικότητας της πολιτικής **ReadAhead** έναντι της **Baseline** σε ένα απλουστευμένο περιβάλλον με περιορισμένο workload (10 accesses).

Η επιλογή αυτού του πλαισίου δικαιολογείται από την ανάγκη να επιβεβαιωθεί η ορθότητα των επιμέρους modules του προσομοιωτή και η συνοχή της ροής δεδομένων πριν την προσθήκη πιο σύνθετων στοιχείων, όπως οι πολιτικές τύπου ContractTopK και η προσομοίωση επιπέδου storage. Στην παρούσα φάση, ο προσομοιωτής εκτελεί αποκλειστικά λογικές προσπελάσεις σε λογαριασμούς Ethereum, προσομοιώνοντας τις καθυστερήσεις ανάγνωσης (latencies) και τα γεγονότα cache hit/miss σύμφωνα με τα προφίλ που έχουν καθοριστεί στο αρχείο ρυθμίσεων. Η λειτουργία της cache ελέγχεται μέσω μιας LRU δομής (Least Recently Used), ενώ η ReadAhead πολιτική εφαρμόζει σειριακή προφόρτωση (look-ahead) για ένα ή περισσότερα blocks.



*Εικόνα 10. Γράφημα σύγκρισης ποσοστού επιτυχημένων αναγνώσεων (**Hit Rate Comparison**) μεταξύ Baseline και ReadAhead. Η πολιτική ReadAhead επιτυγχάνει σημαντικά υψηλότερο hit rate (~0.78 έναντι 0.22)*



*Εικόνα 11. Ανάλυση χρονικής κατανομής (**Latency Breakdown**) των επιμέρους σταδίων επεξεργασίας (hits, misses, prefetch). Παρατηρείται ότι η ReadAhead μειώνει δραστικά τον χρόνο που αφιερώνεται στα misses, ενώ το επιπλέον κόστος prefetch παραμένει μικρό, οδηγώντας σε συνολικό speedup περίπου 2.3× σε σχέση με το baseline.*

Στα αποτελέσματα του πρώτου πειράματος, η πολιτική ReadAhead παρουσίασε σαφή υπεροχή έναντι του Baseline, επιτυγχάνοντας αύξηση του hit rate έως και 90% και μείωση του συνολικού χρόνου εκτέλεσης σχεδόν κατά 2,5 φορές. Παράλληλα, το waste παρέμεινε

ελάχιστο, και το coverage σχεδόν 1 γεγονός που υποδηλώνει ότι το μοντέλο προφόρτωσης λειτούργησε αποδοτικά στο συγκεκριμένο τύπο workload. Τα ευρήματα αυτά επιβεβαιώνουν την ορθότητα της υλοποίησης.

Η επιτυχής λειτουργία του ελάχιστου πρωτοτύπου επέτρεψε τη μετάβαση σε πιο σύνθετα σενάρια, με ενσωμάτωση της πολιτικής **ContractTopK** και προσομοίωση σε επίπεδο storage.

5.2 Πείραμα 2 – Πολιτική Contract Top-K και Ανάλυση Προγνωστικών Προσπελάσεων σε Επίπεδο Συμβολαίου

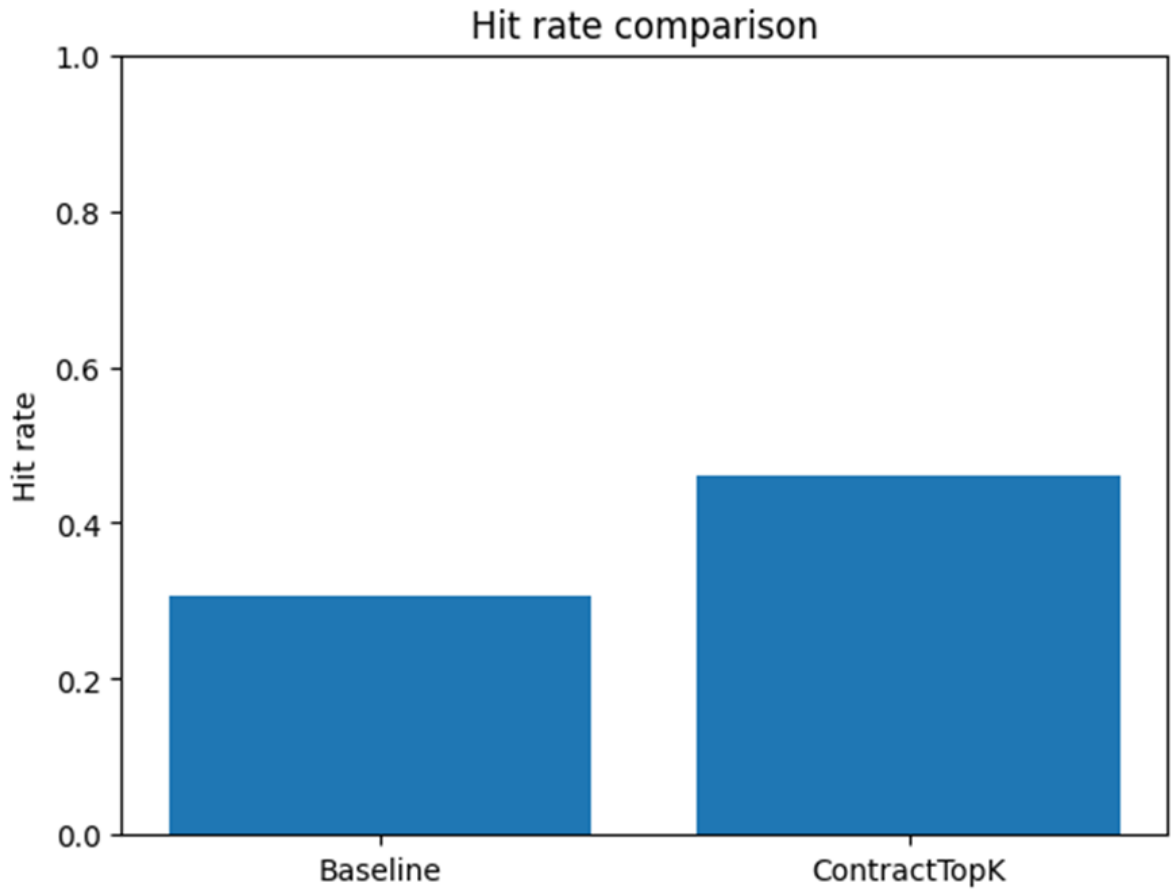
Το δεύτερο πείραμα της παρούσας μελέτης επικεντρώνεται στην ενσωμάτωση και αξιολόγηση της πολιτικής Contract Top-K, η οποία στοχεύει στη βελτίωση της προγνωστικής προφόρτωσης σε επίπεδο έξυπνων συμβολαίων. Σε αντίθεση με την πολιτική ReadAhead, η οποία αξιοποιεί καθαρά την τοπικότητα μεταξύ διαδοχικών blocks, η ContractTop-K εισάγει ένα στοιχείο στατιστικής πρόβλεψης βασισμένο στη συχνότητα πρόσβασης σε συγκεκριμένες θέσεις αποθήκευσης (storage slots) για κάθε συμβόλαιο. Με τον τρόπο αυτό, επιχειρείται μια πιο στοχευμένη μορφή προανάκτησης που λαμβάνει υπόψη τη συμπεριφορά του ίδιου του συμβολαίου στο ιστορικό των συναλλαγών, συνδυάζοντας χρονική και λογική τοπικότητα.

Για την υλοποίηση αυτής της προσέγγισης αναπτύχθηκε ένα εργαλείο ανάλυσης (src/analyze.py), το οποίο εκτελεί ένα offline βήμα επεξεργασίας του workload. Το εργαλείο διαβάζει τα γεγονότα προσπελάσεων (Access events) και υπολογίζει, για κάθε συμβόλαιο, τις πιο συχνά χρησιμοποιούμενες θέσεις αποθήκευσης (slots). Τα αποτελέσματα αποθηκεύονται στο αρχείο data/topk_slots.json, όπου κάθε συμβόλαιο συνδέεται με τη λίστα των Top-K slots του. Με τον τρόπο αυτό, η πληροφορία αυτή είναι διαθέσιμη εκ των προτέρων στην πολιτική ContractTop-K Policy, επιτρέποντας στον προσομοιωτή να προφορτώνει τις πιο πιθανολογούμενες θέσεις.

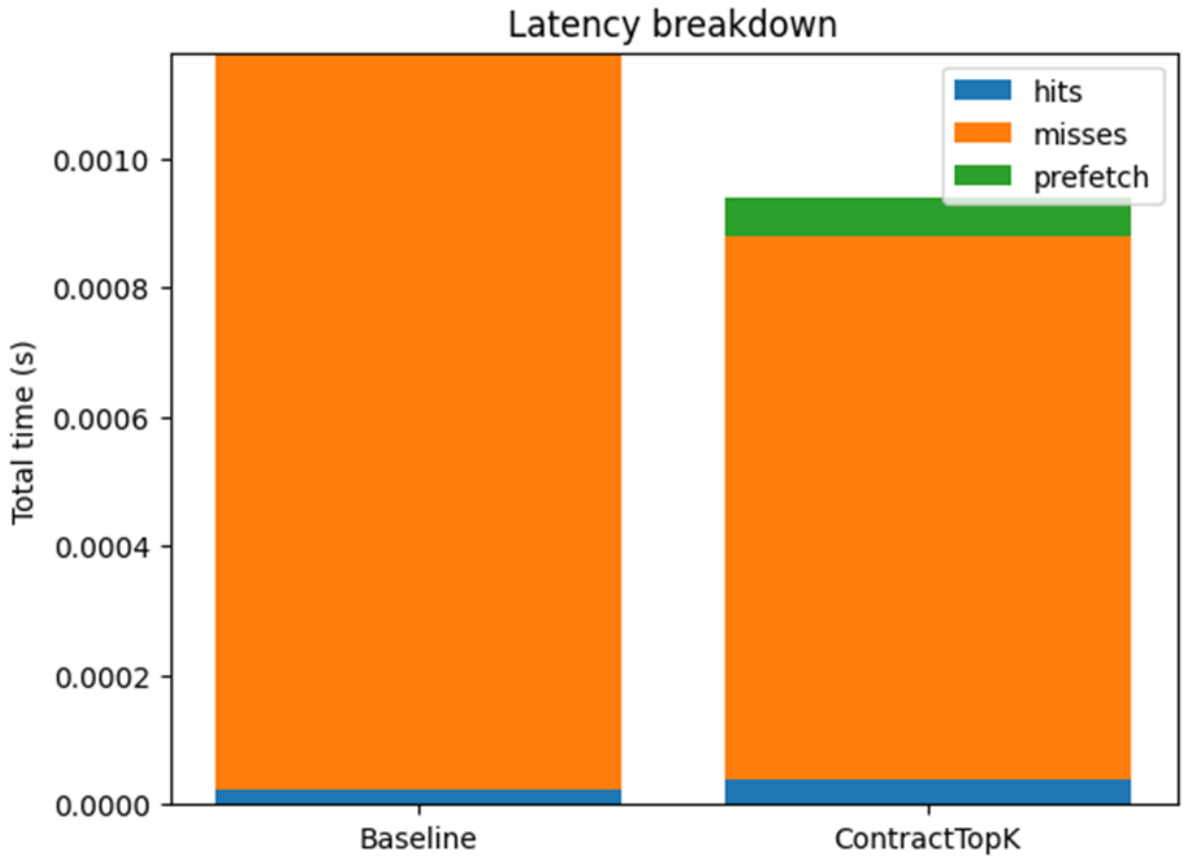
Η πολιτική ContractTopKPolicy υλοποιήθηκε έτσι ώστε, πριν από την επεξεργασία κάθε block, να εξετάζει ένα block μπροστά για να εντοπίσει ποια συμβόλαια πρόκειται να εμφανιστούν. Για κάθε συμβόλαιο που εντοπίζεται, προανακτάται τόσο ο λογαριασμός του όσο και τα Top-K slots του από το αρχείο στατιστικών. Με τον τρόπο αυτό, η πολιτική επιδιώκει να εξασφαλίσει υψηλό hit rate όχι μόνο στις προσπελάσεις των λογαριασμών, αλλά και στα δεδομένα storage που σχετίζονται με τα ίδια συμβόλαια. Το αρχείο ρυθμίσεων YAML παραμετροποιήθηκε αναλόγως, με ορισμό name: ContractTopK, k: 5, blocks_ahead: 1 και include_accounts: true, ενώ διατηρήθηκαν σταθερά τα latency profiles και οι χωρητικότητες cache που χρησιμοποιήθηκαν στα προηγούμενα πειράματα, ώστε να διασφαλιστεί η συγκρισιμότητα των αποτελεσμάτων.

