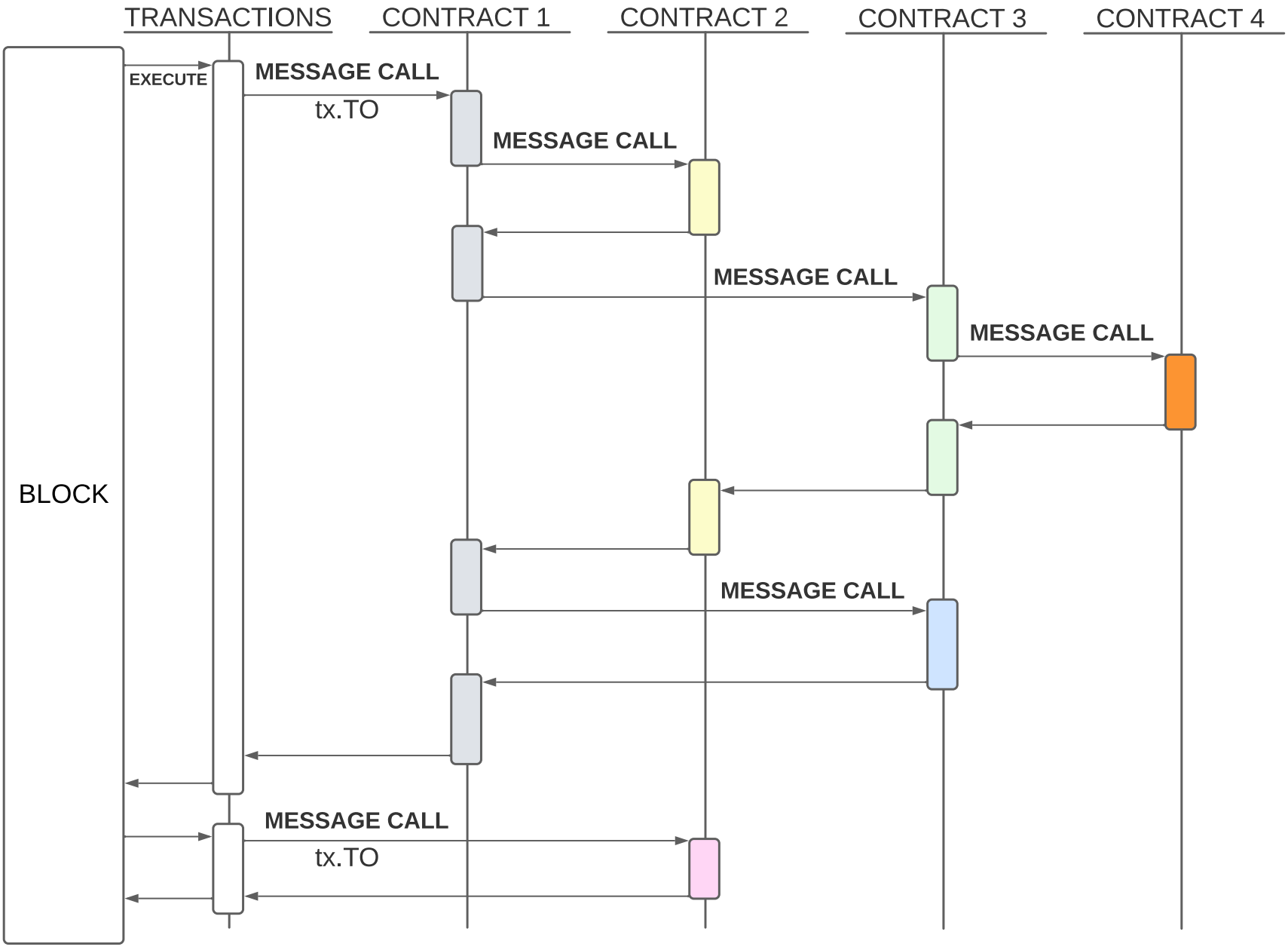

# Επιτάχυνση επιπέδου αποθήκευσης των Blockchain Clients με στατική ανάλυση και υποθετική εκτέλεση των Smart Contracts | prefetch?

Διπλωματική Εργσία ....

# Ethereum
# Λογαριασμοί και Συμβόλαια

- Λογαριασμοί: Εξωτερικοί (EOA) ή Συμβόλαια (SC)
- Transaction: μεταφορά Ether ή/και μηνύματος από EOA σε άλλο EOA ή SC, ή δημιουργία νέου SC
- Προς SC: είναι κλήση-μήνυμα (message call) και εκτελεί κώδικα του SC
- Message calls γίνονται και από SC σε SC, στη διάρκεια του ίδιου transaction