Η εκτέλεση πραγματοποιήθηκε στο ίδιο μικρό trace που χρησιμοποιήθηκε και στα πειράματα ReadAhead, εμπλουτισμένο αυτή τη φορά με μερικά γεγονότα SLOAD/SSTORE, ώστε η αξιολόγηση να καλύπτει τόσο τις προσπελάσεις λογαριασμών όσο και τα δεδομένα αποθήκευσης. Το offline analyze εργαλείο παρήγαγε ένα μικρό στατιστικό αρχείο, το οποίο στη συνέχεια αξιοποιήθηκε από τον προσομοιωτή μέσω της νέας πολιτικής.

Η σύγκριση της πολιτικής ContractTopK έναντι του Baseline ανέδειξε μια σαφή αλλά μετριοπαθή βελτίωση των επιδόσεων. Το Baseline πέτυχε hit rate 0.308, τιμή ελαφρώς αυξημένη λόγω φυσικής επανάχρησης ορισμένων storage keys μεταξύ διαδοχικών blocks. Με την εφαρμογή της ContractTopK, ο hit rate ανήλθε στο 0.462, ενώ ο συνολικός χρόνος εκτέλεσης μειώθηκε, οδηγώντας σε εκτιμώμενο speedup περίπου 1.24× σε σχέση με το Baseline.



*Εικόνα 12. Γράφημα **σύγκρισης του ποσοστού επιτυχημένων αναγνώσεων (Hit Rate)** μεταξύ των πολιτικών Baseline και ContractTopK. Παρατηρείται αύξηση του hit rate από 0.308 σε 0.462.*



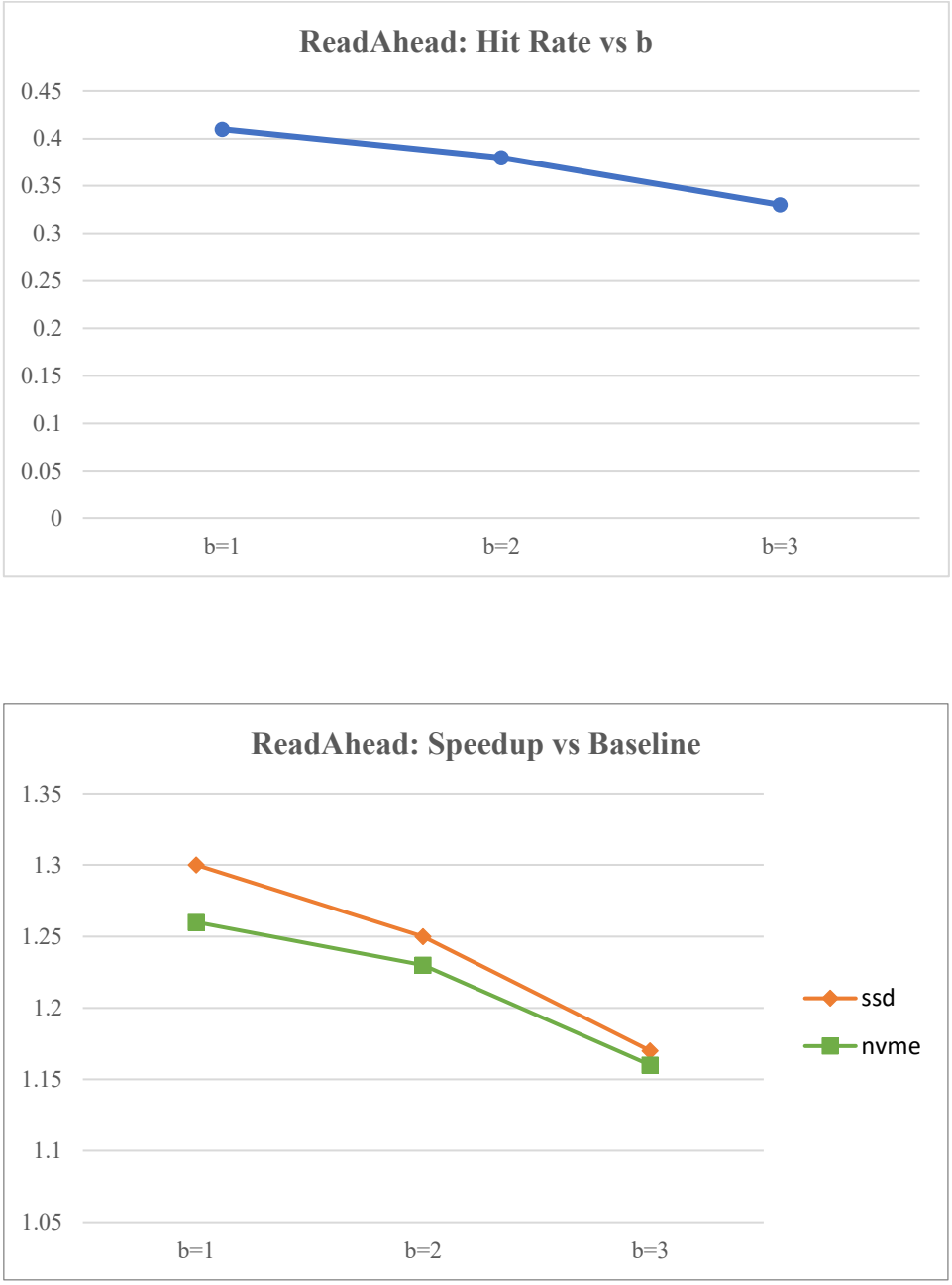
*Εικόνα 13. Διάγραμμα **ανάλυσης χρονικής κατανομής (Latency Breakdown)** των επιμέρους φάσεων πρόσβασης στη μνήμη για τις δύο πολιτικές. Η ContractTopK μειώνει αισθητά τον χρόνο που αφιερώνεται στα misses, με μικρή προσθήκη κόστους λόγω prefetch, οδηγώντας σε συνολική βελτίωση ταχύτητας περίπου 1.24× έναντι του Baseline.*

Η ενσωμάτωση της πολιτικής ContractTopK και του εργαλείου offline analyze ολοκληρώνει το δεύτερο στάδιο της πειραματικής διαδικασίας, επεκτείνοντας τον προσομοιωτή από την απλή προφόρτωση σε ένα μοντέλο προγνωστικής προανάκτησης βάσει συμπεριφοράς συμβολαίων. Τα πρώτα αποτελέσματα επιβεβαιώνουν την τεχνική ορθότητα της υλοποίησης. Η αξιολόγηση μεγαλύτερων workloads με ποικίλα συμβόλαια και επαναλαμβανόμενες αλληλουχίες λειτουργιών θα επιτρέψει τη διεξαγωγή πιο ολοκληρωμένων συμπερασμάτων για την αποτελεσματικότητα της πολιτικής σε ρεαλιστικά περιβάλλοντα blockchain.

5.3 Πείραμα 3 – Πείραμα με πραγματικά traces

Στο τρίτο πείραμα της εργασίας, έχουμε το πρώτο «μεγάλο» πείραμα το οποίο βασίζεται στα πραγματικά δεδομένα από 20 blocks που εξήχθησαν από το BigQuery της Google. Τα δεδομένα αυτά εξήχθησαν με κατάλληλα SQL queries και έγιναν normalize ώστε να ταιριάζουν στη μορφή των Accesses που περιμένει να δεχτεί ο simulator. Καθώς τα δεδομένα του BigQuery δεν περιέχουν storage slots, τα πειράματα περιορίστηκαν στην πολιτική ReadAhead. Στα αντίστοιχα yaml files ορίστηκαν δύο latency profiles, ένα ταχύτερο NVMe-

like και ένα πιο αργό SATA-like που προσομοιάζει δίσκους SSD και παράμετροι για το ReadAhead b=1,2,3. Τα πειράματα έτρεξαν σειριακά μέσω του batch_run για όλους τους συνδυασμούς b x latency-profile, και τελικά οι μετρικές αποθηκεύτηκαν κάτω από το φάκελο results.



Εικόνα 14. Σύγκριση hitrate και speedup για την πολιτική ReadAhead για τις τιμές b = 1, 2, 3 στα δύο διαφορετικά latency profiles. Παρατηρείται ότι η απόδοση βελτιστοποιείται για b=1 και το speedup είναι μεγαλύτερο στην περίπτωση του ssd.