# Message calls



# EVM

- TODO: σύντομη περιγραφή

# World State

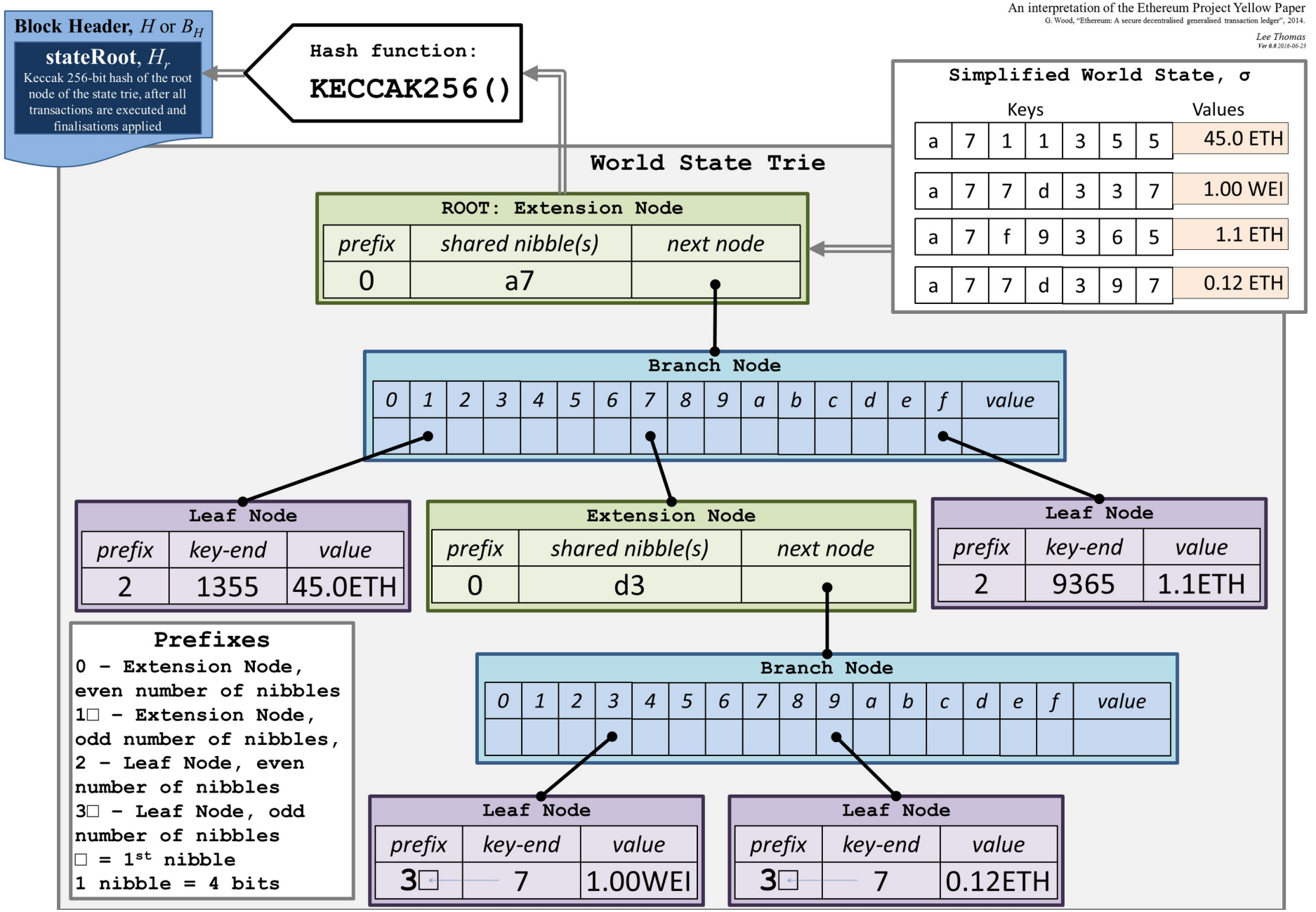
- Κάθε λογαριασμός είναι τετράδα
  (nonce, balance, code hash, storage root)
  Προσδιορίζεται από τη διεύθυνσή του (160bit)
  Διεύθυνση=Key → Value=Λογαριασμός
- Κάθε SC έχει χώρο μόνιμης αποθήκευσης μορφής slot=Key*→ Value=περιεχόμενο (256bit και τα δύο)
- Αποθήκευσή τους σε modified Merkle Patricia Trie
- Το σύνολο όλων είναι το World State, ίδιο σε όλους τους clients, τροποποιήται από τα Transactions

* για προστασία DOS, περνά πρώτα από hash

# Merkle Patricia Trie

- Λειτουργεί ως KV store
- Δενδρική δομή, το κλειδί είναι το μονοπάτι από τη ρίζα στον κόμβο με την τιμή για το κλειδί αυτό
- Κόμβοι branch βάζουν 1 δεκαδικό ψηφίο στο κλειδί
- Κόμβοι επέκτασης βάζουν πολλά ψηφία
- Κόμβοι φύλλα κρατάνε την αντίστοιχη τιμή
  Και τα branch μπορούν να έχουν τιμή
- Οι ακμές αποτυπώνονται με βάση το hash των κόμβων, όχι απλά δείκτες-διευθύνσεις μνήμης
- → Hash της ρίζας εξαρτάται από hash όλων

# Merkle Patricia Trie




Lee Thomas, https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture

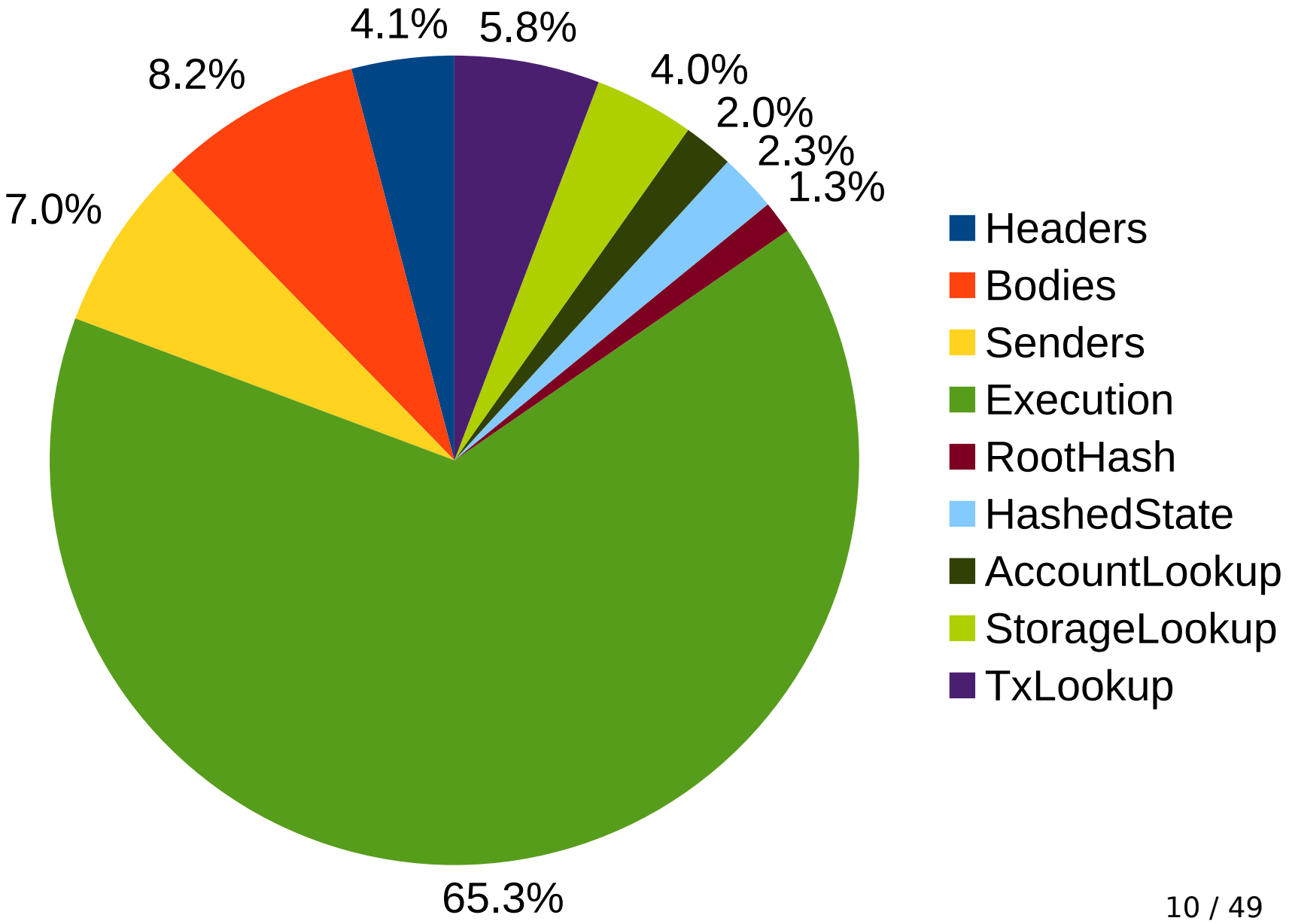
# KV-store in KV-store

- Το Trie αποθηκεύεται σε βάση KV-store στο δίσκο (LevelDB, RocksDB)
- Hash κόμβου = Key → Value = Περιεχόμενο κόμβου
- Κάθε hash 2 φορές: μία μέσα σε value, μία ως key
- Το Trie το ίδιο είναι δομή KV-store
- Τα Account αποθηκεύονται μέσα στο Trie το οποίο αποθηκεύεται μέσα στη database
- Έχει και πλεονεκτήματα: πχ ιστορική αναδρομή είναι trivial