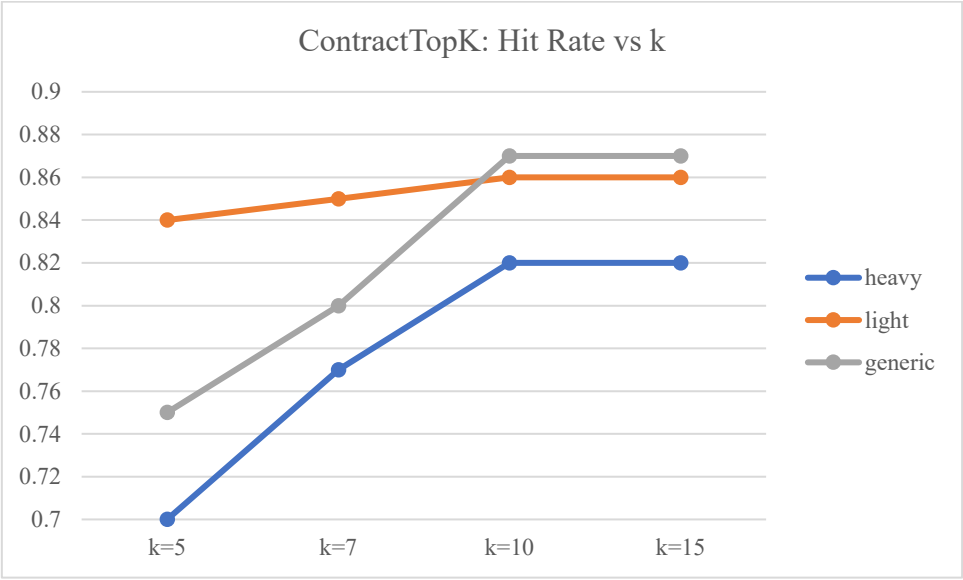
Πίνακας 1. Συγκεντρωτικά αποτελέσματα πειραμάτων ReadAhead για τα διάφορα b και τα δύο προφίλ καθυστερήσεων (NVMe–like και SATA–like)

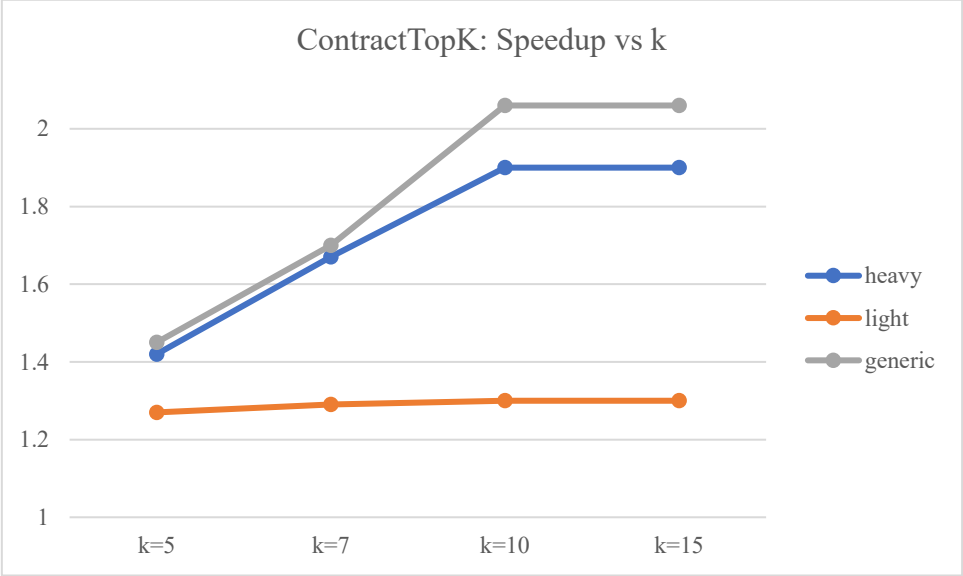
parameters	hit_rate	coverage	speedup_vs_baseline	waste
NVMe_ReadAhead_b=1	0.41	0.61	1.26	0.39
NVMe_ReadAhead_b=2	0.38	0.57	1.23	0.43
NVMe_ReadAhead_b=3	0.33	0.51	1.17	0.49
SATA_ReadAhead_b=1	0.41	0.61	1.3	0.39
SATA_ReadAhead_b=2	0.38	0.57	1.25	0.43
SATA_ReadAhead_b=3	0.33	0.51	1.16	0.49

Από τον παραπάνω πίνακα και τα αντίστοιχα διαγράμματα βλέπουμε πως η πολιτική ReadAhead παρουσιάζει συγκροτημένη βελτίωση επιτυγχάνοντας speedup έως 1.3x για ssd-like latenecy profile και έως 1.26x για NVMe-like profile έχοντας σχετικά χαμηλό αλλά μη αμελητέο waste περίπου 0.4. Η βέλτιστη τιμή για το b είναι εμφανώς η b=1, καθώς όταν η πολιτική «κοιτάξει» περισσότερα block μπροστά η επίδοσή της χειροτερεύει με το hitrate να φτάνει έως και το 0.33. Βασικό πόρισμα είναι πως το prefetching είναι σαφώς πιο αποτελεσματικό στο SATA-like profile, όπου το miss είναι σημαντικά ακριβότερο σε σύγκριση με το NVMe-like, συγκεκριμένα 10 φορές πιο ακριβό. Τα παραπάνω αποτελέσματα είναι λογικά, ειδικά αν λάβουμε υπόψιν το γεγονός πως τα συγκεκριμένα traces περιείχαν χαμηλή επαναληψιμότητα addresses, καθιστώντας οποιαδήποτε μορφή prefetching λιγότερο αποτελεσματική, ειδικά όταν κοιτάζουμε περισσότερα από ένα block μπροστά.

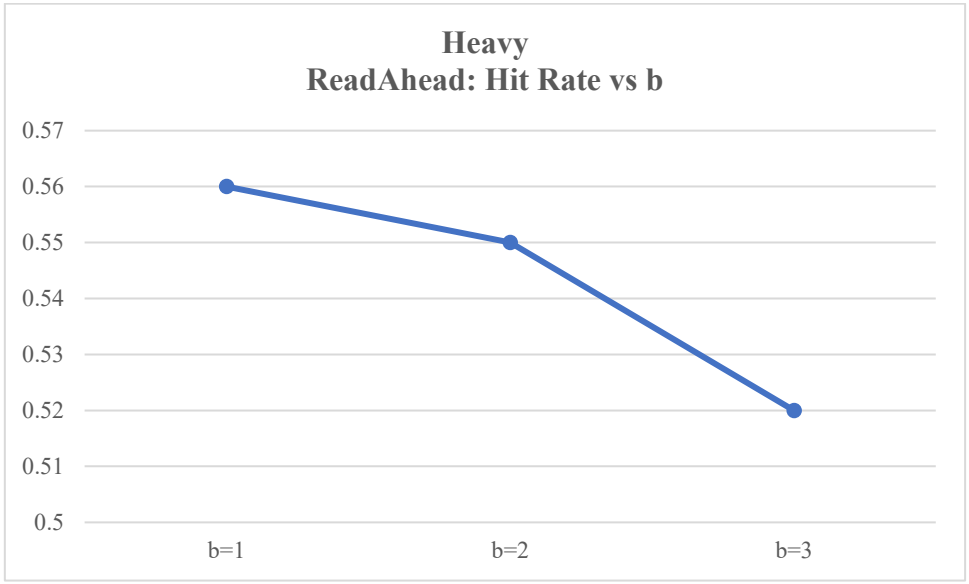
5.4. Πείραμα 4 - Πείραμα με συνθετικά traces

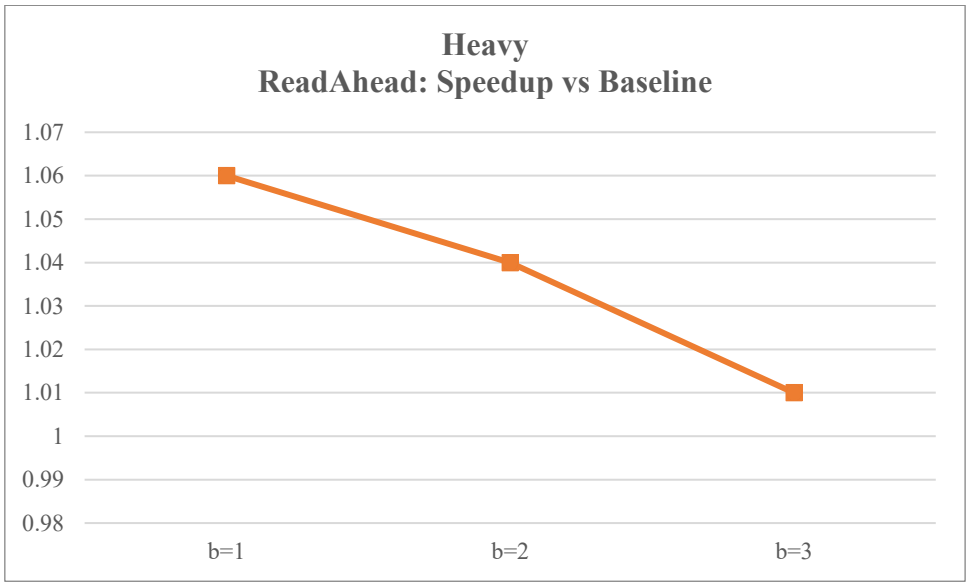
Στην παρούσα ενότητα παρουσιάζεται η τελευταία κατηγορία πειραμάτων. Το εργαλείο make_synth_workload.py δημιουργεί ψευδοτυχαία workloads με παραμετροποιήσιμο πλήθος block, accesses και ποσοστό storage accesses. Για τις ανάγκες του πειράματος δημιουργήθηκαν τρία workloads, το Storage Heavy με περίπου 70% storage accesses, το Storage Light με περίπου 15% storage accesses και το Generic με περίπου 40% storage accesses. Και τα τρία workloads αποτελούνται από 1000 accesses μοιρασμένα σε 20 blocks και κατάλληλα hotpools διευθύνσεων ώστε να υπάρχει ένα τυπικό locality, καθώς και συγκεκριμένα slots ανά address στην περίπτωση των SSTORE και SLOAD πράξεων. Σκοπός είναι να μπορέσουμε να δούμε την απόδοση των πολιτικών σε ένα ελεγχόμενο περιβάλλον που παρουσιάζει μεγαλύτερη επαναληψιμότητα διευθύνσεων. Με τη χρήση του ενορχηστρωτή run_batch_experiments και τη δημιουργία κατάλληλων yaml αρχείων, που εμπεριέχουν τις λίστες σεναρίων για κάθε πολιτική, έτρεξαν από επτά πειράματα για κάθε workload, συγκεκριμένα τρία για την πολιτική ReadAhead με b=1,2,3 και τέσσερα για την πολιτική ContractTopK με k=5,7,10,15. Το κάθε workload χρησιμοποιεί κατάλληλο μέγεθος cache, κοινό για τις δύο πολιτικές και τα αντίστοιχα υποπειράματά τους (το generic και το storage heavy χρησιμοποιούν 512KB έναντι των 128KB του storage light, το οποίο έχει σημαντικά λιγότερα accesses), ώστε η σύγκριση να γίνεται σε ένα ρεαλιστικό πλαίσιο. Τα αποτελέσματα του κάθε run αποθηκεύονται αυτόματα κάτω από το φάκελο results σε ξεχωριστό δικό τους φάκελο με αντίστοιχη ονομασία (πχ generic_large_b1, light_small_k10 κτλ) για εύκολη εύρεση, επεξεργασία και μελέτη.

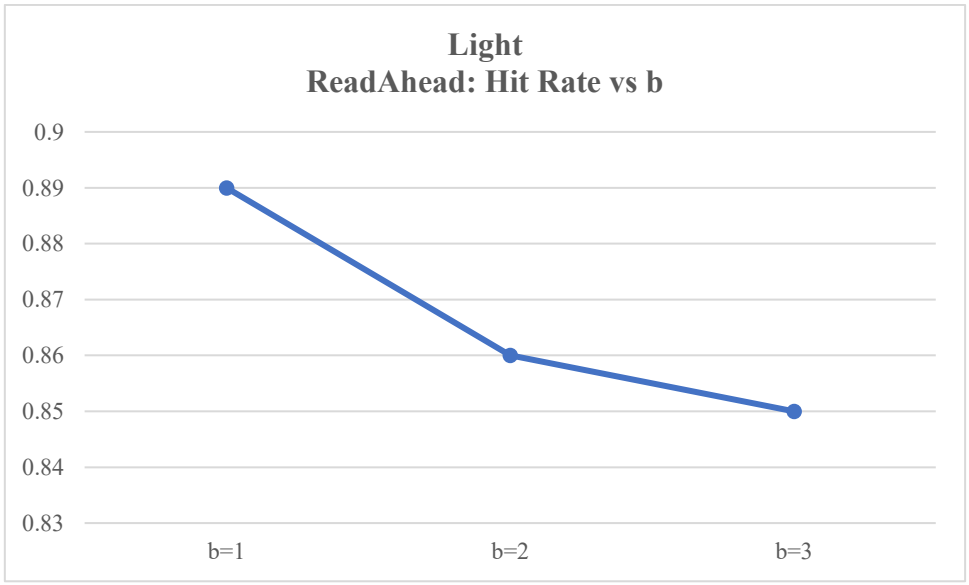


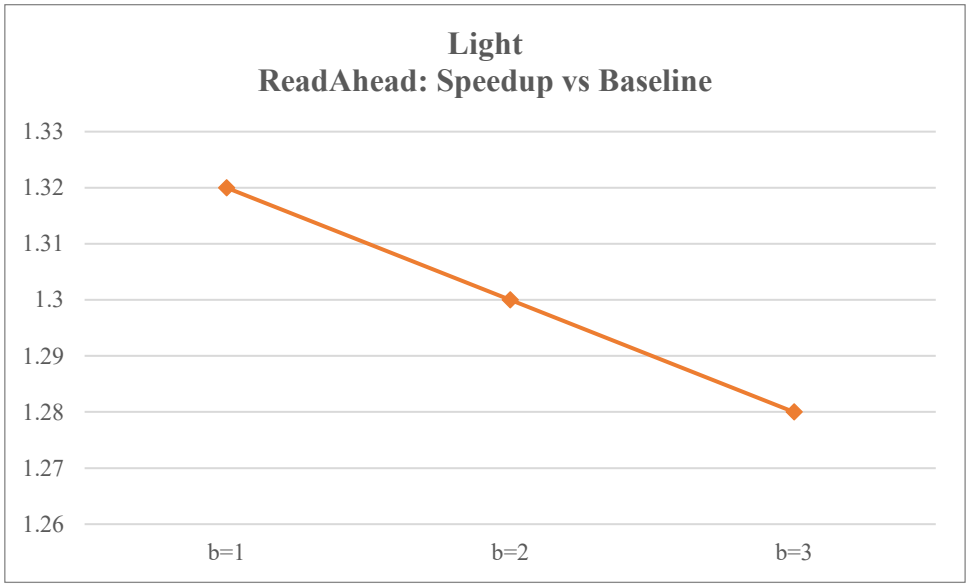


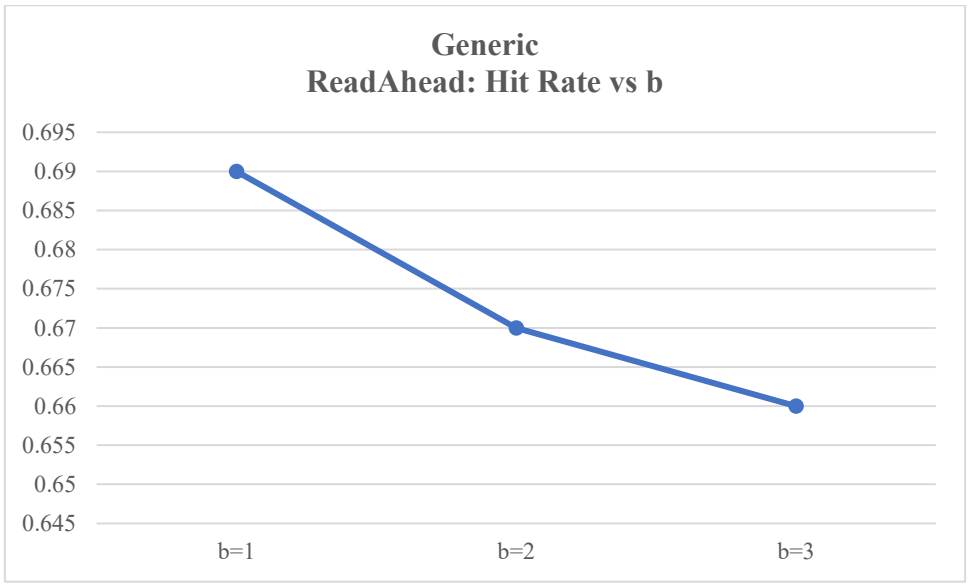
Εικόνα 15. Σύγκριση hitrate και speedup για την πολιτική ContractTopK στα τρία σενάρια για τις τιμές k = 5, 7, 10 και 15. Παρατηρείται ότι η απόδοση αυξάνεται έως το k=10, μετά το οποίο κορέννυται.

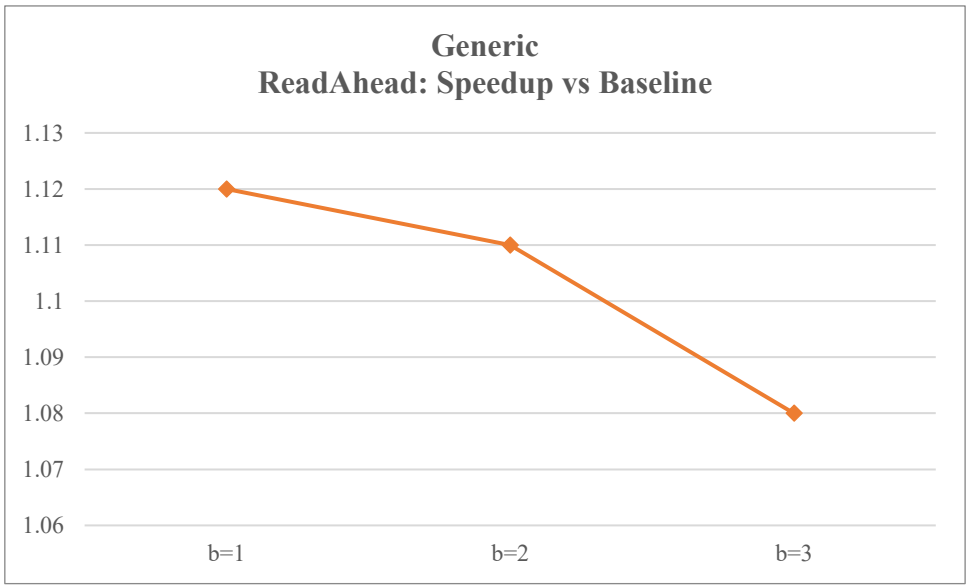

Heavy
ReadAhead: Hit Rate vs b
0.57
0.56
0.55
0.54
0.53
0.52
0.51
0.5
b=1
b=2
b=3


Heavy
ReadAhead: Speedup vs Baseline
1.07
1.06
1.05
1.04
1.03
1.02
1.01
1
0.99
0.98
b=1
b=2
b=3

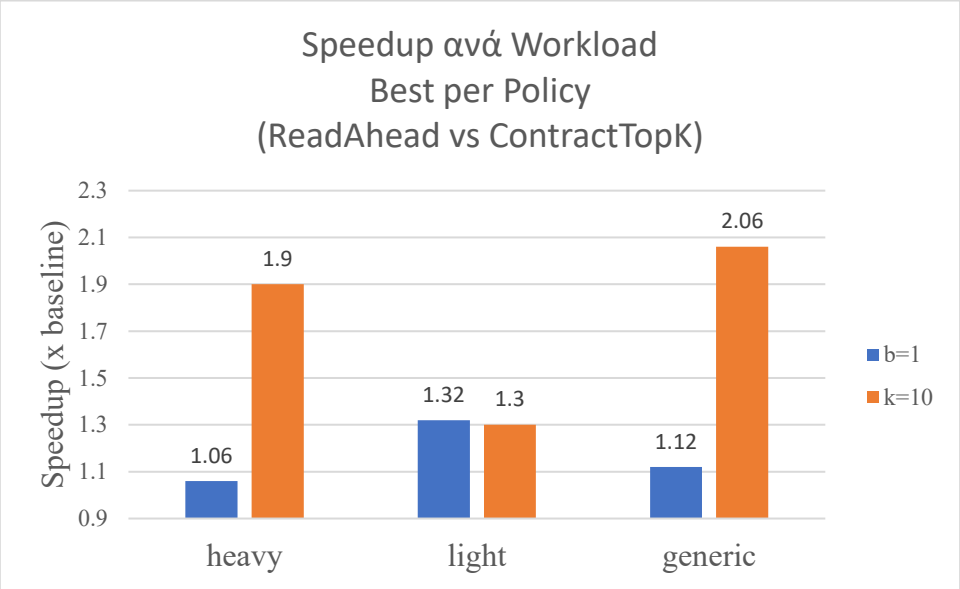

Light
ReadAhead: Hit Rate vs b
0.9
0.89
0.88
0.87
0.86
0.85
0.84
0.83
b=1
b=2
b=3







Εικόνα 16. Σύγκριση hitrate και speedup για την πολιτική ReadAhead στα τρία σενάρια για τις τιμές b=1,2,3. Παρατηρείται μικρή διακύμανση μεταξύ των τιμών με βέλτιστη τιμή για όλα τα σενάρια η b=1.



Εικόνα 17. Σύγκριση speedup για τη βέλτιστη παραμέτρο κάθε πολιτικής στα τρία σενάρια

Πίνακας 2. Συγκριτικός Πίνακας Αποτελεσμάτων Πολιτικών Prefetching

Πείραμα	**Πολιτική**	**Hit Rate**	**Coverage**	**Waste**	**Speedup**
Storage Heavy – Large Cache (512 KB)	ReadAhead (b=1)	0.56	0.45	0.55	1.06
Storage Heavy – Large Cache (512 KB)	ReadAhead (b=2)	0.55	0.43	0.57	1.04

Πείραμα	Πολιτική	Hit Rate	Coverage	Waste	Speedup
Storage Heavy – Large Cache (512 KB)	ReadAhead (b=3)	0.52	0.41	0.59	1.01
Storage Heavy – Large Cache (512 KB)	ContractTopK (k=5)	0.7	0.72	0.28	1.42
Storage Heavy – Large Cache (512 KB)	ContractTopK (k=7)	0.77	0.76	0.24	1.67
Storage Heavy – Large Cache (512 KB)	ContractTopK (k=10,15)	0.82	0.79	0.21	1.9
Storage Light – Small Cache (128 KB)	ReadAhead (b=1)	0.89	0.89	0.11	1.32
Storage Light – Small Cache (128 KB)	ReadAhead (b=2)	0.86	0.87	0.13	1.3
Storage Light – Small Cache (128 KB)	ReadAhead (b=3)	0.85	0.83	0.17	1.28
Storage Light – Small Cache (128 KB)	ContractTopK (k=5)	0.84	0.85	0.15	1.27
Storage Light – Small Cache (128 KB)	ContractTopK (k=7)	0.85	0.85	0.15	1.29
Storage Light – Small Cache (128 KB)	ContractTopK (k=10,15)	0.86	0.86	0.14	1.3
Generic – Large Cache (512 KB)	ReadAhead (b=1)	0.69	0.66	0.34	1.12
Generic – Large Cache (512 KB)	ReadAhead (b=2)	0.67	0.65	0.35	1.11
Generic – Large Cache (512 KB)	ReadAhead (b=3)	0.66	0.65	0.35	1.08
Generic – Large Cache (512 KB)	ContractTopK (k=5)	0.75	0.77	0.23	1.45
Generic – Large Cache (512 KB)	ContractTopK (k=7)	0.8	0.81	0.19	1.7
Generic – Large Cache (512 KB)	ContractTopK (k=10,15)	0.87	0.85	0.15	2.06

Ο παραπάνω πίνακας , σε συνδυασμό με τα γραφήματα, συνοψίζει τις μετρικές απόδοσης των δύο πολιτικών προφόρτωσης, ReadAhead και ContractTopK, σε τρια διαφορετικά σενάρια εργασίας (workloads) που διαφέρουν ως προς το είδος του workload. Οι δείκτες που παρουσιάζονται, (*Hit Rate*, *Coverage*, *Waste* και *Speedup*) επιτρέπουν μια συνολική αξιολόγηση της αποτελεσματικότητας κάθε πολιτικής, τόσο ως προς τη βελτίωση της αποδοτικότητας όσο και ως προς την αξιοποίηση των διαθέσιμων πόρων.