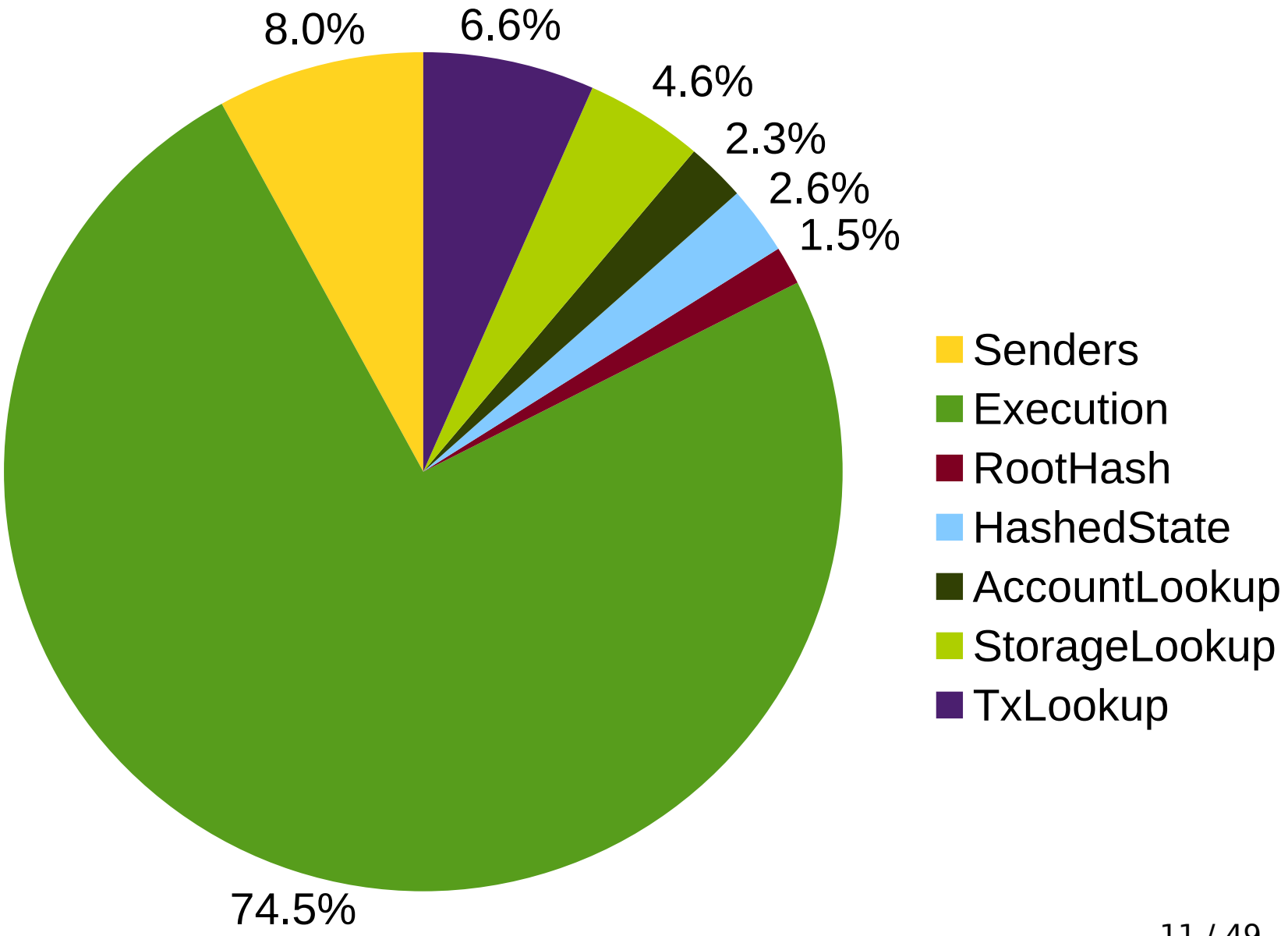
# Erigon

- Fork του official client (go-ethereum)
- Η δουλειά χωρίζεται σε stages, πχ download headers, download blocks, execute transactions, ...
- Δεν χρησιμοποιεί Trie, αντιθέτως αποθηκεύει τα Accounts κατ’ευθείαν στη database, “Plain State”
- Επιπλέον στάδιο: υπολογισμό των hashes (τα block θέλουν το root hash του Trie)
- Database: MDBX (fork της LMDB), mmap
- Σημαντική βελτίωση σε χρόνο και χώρο
- SOTA, από θέμα ταχύτητας απ’ όσο γνωρίζω

# Stages


4.1%
5.8%
8.2%
4.0%
2.0%
2.3%
1.3%
7.0%
65.3%
Headers
Bodies
Senders
Execution
RootHash
HashedState
AccountLookup
StorageLookup
TxLookup

# Stages (χωρίς δικτύου)



# Execution Stage

- Γίνεται η εκτέλεση των block (πχ αμοιβή miner), transaction και SC
- Σειριακή εκτέλεση, άγνωστα dependencies μεταξύ transaction (άγνωστες σχέσεις μεταξύ των SC)
- CPU και IO heavy
- Συνεχής πρόσβαση στο World State (DB), χρονοβόρα blocking reads
- Αν γινόταν prefetching, οι σελίδες της βάσης θα ήταν στη μνήμη (FS cache) και τα reads γρήγορα
- → στόχος της εργασίας