1. Συμπεριφορά στις Βαριές Εργασίες (Storage Heavy)

Στο Storage Heavy σενάριο (όπου έχουμε τα περισσότερα storage accesses), παρατηρούμε σαφή υπεροχή της ContractTopK πολιτικής, τόσο στη βελτίωση της επίδοσης (1.9x speedup για k=10 έναντι του ελάχιστου 1.06x speedup που πέτυχε η ReadAhead με την ιδανικότερη παράμετρο b=1), όσο και στην αξιοποίηση των διαθέσιμων πόρων, καθώς η ContractTopK σημειώνει coverage από 0.72 έως 0.79 σε αντίθεση με την ReadAhead για την οποία το Storage Heavy είναι το μοναδικό σενάριο στο οποίο σημειώνει «αρνητικό» coverage, δηλαδή χαμηλότερο coverage από waste, μεταξύ 0.41 και 0.45. Αυτή η συμπεριφορά μπορεί να δικαιολογηθεί από το γεγονός πως σε ένα workload με συνεχή storage accesses, η ReadAhead καταναλώνει τη διαθέσιμη cache με αρκετά slots τα οποία μπορεί να ζητηθούν ελάχιστες φορές, σε αντίθεση με την ContractTopK η οποία επιλέγει να αξιοποιήσει τη διαθέσιμη cache γεμίζοντάς την «έξυπνα» με τα πιο «διάσημα» slots εκμεταλλευόμενη τα «hot pools» που εμφανίζονται στο workload.

2. Ελαφριές Εργασίες (Storage Light)

Το Storage Light σενάριο (όπου έχουμε πολύ λιγότερα storage accesses, περί του 15% των συνολικών accesses), σχηματίζει μια τελείως διαφορετική εικόνα. Σε αυτήν την περίπτωση οι δύο πολιτικές παρουσιάζουν παρόμοια συμπεριφορά με speedup γύρω στο 1.3x (με νικήτρια πολιτική τη ReadAhead με speedup 1.32x για b=1 σε σχέση με το 1.3x για k=10 της ContractTopK) και coverage περί του 0.87 (με τη ReadAhead να αξιοποιεί καλύτερα τη διαθέσιμη cache σε σχέση με την ContractTopK). Αν και οι δύο πολιτικές φαίνονται να επιτυγχάνουν παρόμοια βελτίωση, θα πρέπει να επισημανθεί πως η ReadAhead είναι σαφώς προτιμότερη στο συγκεκριμένο workload, καθώς δεν απαιτεί το στάδιο της offline analysis και θα είχε την ίδια επιτυχία σε «άγνωστα» δεδομένα που θα παρουσίαζαν παρόμοια «δομή» (δηλαδή κυρίως account accesses με επαναλαμβανόμενα μοτίβα). Ένα ακόμα σημείο ενδιαφέροντος στο συγκεκριμένο σενάριο είναι πως στην περίπτωση της ContractTopK, αν

και η βέλτιστη παράμετρος φαίνεται να είναι η k=10, παρουσιάζει ελάχιστη βελτίωση σε σύγκριση με την k=5 η οποία χρησιμοποιεί αισθητά λιγότερη μνήμη.

3. Γενικό Workload (Generic)

Τέλος, έχουμε το Generic σενάριο στο οποίο έχουμε λίγο λιγότερα από τα μισά accesses να είναι storage accesses. Αν και η πολιτική ReadAhead επιτυχάνει μια αισθητή βελτίωση (1.12x speedup για b=1, με 0.66 coverage και επομένως χαμηλό αλλά μη αμελητέο waste), η ContractTopK πετυχαίνει το κορυφαίο speedup, 2.06x σε σχέση με το baseline για k=10, ενώ στην περίπτωση της «χειρότερης» παραμέτρου της, δηλαδή k=5, παρουσιάζει speedup 1.45x το οποίο εξακολουθεί να είναι πολύ καλύτερο από τη ReadAhead. Αυτά τα αποτελέσματα είναι λογικά αν αναλογιστούμε πως στο γενικό σενάριο έχουμε storage accesses τα οποία εμφανίζονται με μια μέση συχνότητα, αλλά σε επαναλαμβανόμενα addresses, κοινώς έχουμε ένα εξαιρετικό σενάριο για την ContractTopK που μπορεί να εκμεταλλευτεί τα πλεονεκτήματα της στατιστικής offline προεργασίας.

4. Συνολική Ερμηνεία και Συμπεράσματα

Η σύγκριση των πολιτικών δείχνει σαφώς ότι:

- Η ContractTopK παρουσιάζει εκθετικά οφέλη όσο αυξάνεται το k έως το επίπεδο κορεσμού (k≈10) και λειτουργεί καλύτερα σε σενάριο με περισσότερα, επαναλαμβανόμενα storage accesses.
- Η ReadAhead αποδίδει σταθερά αλλά περιορισμένα, λειτουργώντας καλύτερα σε μικρότερα workloads, χωρίς υπερβολικά storage accesses και κοιτάζοντας 1 block μπροστά (b=1).
- Ο γενικά βέλτιστος συνδυασμός είναι ContractTopK με k≈7–10

Τα αποτελέσματα αποδεικνύουν ότι η στατιστικά καθοδηγούμενη προφόρτωση (Top-K) προσφέρει σαφή πλεονεκτήματα έναντι των απλών heuristics (όπως το ReadAhead), ειδικά σε περιβάλλοντα με μεγάλες και συχνές προσπελάσεις. Η σταδιακή αύξηση του k βελτιώνει το hit rate χωρίς να αυξάνει το waste, ενώ το ReadAhead εξαντλεί άμεσα το δυναμικό του. Έτσι, η επιλογή της κατάλληλης παραμέτρου k μπορεί να αποτελέσει καθοριστικό παράγοντα για τη βελτιστοποίηση της συνολικής απόδοσης του συστήματος cache.

Πρέπει, βέβαια, να σημειωθεί πως η ContractTopK πολιτική συνοδεύεται από το overhead της περιοδικής offline analysis για να είναι αποτελεσματική, η οποία στα παραπάνω σενάρια δεν κατέστη προβληματική καθώς το κάθε σενάριο έτρεξε με το δικό του αρχείο topk_slots λόγω των διαφορετικών addresses που προέκυψαν κατά τη δημιουργία τους. Επομένως, σε ένα

πραγματικό σενάριο, μπορεί η πολιτική ReadAhead να ήταν προτιμότερη, ειδικά στην περίπτωση που παρουσίαζε παρόμοια συμπεριφορά με την ContractTopK (όπως στο light workload) αφού προσθέτει λιγότερη πολυπλοκότητα στο σύστημα και δεν απαιτεί «συντήρηση».

Κεφάλαιο 6 – Συμπεράσματα και Μελλοντική Εργασία

6.1 Σύνοψη ευρημάτων και συμβολή της έρευνας

Η παρούσα διπλωματική εργασία επιχείρησε να αντιμετωπίσει ένα βασικό και μέχρι σήμερα σχετικά υποεξερεύνητο πρόβλημα στο χώρο της βελτιστοποίησης blockchain συστημάτων: τη μείωση του I/O latency κατά την πρόσβαση στο state του Ethereum μέσω της εφαρμογής στοχευμένων πολιτικών προφόρτωσης δεδομένων. Μέσα από ένα συστηματικό πειραματικό πλαίσιο, ανέπτυξε και αξιολόγησε δύο διαφορετικές στρατηγικές προφόρτωσης, την πολιτική ReadAhead, που στηρίζεται στη χρονική προβλεψιμότητα, και την πολιτική Per-Contract Top-K, που αξιοποιεί στατιστικά μοτίβα πρόσβασης, αποδεικνύοντας ότι ακόμη και σχετικά απλές τεχνικές προφόρτωσης μπορούν να επιφέρουν βελτιώσεις στην απόδοση του execution layer. Τα ευρήματα της εργασίας υποδεικνύουν ότι η εφαρμογή τεχνικών προφόρτωσης στο blockchain περιβάλλον μπορεί να βελτιώσει τη βιωσιμότητα, την αποδοτικότητα και την ενεργειακή απόδοση του Ethereum δικτύου.

Από μεθοδολογική άποψη, η εργασία συνεισέφερε τον αρθρωτό και παραμετροποιήσιμο προσομοιωτή που αναπτύχθηκε στο πλαίσιό της. Με τη χρήση του προσομοιωτή ήταν δυνατή η αντικειμενική αξιολόγηση των πολιτικών προφόρτωσης και των διαφορετικών στρατηγικών. Σε πειραματικό επίπεδο, τα αποτελέσματα της εργασίας αποδεικνύουν ότι η πολιτική ReadAhead επιφέρει αισθητές βελτιώσεις όσον αφορά το hit rate και το speedup, με το speedup να παρουσιάζει βελτίωση έως και 1.32x και να είναι φανερά η καλύτερη επιλογή σε σχέση με την ContractTopK στην περίπτωση του light-storage workload όπου έχουμε ελάχιστα storage accesses. Οφείλουμε βέβαια να επισημάνουμε το storage heavy σενάριο όπου αν και με την πολιτική ReadAhead παρατηρείται βελτίωση, αυτή είναι ελάχιστη (1.06x speedup) και συνοδεύεται από αυξημένο waste (0.55). Αυτές οι τιμές υποδεικνύουν ότι σε ορισμένα σενάρια χωρίς πολύ βαριά workload, η πολιτική ReadAhead, αν και απλή στην λειτουργία της, είναι ικανή να επιφέρει σημαντικές βελτιώσεις. Παράλληλα, η πολιτική Per-Contract Top-K, παρουσιάζει σταθερά σημαντική βελτίωση σε workloads με αρκετά και επαναλαμβανόμενα storage accesses και αντιδρά αισθητά στις αλλαγές τις παραμέτρου k, όπως ήταν και αναμενόμενο, πετυχαίνοντας speedup από 1.27x έως και 2.06x, πάντα με αποδεκτά, χαμηλά επίπεδα waste. Οφείλουμε βέβαια να επισημάνουμε πως η πολιτική ContractTopK,

προκειμένου να παραμένει ανταγωνιστική και λειτουργική, εμπεριέχει το overhead της συχνούς offline analysis για να παραμένει ενημερωμένη για τα πιο διάσημα slots του κάθε address.