# Πόσο I/O;

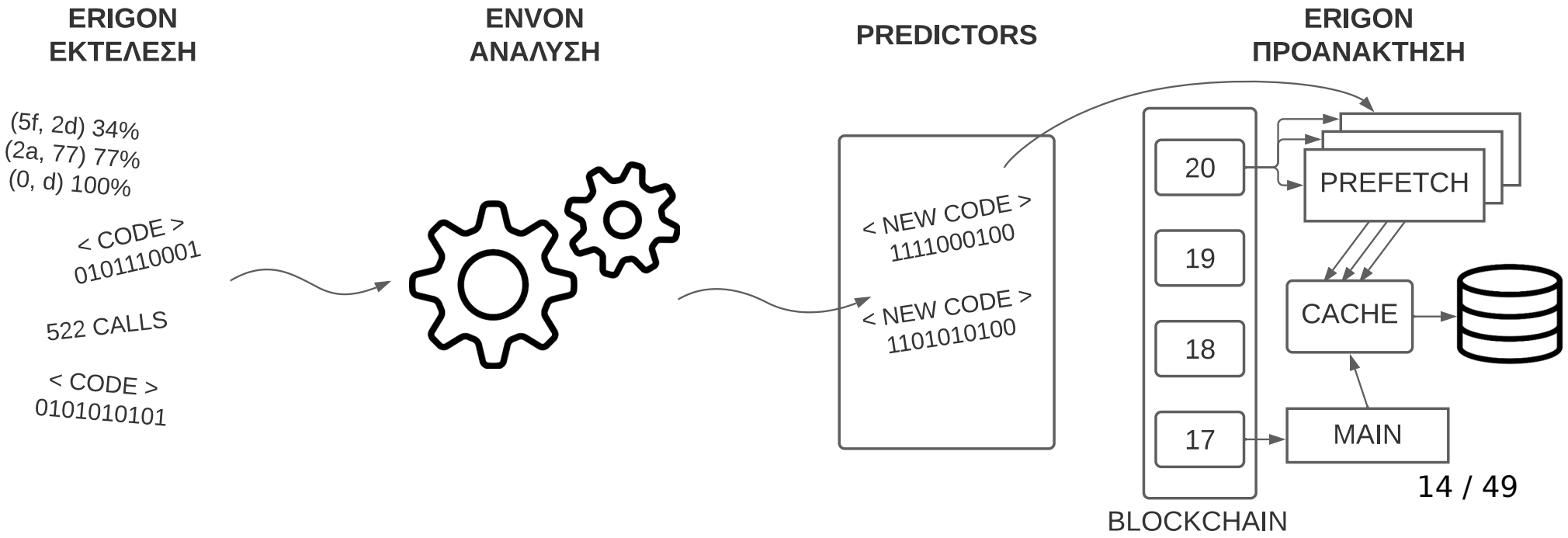
- Δοκιμαστική εκτέλεση 30K block,
  μέτρηση πόσου χρόνου περιμένει στη βάση
- Με γρήγορο δίσκο NVME:
  20 δευτ. από τα  89 συνολικά ( 22 % )
- Με πιο αργό δίσκο sata SSD:
  52 δευτ. από τα 150 συνολικά ( 35 % )

- Υπάρχει η δυνατότητα για σημαντική βελτίωση

(με σκληρό δίσκο δεν έγιναν δοκιμές, η ταχύτητα εκτέλεσης
είναι 1-2 τάξεις μεγέθους χαμηλότερη)

# Δομή του συστήματος

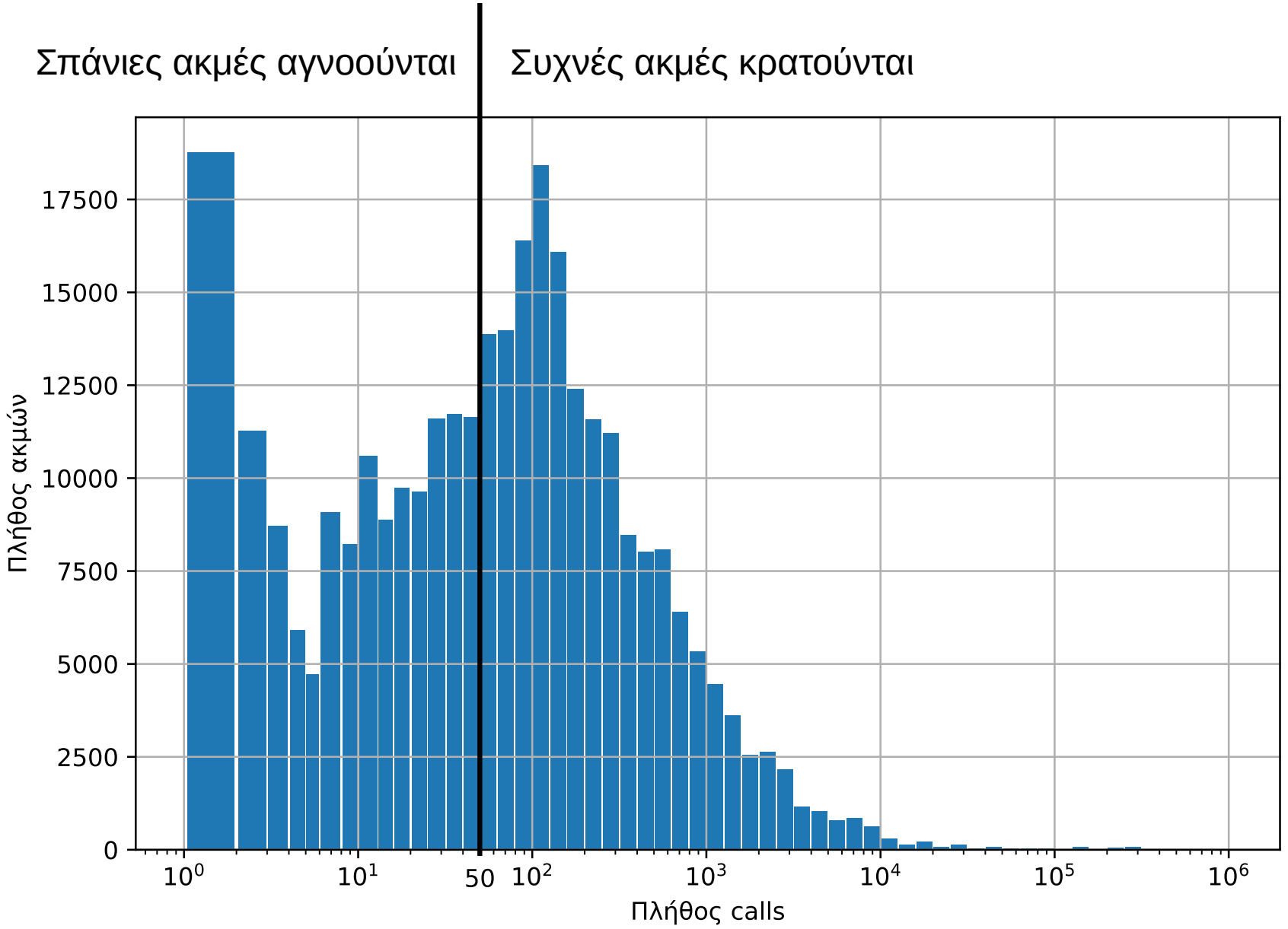
- Tracing: Συλλογή μετρικών και κώδικα των SC
- Analysis: Ανάλυση των SC που συλλέχθηκαν και σύνθεση νέων προγραμμάτων (predictors)
- Prefetching: Προανάκτηση και υποθετική εκτέλεση των predictors



# Tracing

- Κατά την πραγματική εκτέλεση, συλλέγουμε τον κώδικα των SC και πλήθος κλήσεων στο καθένα
- Τα δημοφιλή αποθηκεύονται για ανάλυση
- Επίσης συλλέγονται για κάθε JUMP, ο PC πριν και μετά και unique ID του message call
- Τα ζεύγη (ακμές CFG) με “πολλές” κλήσεις δίνονται σαν υποβοήθηση στον αναλυτή

# Tracing


Σπάνιες ακμές αγνοούνται
Συχνές ακμές κρατούνται
Πλήθος ακμών
17500
15000
12500
10000
7500
5000
2500
0
$10^0$
$10^1$
50
$10^2$
$10^3$
$10^4$
$10^5$
$10^6$
Πλήθος calls

# Ανάλυση

# Ανάλυση

- Ξεχωριστή διεργασία, σε Python
- Έχει ως είσοδο contract code, παράγει ως έξοδο προγράμματα (predictors) με παρόμοιο κώδικα
- Οι predictors περιέχουν μόνο “χρήσιμες” εντολές, για το prefetching: πρόσβασης storage, calls και return data
- Η συμπεριφορά τους επιτρέπεται να διαφέρει από αυτή του αρχικού κώδικα (ευκαιρίες για unsafe optimization, πχ pruning σπάνιων εντολών)

# Ανάλυση 1/12

- Παίρνει ως είσοδο τον κώδικα ενός SC, που είναι σε μορφή EVM bytecode

```
6080604052600436106100565763ffffffff7c010000000000000000000000000000000000000000000000000000000060003504166636ae17ab7811461005b578063771c0ad91461008d578063e80db5db146100ab575b600080fd5b348015610067576000080f...
```

# Ανάλυση 2/12

- Κάνει disassemble

```
00000000: PUSH1 0x80
00000002: PUSH1 0x40
00000004: MSTORE
00000005: PUSH1 0x4
00000007: CALLDATASIZE
00000008: LT
00000009: PUSH2 0x56
0000000c: JUMPI
0000000d: PUSH4 0xffffffff
00000012: PUSH29 0x1000000000000...
00000030: PUSH1 0x0
00000032: CALLDATALOAD
00000033: DIV
00000034: AND
....
```