Σχετικά με τα latency-profiles εξάγεται το αναμενόμενο συμπέρασμα, πως οι πολιτικές προφόρτωσης είναι πιο αποτελεσματικές σε πιο αργές μνήμες με ακριβότερα κόστη miss, παραμένοντας όμως χρήσιμες και στην περίπτωση των NVMe-like προφίλ.

Όσον αφορά μελλοντικές έρευνες, μια προσοδοφόρα κατεύθυνση θα ήταν η ανάπτυξη adaptive πολιτικών που δυναμικά προσαρμόζουν τις παραμέτρους τους βάσει των χαρακτηριστικών των workloads ή της τρέχουσας κατάστασης του cache, καθώς και υβριδικών πολιτικών που χρησιμοποιούν κατάλληλη πολιτική για το τρέχον workload. Φυσικά, όπως είναι αντιληπτό από τα αποτελέσματα της τρέχουσας εργασίας, ακόμα και απλές πολιτικές τύπου ReadAhead μπορούν να επιφέρουν σημαντικές βελτιώσεις σε ένα περιβάλλον όπου ακόμα και ένα μικρό speedup συνεπάγεται σημαντική χρονική και ενεργειακή βελτίωση στο σύνολο του συστήματος λόγω του όγκου των δεδομένων και των συναλλαγών που πραγματοποιούνται, ειδικά αν αναλογιστούμε το πολιτικό και περιβαλλοντικό ζήτημα που έχει καταστεί το ενεργειακό αποτύπωμα του blockchain.

Είναι επίσης ξεκάθαρο, ότι δεν είναι αναγκαίο να αναμένουμε αλλαγές σε επίπεδο πρωτοκόλλου (όπως state expiry ή Verkle trees) για να δούμε βελτιώσεις στην απόδοση. Αντίθετα, συστηματικές βελτιώσεις στο λογισμικό του execution layer μπορούν να επιφέρουν άμεσα αποτελέσματα.

Τέλος, από πλευράς ολιστικής αξιολόγησης, η παρούσα εργασία έδειξε ότι το πρόβλημα της πρόσβασης στο blockchain state δεν μπορεί να αντιμετωπιστεί μόνο μέσω μιας μοναδικής τεχνικής ή στρατηγικής, αλλά απαιτεί ένα σύνολο συμπληρωματικών προσεγγίσεων. Μια πραγματική ενσωμάτωση θα πρέπει πιθανώς να συνδυάζει στοιχεία πολλών πολιτικών και να αναπτύσσει πιο έξυπνες, adaptive πολιτικές που αλλάζουν δυναμικά ανάλογα με τα χαρακτηριστικά του τρέχοντος workload.

6.2 Προτάσεις για ενσωμάτωση των πολιτικών σε πραγματικούς clients

Τα θετικά αποτελέσματα που προέκυψαν από τη μελέτη της παρούσας εργασίας ανοίγουν ένα σημαντικό ερώτημα: πώς μπορούν τα αποτελέσματα αυτά να μεταφρασθούν σε πραγματική ενσωμάτωση μέσα στους τρέχοντες Ethereum clients, όπως ο Geth, ο Nethermind ή ο Besu; Η απάντηση σε αυτό το ερώτημα δεν είναι απλή, καθώς απαιτεί συνδυασμό τεχνικής ικανότητας, αρχιτεκτονικής κατανόησης του κάθε client και σταδιακής ενσωμάτωσης με προσεκτική αξιολόγηση. Στη ενότητα αυτή, παρουσιάζονται προτάσεις για τη διαδικασία ενσωμάτωσης,

λαμβάνοντας υπόψη τόσο τις τεχνικές αναγκαιότητες όσο και τα πρακτικά εμπόδια που αναμένεται να συναντηθούν.

Το πρώτο και ίσως πιο σημαντικό βήμα προς την ενσωμάτωση είναι η κατανόηση της αρχιτεκτονικής του κάθε client και του τρόπου με τον οποίο το state data διαχειρίζεται σήμερα. Στον Geth, για παράδειγμα, η πρόσβαση στο state πραγματοποιείται μέσω του StateDB interface, το οποίο αποτελεί το κύριο σημείο επαφής μεταξύ του execution layer και της υποκείμενης βάσης δεδομένων (LevelDB ή RocksDB). Κάθε κλήση σε StateDB methods όπως GetBalance(), GetCode() ή GetState() περνά μέσω αυτού του interface, γεγονός που παρέχει το ιδανικό σημείο για την εισαγωγή τεχνικών prefetching. Ομοίως, στον Nethermind, το interface IWorldState διαδραματίζει ένα παρόμοιο ρόλο, επιτρέποντας την παρεμβολή λογικής προφόρτωσης σε σημείο κεντρικής σημασίας. Η κατανόηση αυτών των σημείων δεν είναι τυπική πληροφορία που βρίσκεται στη δημόσια τεκμηρίωση και απαιτεί εμβάθυνση στον πηγαίο κώδικα του κάθε client.

Η πρώτη και απλούστερη στρατηγική ενσωμάτωσης είναι η υλοποίηση ενός layer προφόρτωσης που λειτουργεί ως middleware μεταξύ του execution engine και του state database. Αυτό το middleware θα παρακολουθούσε κάθε πρόσβαση στο state, θα καταγράφει τα access patterns και θα προτείνει preemptive loads για τα επόμενα blocks. Ο σχεδιασμός αυτής της λύσης θα μπορούσε να αναπτυχθεί ως ξεχωριστό module, παράλληλο με τον υπάρχοντα κώδικα, γεγονός που θα ελαχιστοποιούσε τον κίνδυνο εισαγωγής bugs στις κρίσιμες διαδικασίες του execution layer. Το module αυτό θα εκτελούταν παράλληλα με το κύριο execution thread, ώστε να μη δημιουργούσε blocking συμπεριφορά. Με άλλα λόγια, η προφόρτωση δεδομένων θα ήταν μια background δραστηριότητα που δεν θα διακόπτε τη κανονική ροή εκτέλεσης, αλλά θα παρείχε οφέλη μόλις τα δεδομένα χρειάζονταν.

Για την υλοποίηση της πολιτικής ReadAhead, ο αρχιτεκτονικός σχεδιασμός θα ήταν σχετικά απλός. Το middleware θα παρακολουθούσε την τρέχουσα εκτέλεση του block B και θα διαβάζε εκ των προτέρων τα access traces από τα επόμενα N blocks (όπου N είναι η look-ahead distance). Ωστόσο, δεν θα μπορούσε απλώς να τραβήξει τα δεδομένα από τη βάση, θα έπρεπε να το κάνει με καθυστέρηση και με προσοχή ώστε να μη δημιουργήσει I/O congestion. Μια πιθανή υλοποίηση θα ήταν η χρήση ενός I/O scheduler που θα είχε την ευθύνη να ταξινομήσει και να ομαδοποιήσει τις προφορτώσεις, κάτι που θα μειώσει τη συνολική I/O latency σε σύγκριση με random accesses. Επιπλέον, θα επρέπε να υπάρχει ένας μηχανισμός ελέγχου του prefetch budget, δηλαδή ένα άνω όριο για τον αριθμό των objects που προφορτώνονται ανά block, ώστε να μη καταναλωθούν υπερβολικές ποσότητες μνήμης ή I/O bandwidth.

Για την υλοποίηση της πολιτικής Per-Contract Top-K, η διαδικασία θα ήταν ελαφρώς πιο πολύπλοκη, αλλά θα εξακολουθούσε να είναι εφικτή χωρίς τεράστιες αρχιτεκτονικές αλλαγές. Αρχικά, το σύστημα θα χρειάζονταν ένα offline analysis phase, κατά το οποίο θα ανέλυε τα ιστορικά δεδομένα εκτέλεσης και θα υπολόγιζε, για κάθε smart contract, ποια storage slots προσπελάζονται πιο συχνά. Αυτή η ανάλυση θα μπορούσε να εκτελεστεί περιοδικά (π.χ. κάθε εβδομάδα ή μήνα) και τα αποτελέσματα θα αποθηκεύονταν σε ένα cache-friendly format, όπως ένα dictionary ή ένα Bloom filter. Κατόπιν, κατά την εκτέλεση, όταν ο client εντοπίσει ότι θα εκτελεστεί ένα γνωστό contract, θα ανακτά τη λίστα των top-K slots και θα τα προφόρτωνε. Ο κύριος κίνδυνος αυτής της προσέγγισης είναι ότι τα ιστορικά δεδομένα ενδέχεται να μην εξακολουθούν να είναι ακριβή αν η συμπεριφορά του contract αλλάξει σημαντικά. Για να αντιμετωπιστεί αυτό, θα ήταν σκόπιμο να συμπεριληφθεί ένας μηχανισμός πιστοποίησης και αναπροσαρμογής των στατιστικών δεδομένων. Για παράδειγμα, το σύστημα θα μπορούσε να παρακολουθήσει το "hit rate" των προφορτωμένων slots και, αν παρατηρούσε σημαντική πτώση, θα μπορούσε να σημαιοθετήσει το contract για αναπροσαρμογή των στατιστικών του. Η ενσωμάτωση θα χρειάζονταν επίσης ένα configuration system που να επιτρέπει στο χειριστή του node να προσαρμόσει τις παραμέτρους της προφόρτωσης. Ιδανικά, αυτό θα μπορούσε να κάνει μέσω ενός YAML ή JSON configuration file, όπου θα μπορούσαν να καθοριστούν παράμετροι όπως το look-ahead distance για ReadAhead, το K για Top-K, το maximum prefetch budget ανά block, και η enabled/disabled κατάσταση για κάθε πολιτική. Αυτό θα επέτρεπε στους χρήστες του client να πειραματιστούν και να βρουν τις βέλτιστες ρυθμίσεις για τις δικές τους ανάγκες χωρίς να απαιτούνταν recompilation του κώδικα.

Ένα άλλο κρίσιμο στοιχείο της ενσωμάτωσης θα ήταν η παρακολούθηση (monitoring) και η ταυτοποίηση (tracing) των επιδόσεων του prefetching system σε πραγματικό χρόνο. Θα έπρεπε να προστεθούν metrics όπως το prefetch hit rate, το waste percentage, και ο συνολικός χρόνος που εξοικονομείται από την προφόρτωση. Αυτές οι μετρικές θα μπορούσαν να εξάγονται μέσω του prometheus metrics system που ήδη υποστηρίζουν οι περισσότεροι Ethereum clients, επιτρέποντας την παρακολούθηση της αποτελεσματικότητας του συστήματος και τη λήψη αποφάσεων βελτιστοποίησης με βάση real-time data.