# Ανάλυση 3/12

- Το χωρίζει σε basic blocks

```
-----   BLOCK ~0 -----
00000000: PUSH1 0x80
00000002: PUSH1 0x40
00000004: MSTORE
00000005: PUSH1 0x4
00000007: CALLDATASIZE
00000008: LT
00000009: PUSH2 0x56
0000000c: JUMPI
```

```
-----   BLOCK ~d -----
0000000d: PUSH4 0xffffffff
00000012: PUSH29 0x100000000...
00000030: PUSH1 0x0
00000032: CALLDATALOAD
00000033: DIV
00000034: AND
00000035: PUSH4 0x6ae17ab7
0000003a: DUP2
0000003b: EQ
0000003c: PUSH2 0x5b
0000003f: JUMPI
```

```
-----   BLOCK ~40 -----
00000040: DUP1
00000041: PUSH4 0x771c0ad9
00000046: EQ
00000047: PUSH2 0x8d
0000004a: JUMPI
```

# Ανάλυση 4/12

- Μετατρέπει τις εντολές σε μορφή SSA

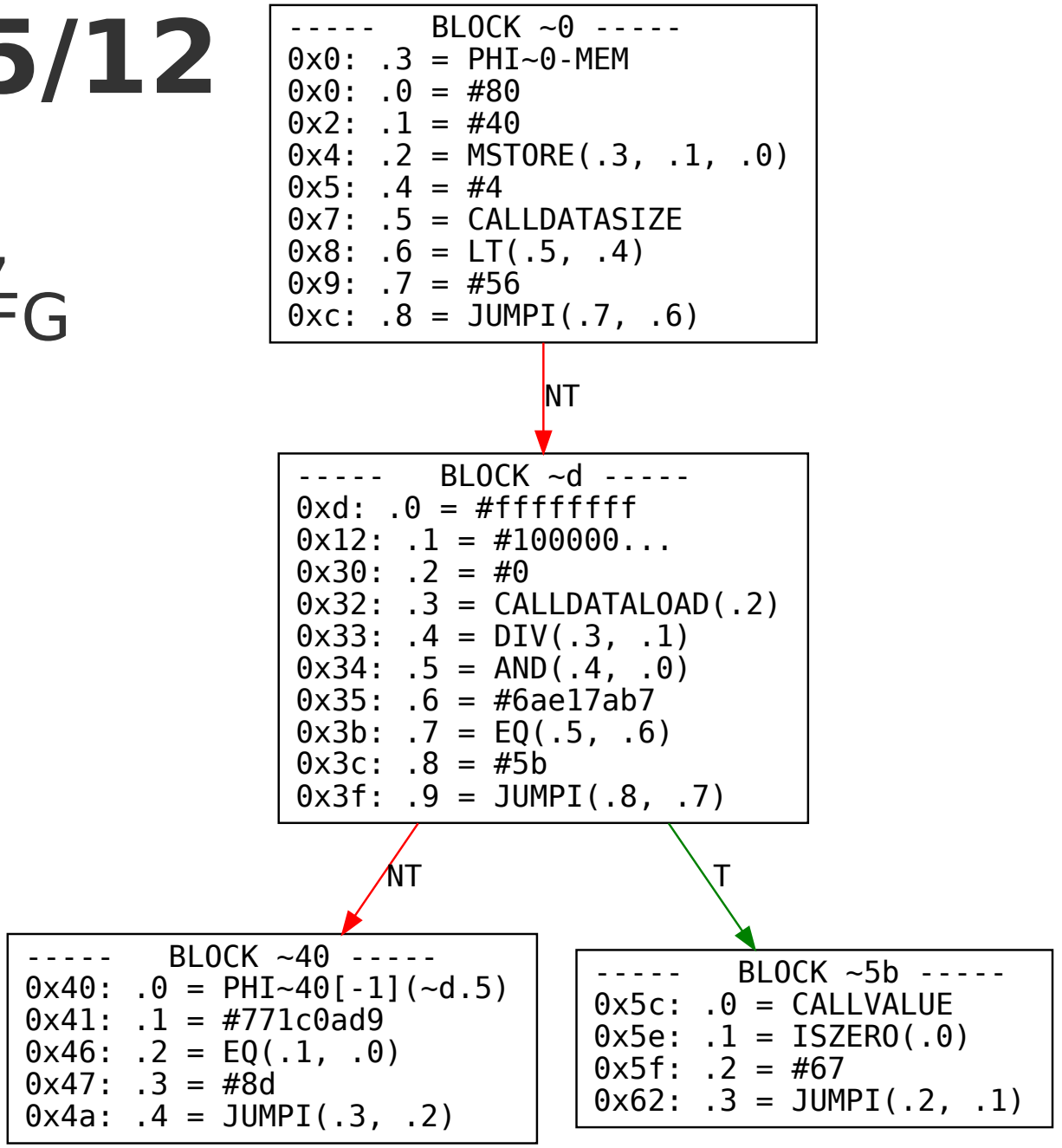
```
-----   BLOCK ~0 -----
0x0: .3 = PHI~0-MEM
0x0: .0 = #80
0x2: .1 = #40
0x4: .2 = MSTORE(.3, .1, .0)
0x5: .4 = #4
0x7: .5 = CALLDATASIZE
0x8: .6 = LT(.5, .4)
0x9: .7 = #56
0xc: .8 = JUMPI(.7, .6)
```

```
-----   BLOCK ~d -----
0xd: .0 = #ffffffff
0x12: .1 = #10000...
0x30: .2 = #0
0x32: .3 = CALLDATALOAD(.2)
0x33: .4 = DIV(.3, .1)
0x34: .5 = AND(.4, .0)
0x35: .6 = #6ae17ab7
0x3b: .7 = EQ(.5, .6)
0x3c: .8 = #5b
0x3f: .9 = JUMPI(.8, .7)
```

```
-----   BLOCK ~40 -----
0x40: .0 = PHI~40[-1]
0x41: .1 = #771c0ad9
0x46: .2 = EQ(.1, .0)
0x47: .3 = #8d
0x4a: .4 = JUMPI(.3, .2)
```