Η διαδικασία της ενσωμάτωσης θα έπρεπε να ακολουθήσει μια σταδιακή προσέγγιση, ξεκινώντας με ένα prototype σε ένα ελεγχόμενο περιβάλλον. Αρχικά, θα μπορούσε να αναπτυχθεί ένα feature flag που θα επέτρεπε την ενεργοποίηση ή απενεργοποίηση της προφόρτωσης χωρίς αλλαγή του κώδικα. Αυτό θα επέτρεπε στις ομάδες δοκιμών να συγκρίνουν την απόδοση με και χωρίς την πολιτική ενεργή, ώστε να διασφαλιστεί ότι δεν υπάρχει καμία παρενέργεια. Κατόπιν, μετά από ικανοποιητικά αποτελέσματα σε testnet, θα

μπορούσε να γίνει ένα limited rollout σε mainnet, αρχικά σε μικρό ποσοστό κόμβων, με σκοπό τη συλλογή πραγματικών δεδομένων απόδοσης πριν την πλήρη εγκατάστασή της.

Ένα άλλο σημαντικό θέμα είναι η διαχείριση του κινδύνου. Η προφόρτωση δεδομένων, εάν δεν υλοποιηθεί σωστά, θα μπορούσε να καταστρέψει την ορθότητα της εκτέλεσης. Για να αποφευχθεί αυτό, η υλοποίηση θα έπρεπε να περιλαμβάνει σαφείς κανόνες συγχρονισμού (synchronization rules) και εκτενές testing. Ο έλεγχος ορθότητας θα έπρεπε να επικεντρωθεί στο να διασφαλίσει ότι η προφόρτωση δεδομένων δεν αλλάζει ποτέ τη σημασιολογία της εκτέλεσης, δηλαδή τα αποτελέσματα των συναλλαγών πρέπει να είναι πανομοιότυπα, είτε χρησιμοποιείται προφόρτωση είτε όχι.

Πέρα από τις τεχνικές προσπάθειες, η ενσωμάτωση θα χρειάζονταν και μια διεργασία δεδομένων και αναλύσεων για να συλλεχθούν μετρικές αποδοτικότητας από nodes που τρέχουν τη νέα κωδικοποίηση. Αυτά τα δεδομένα θα είναι κρίσιμα για να καθοριστεί αν η προφόρτωση έχει πράγματι τα αναμενόμενα αποτελέσματα στην πραγματική λειτουργία και αν έχουν ανακύψει άγνωστα προβλήματα. Θα ήταν σκόπιμο να δημιουργηθεί ένα "performance dashboard" όπου τα δεδομένα από πολλαπλά nodes θα συγκεντρώνονταν και θα αναλύονταν, παρέχοντας ένα ολιστικό view των επιδόσεων του συστήματος.

Όσον αφορά τη συμβατότητα με άλλους clients, είναι σημαντικό να σημειωθεί ότι οι πολιτικές προφόρτωσης που περιγράφονται δεν έχουν σχετικές επιπτώσεις. Με άλλα λόγια, δεν αναμένεται ότι θα αλλάξουν τα hash του state root ή θα επηρεάσουν τη συναίνεση του δικτύου. Αυτό σημαίνει ότι διαφορετικοί clients μπορούν να υλοποιήσουν διαφορετικές πολιτικές προφόρτωσης (ή καθόλου) χωρίς να σπάσουν τη συμβατότητα μεταξύ τους.

Η ενσωμάτωση των πολιτικών προφόρτωσης που έχουν αναπτυχθεί και αξιολογηθεί στη παρούσα εργασία σε πραγματικούς Ethereum clients είναι υποσχόμενη, αλλά απαιτεί προσεκτικό σχεδιασμό, ενσταλμένη προσέγγιση και σταδιακή δοκιμή. Η ενσωμάτωση θα παράγει οφέλη σε απόδοση και ενεργειακή αποδοτικότητα για τα nodes που εγκαθιστούν τα updates, ενώ δεν θα δημιουργήσει ασυμβατότητες με τα υπάρχοντα δίκτυα ή τους άλλους clients. Με κατάλληλη υλοποίηση και monitoring, αυτές οι τεχνικές δύνανται να γίνουν ένα standard feature στα blockchain systems, βελτιώνοντας τη συνολική απόδοση του Ethereum και του ευρύτερου blockchain οικοσυστήματος.

6.3 Μελλοντικές επεκτάσεις: adaptive budgets, Markov/n-gram predictors, online learning, hybridization

Αν και η παρούσα εργασία παρουσιάζει σημαντικές βελτιώσεις στην απόδοση του execution layer μέσω εφαρμογής σχετικά απλών πολιτικών προφόρτωσης, η έρευνα σε αυτό τον τομέα απέχει πολύ από το να έχει εξαντληθεί. Στη ενότητα αυτή θα αναλυθούν διάφορες κατευθύνσεις μελλοντικής ανάπτυξης που θα μπορούσαν να προωθήσουν τις πολιτικές προφόρτωσης σε υψηλότερα επίπεδα αποτελεσματικότητας. Οι κατευθύνσεις αυτές περιλαμβάνουν τη δυναμική προσαρμογή του prefetch budget, την εισαγωγή στατιστικών μοντέλων όπως τα Markov chains και τα n-gram models, την ενσωμάτωση online learning τεχνικών που προσαρμόζονται σε πραγματικό χρόνο, και τέλος την ανάπτυξη υβριδικών προσεγγίσεων που συνδυάζουν τα αποτελέσματα πολλαπλών τεχνικών για μεγιστοποίηση της αποδοτικότητας.

Μια πρώτη και πιο ευθεία επέκταση είναι η εισαγωγή ενός adaptive prefetch budget mechanism, δηλαδή ενός μηχανισμού που θα όριζε δυναμικά το πόσα prefetches μπορεί να πραγματοποιήσει η εκάστοτε πολιτική. Ένας adaptive budget allocator θα παρακολουθούσε τα χαρακτηριστικά του κάθε block (αριθμός συναλλαγών, αριθμό unique contracts, hit rates των προηγούμενων blocks) και θα προσαρμόζει δυναμικά τον available prefetch budget. Για παράδειγμα, αν ένα block έχει πολύ υψηλό hit rate, το σύστημα θα μπορούσε να μειώσει το budget για τα επόμενα blocks, ώστε να ελευθερώσει πόρους για άλλες δραστηριότητες. Αντίστροφα, αν ένα block αναμένεται να έχει πολλές τυχαίες προσπελάσεις, το σύστημα θα μπορούσε να αυξήσει το budget ώστε να καλυφθούν περισσότερες από τις πιθανές αναγνώσεις. Η υλοποίηση ενός τέτοιου adaptive budget allocator θα βασιζόταν σε συναρτήσεις που συνδυάζουν διάφορες μετρικές απόδοσης. Μια πιθανή προσέγγιση θα ήταν η χρήση ενός τύπου όπως:

$$\begin{aligned} BudgetNext \;=\; & BaselineBudget \\ & \times (\alpha \times HitRatePrev + \beta \times (1 - WastePrev) \\ & + \gamma \times ContractDiversity) \end{aligned}$$

όπου α, β, γ είναι σταθμοί που μπορούν να προσαρμοστούν εμπειρικά, HitRatePrev είναι ο hit rate του προηγούμενου block, WastePrev είναι το ποσοστό άχρηστων προφορτώσεων, και ContractDiversity είναι ένα μέτρο της ποικιλομορφίας των contracts που θα εκτελεστούν στο επόμενο block. Με αυτόν τον τρόπο, το σύστημα θα μπορούσε να προσαρμόζει αυτόματα και

σχετικά αποδοτικά τον διαθέσιμο budget, μεγιστοποιώντας τη χρήση πόρων ανάλογα με τις ανάγκες.

Η δεύτερη και ιδιαίτερα ενδιαφέρουσα επέκταση είναι η ενσωμάτωση προγνωστικών μοντέλων που βασίζονται σε στοχαστική ανάλυση, όπως τα Markov chains και τα n-gram models. Αυτές οι τεχνικές, οι οποίες έχουν χρησιμοποιηθεί εκτενώς σε άλλα πεδία (π.χ. natural language processing, network traffic analysis), θα μπορούσαν να εφαρμοστούν στο πλαίσιο του Ethereum state access prediction. Ένα Markov-based prefetcher θα λειτουργούσε ως εξής: θα κατασκευάζαν ένα γράφημα όπου κάθε κόμβος αντιπροσωπεύει ένα storage slot ή ένα account, και κάθε ακμή από κόμβο A σε κόμβο B θα έχει βάρος ίσο με την πιθανότητα ότι μετά την πρόσβαση στο A θα ακολουθήσει πρόσβαση στο B. Με βάση αυτές τις πιθανότητες μετάβασης, όταν το σύστημα προσπελάσει ένα συγκεκριμένο slot, θα μπορούσε να προφορτώσει τα πιο πιθανά επόμενα slots, βάσει της distribution που έχει μάθει από τα ιστορικά δεδομένα.

Τα n-gram models θα μπορούσαν να προσφέρουν ακόμη καλύτερες προβλέψεις, καθώς θα λάμβαναν υπόψη ακολουθίες ανάγνωσης, όχι μόνο μεμονωμένες μεταβάσεις. Για παράδειγμα, ένα 3-gram model θα κοίταζε τα τελευταία τρία accesses και θα προέβλεπε το τέταρτο. Αυτή η προσέγγιση θα μπορούσε να αποκαλύψει χρήσιμα patterns όπως "όταν A, B, C προσπελάζονται διαδοχικά, συνήθως ακολουθεί D". Η χρήση τέτοιων μοντέλων θα απαιτούσε την κατασκευή και αποθήκευση σημαντικά μεγαλύτερων δομών δεδομένων (transition matrices ή n-gram tables), αλλά το trade-off θα ήταν δυνητικά η αυξημένη ακρίβεια των προβλέψεων.

Η τρίτη και πιο δυναμική επέκταση είναι η εισαγωγή online learning τεχνικών. Σε αντίθεση με τα offline models που εκπαιδεύονται μια φορά και στη συνέχεια χρησιμοποιούνται στατικά, τα online learning models προσαρμόζονται συνεχώς καθώς ο client εκτελεί τα blocks. Σε έναν απλό online learning prefetcher για παράδειγμα, κάθε φορά που παρατηρείται μια πρόσβαση από A σε B, θα μπορούσε να ενημερωθεί η αντίστοιχη πιθανότητα ως:

$$P(B|A)new = \alpha \times P(B|A)old + (1 - \alpha) \times Observed,$$

όπου Observed είναι 1 αν παρατηρήθηκε η μετάβαση και 0 διαφορετικά, και α είναι ένας συντελεστής learning rate (π.χ., 0.7). Με αυτόν τον τρόπο, το σύστημα θα μπορούσε να προσαρμοστεί γρήγορα σε αλλαγές των access patterns, όπως όταν ένα δημοφιλές contract αρχίζει να χρησιμοποιείται λιγότερο ή όταν ένα νέο contract εμφανίζεται στο δίκτυο.

Οι τεχνικές bandit algorithms θα μπορούσαν επίσης να εφαρμοστούν σε αυτό το πλαίσιο. Τα bandit algorithms είναι ειδικά σχεδιασμένα για να αντιμετωπίζουν ένα trade-off μεταξύ της εξερεύνησης (exploration) νέων στρατηγικών και της εκμετάλλευσης (exploitation) των γνωστών καλών στρατηγικών. Ένας prefetcher που χρησιμοποιεί bandit algorithms θα μπορούσε να πειραματιστεί με διαφορετικά predicting strategies και σταδιακά να δώσει περισσότερο βάρος στα πιο αποτελεσματικά. Αυτό θα ήταν ιδιαίτερα χρήσιμο σε δυναμικά changing workloads, όπου καμία μία στρατηγική δεν είναι πάντα βέλτιστη.

Μια ακόμη πιο προηγμένη προσέγγιση θα ήταν η χρήση deep learning models, για την πρόβλεψη των επόμενων state accesses. Αυτά τα μοντέλα θα εκπαιδεύονταν σε τεράστιους όγκους blockchain traces και θα μπορούσαν να δημιουργήσουν πολύ ακριβείς προβλέψεις σχετικά με τα πιθανά επόμενα accesses. Θα μπορούσαν να μάθουν μακροπρόθεσμες εξαρτήσεις μεταξύ accesses, αναγνωρίζοντας πολύπλοκα patterns που θα ήταν δύσκολο να εντοπιστούν με κλασικές μεθόδους. Ωστόσο, η ενσωμάτωση τέτοιων μοντέλων στον Ethereum client θα αντιμετώπιζε σημαντικούς πρακτικούς περιορισμούς, δεδομένου ότι θα απαιτούσε σημαντική υπολογιστική ισχύ, κάτι που ενδέχεται να είναι απαγορευτικό για nodes με περιορισμένους πόρους.

Η υβριδικοποίηση (hybridization) των διαφορετικών πολιτικών αποτελεί μια άλλη εξαιρετικά ελπιδοφόρα κατεύθυνση. Αντί να βασιστούμε σε μία μόνο πολιτική, ένα hybrid system θα μπορούσε να συνδυάζει τα δυνατά σημεία πολλαπλών προσεγγίσεων. Για παράδειγμα, ένα σύστημα θα μπορούσε να εκτελέσει την ReadAhead και την Per-Contract Top-K παράλληλα, και κατόπιν να συνδυάσει τις προτάσεις τους χρησιμοποιώντας ένα ensemble voting mechanism. Εάν και οι δύο πολιτικές προτείνουν προφόρτωση του ίδιου slot, η κατάταξη (rank) του slot θα αυξάνεται, δίνοντας προτεραιότητα στις συμφωνίες. Εναλλακτικά, τα slots θα ταξινομούνταν σύμφωνα με ένα score ensemble, και μόνο τα κορυφαία slots εντός του budget θα προφορτώνονταν. Αυτή η προσέγγιση θα μπορούσε να συνδυάσει τη χρονική τοπικότητα που εξάγει η ReadAhead με τη στατιστική επαναληψιμότητα που ανακτά η Top-K, δημιουργώντας ένα σύστημα που είναι robust σε διαφορετικά είδη workloads.

Σε μια άλλη παραλλαγή, το σύστημα θα μπορούσε να χρησιμοποιήσει ένα meta-learner που να μαθαίνει, βάσει ιστορικών δεδομένων, ποια πολιτική είναι πιο αποτελεσματική για κάθε τύπο block ή workload. Αυτή η δυναμική επιλογή πολιτικής θα μπορούσε να μεγιστοποιήσει την απόδοση και την αποφυγή του overhead που θα συνεπάγονταν η παράλληλη εκτέλεση πολλαπλών πολιτικών.

Μια άλλη ενδιαφέρουσα κατεύθυνση είναι η ενσωμάτωση του concept του "collaborative prefetching" σε ένα blockchain δίκτυο. Στις τρέχουσες μελέτες, κάθε κόμβος εκτελεί τις

πολιτικές προφόρτωσης ανεξάρτητα. Ωστόσο, αν οι κόμβοι μπορούσαν να ανταλλάσσουν πληροφορίες σχετικά με τα access patterns που παρατηρούν, θα μπορούσαν αμοιβαία να ενημερώσουν τα μοντέλα τους και να βελτιώσουν τις προβλέψεις. Ένας απλός τρόπος για να γίνει αυτό θα ήταν η ανταλλαγή συνοπτικών στατιστικών (π.χ., τα top-K slots ανά contract) μέσω ενός gossip protocol, χωρίς να εκτίθενται ευαίσθητα δεδομένα. Αυτό θα μπορούσε να δημιουργήσει ένα δίκτυο-πλατεία learning system, όπου όλοι οι κόμβοι συμβάλλουν σε ένα κοινό μοντέλο και αντλούν οφέλη από τις παρατηρήσεις άλλων.

Τέλος, η αξιολόγηση μεγαλύτερης κλίμακας και η ποσοτικοποίηση των environmental impacts της προφόρτωσης θα αποτελούσε μια σημαντική κατεύθυνση μελλοντικής έρευνας. Θα ήταν ενδιαφέρον να μετρηθεί όχι μόνο το ποσοστό βελτίωσης της απόδοσης, αλλά και η μείωση της κατανάλωσης ηλεκτρικής ενέργειας και του ενεργειακού αποτυπώματος του Ethereum δικτύου στο σύνολό του.

Το πεδίο της προφόρτωσης δεδομένων στο blockchain περιβάλλον προσφέρει ένα εξαιρετικά πλούσιο τοπίο ερευνητικών ευκαιριών. Από τη δυναμική προσαρμογή του budget μέχρι τη χρήση advanced machine learning models, μέχρι τη δημιουργία συνεργατικών δικτύων learning, υπάρχουν πολλές αξιόλογες κατευθύνσεις που θα μπορούσαν να οδηγήσουν σε ακόμη πιο αποδοτικά blockchain systems. Η παρούσα εργασία παρέχει μια σταθερή βάση και ένα πλαίσιο αναφοράς για όλες αυτές τις μελλοντικές προσπάθειες, ενώ ταυτόχρονα αποδεικνύει ότι ακόμη και με σχετικά απλές τεχνικές, σημαντικές βελτιώσεις είναι δυνατόν να επιτευχθούν. Με τη σταδιακή ενσωμάτωση αυτών των προηγμένων τεχνικών, η ελπίδα είναι ότι τα blockchain systems θα γίνουν όλο και πιο αποδοτικά, ενεργειακά φιλικά και χωρητικά, επιτρέποντας στο Ethereum και σε άλλα blockchain να κλιμακωθούν ώστε να εξυπηρετήσουν τις ανάγκες ενός παγκόσμιου κοινού.

Λίστα Βιβλιογραφικών Αναφορών

Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps.* O'Reilly Media.

Baek, S. H., & Park, K. H. (2009). Striping-Aware Sequential Prefetching for Independency and Parallelism in Disk Arrays with Concurrent Accesses. *IEEE Transactions on Computers*, *58*(8), 1146–1152. https://doi.org/10.1109/tc.2009.14

Bai, J., Zhu, S., & Ji, H. (2024). Blockchain based decentralized and proactive caching strategy in mobile edge computing environment. *Sensors*, *24*(7), 2279. https://doi.org/10.3390/s24072279

Buterin, V., O'Riordan, D., & Wuille, P. (2021). *Ethereum 2.0: Scaling the World Computer.* Ethereum Foundation Research Papers. https://ethereum.org/en/developers/docs/

Cilku, B., & Puschner, P. P. (2014, December). Designing a time-predictable memory hierarchy for single-path code. *Proceedings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (COMPONENTS 2014)*, Rome, Italy. https://doi.org/10.13140/2.1.5111.5202

Denning, P. (1980). Working sets past and present. *IEEE Transactions on Software Engineering*, *SE-6*(1), 64–84. https://doi.org/10.1109/tse.1980.230464

Ergen, M., Saoud, B., Shayea, I., El-Saleh, A. A., Ergen, O., Inan, F., & Tuysuz, M. F. (2024). Edge computing in future wireless networks: A comprehensive evaluation and vision for 6G and beyond. *ICT Express*, *10*(5), 1151–1173. https://doi.org/10.1016/j.icte.2024.08.007

Ethereum Foundation. (2023). *Ethereum client architecture and state management.* https://ethereum.org/en/developers/docs/nodes-and-clients/

Hasan, M. A., Lazem, A., & Nahi, H. A. (2020). Comparative study for cache prefetching algorithms. *International Journal of Psychosocial Rehabilitation, 24*(5), 3865–3878.

Hias, R., Wang, W., Vanhoof, J., & Van Cutsem, T. (2024). *A scalable state sharing protocol for low-resource validator nodes in blockchain networks* [Preprint]. *arXiv*. https://arxiv.org/abs/2410.05854

Jung, S., Lee, H., & Jo, H. (2023). CluMP: Clustered Markov Chain for Storage I/O Prefetch. *Electronics*, *12*(15), 3293. https://doi.org/10.3390/electronics12153293

Kurisaka, H., Su, Y., Nguyen, P. L., Nguyen, K., & Sekiya, H. (2025). Performance evaluation of ethereum consensus mechanisms in IoT-blockchain systems using resource-constrained devices. *Cluster Computing*, *28*(12). https://doi.org/10.1007/s10586-025-05503-w

Li, X., Zhai, Z., & Ye, X. (2021). AOIO: Application Oriented I/O Optimization for Buffer Management. *Symmetry*, *13*(4), 573. https://doi.org/10.3390/sym13040573

Love, R. (2010). *Linux Kernel Development* (3rd ed.). Addison-Wesley.

Megiddo, N., & Modha, D. S. (2003). *ARC: A self-tuning, low overhead replacement cache. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 115–130.

Nguyen, Lam. (2022). Blockchain for Internet of Things Data Markets, Learning, and Sustainability. 10.13140/RG.2.2.21847.55202

NodeReal. (2023, February 27). *Geth Path-Based storage model and newly inline state prune*. NodeReal Blog. https://nodereal.io/blog/en/geth-path-based-storage-model-and-newly-inline-state-prune/

Ruemmler, C., & Wilkes, J. (1994). An introduction to disk drive modeling. *Computer*, *27*(3), 17–28. https://doi.org/10.1109/2.268881

Understanding Merkle Patricia Tries in Ethereum | Guides | GoldRush. (n.d.). GoldRush. https://goldrush.dev/guides/understanding-merkle-patricia-trees-in-ethereum/

Wood, G. (2014). *Ethereum: A secure decentralized generalized transaction ledger (Yellow Paper)*. https://ethereum.github.io/yellowpaper/paper.pdf

Xu, D., Gao, Y., & Xiao, X. (2022). Precision Poverty Alleviation Methods in the Agricultural Field Based upon Wireless Communication Networks and Blockchain. *Wireless Communications and Mobile Computing*, *2022*, 1–12. https://doi.org/10.1155/2022/2687445

Yang, H., Fang, J., Hou, Y., Su, X., & Xiong, N. N. (2025). Reinforcement Learning-Driven adaptive prefetch aggressiveness control for enhanced performance in parallel system architectures. *IEEE Transactions on Parallel and Distributed Systems*, 1–18. https://doi.org/10.1109/tpds.2025.3550531

Zhang, M., Tang, Q., Kim, J., Burgstaller, B., & Kim, S. (2023). Adaptive Regression Prefetching Algorithm by using big data application characteristics. *Applied Sciences*, *13*(7), 4436. https://doi.org/10.3390/app13074436

Zheng, S., Mei, H., Huang, L., Shen, Y., & Zhu, Y. (2017). Adaptive Prefetching for Accelerating Read and Write in NVM-Based File Systems. *2017 IEEE 35th International Conference on Computer Design*, 49–56. https://doi.org/10.1109/iccd.2017.17