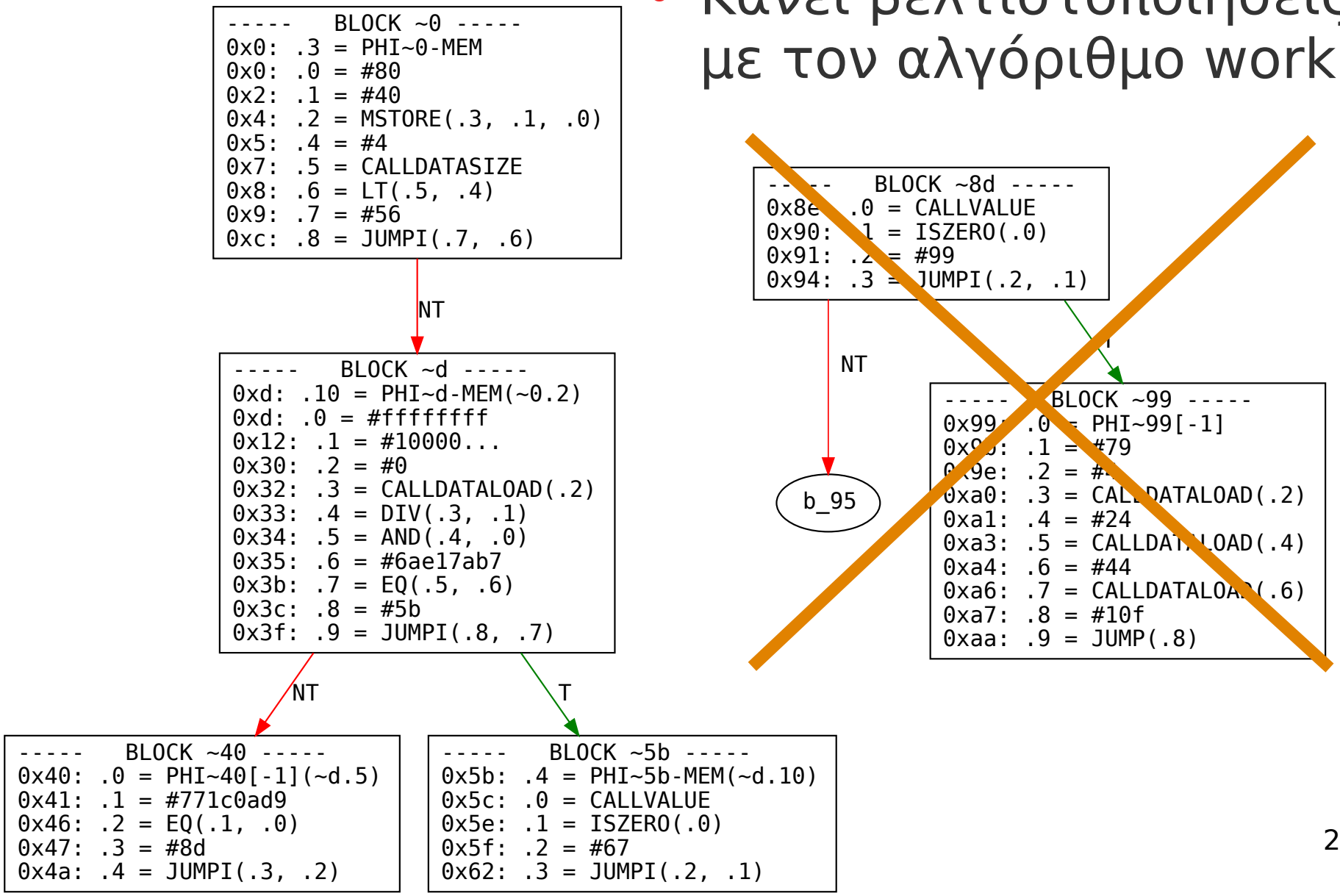
# Ανάλυση 5/12

- Σύνδεση των block, αρχική εκτίμηση CFG



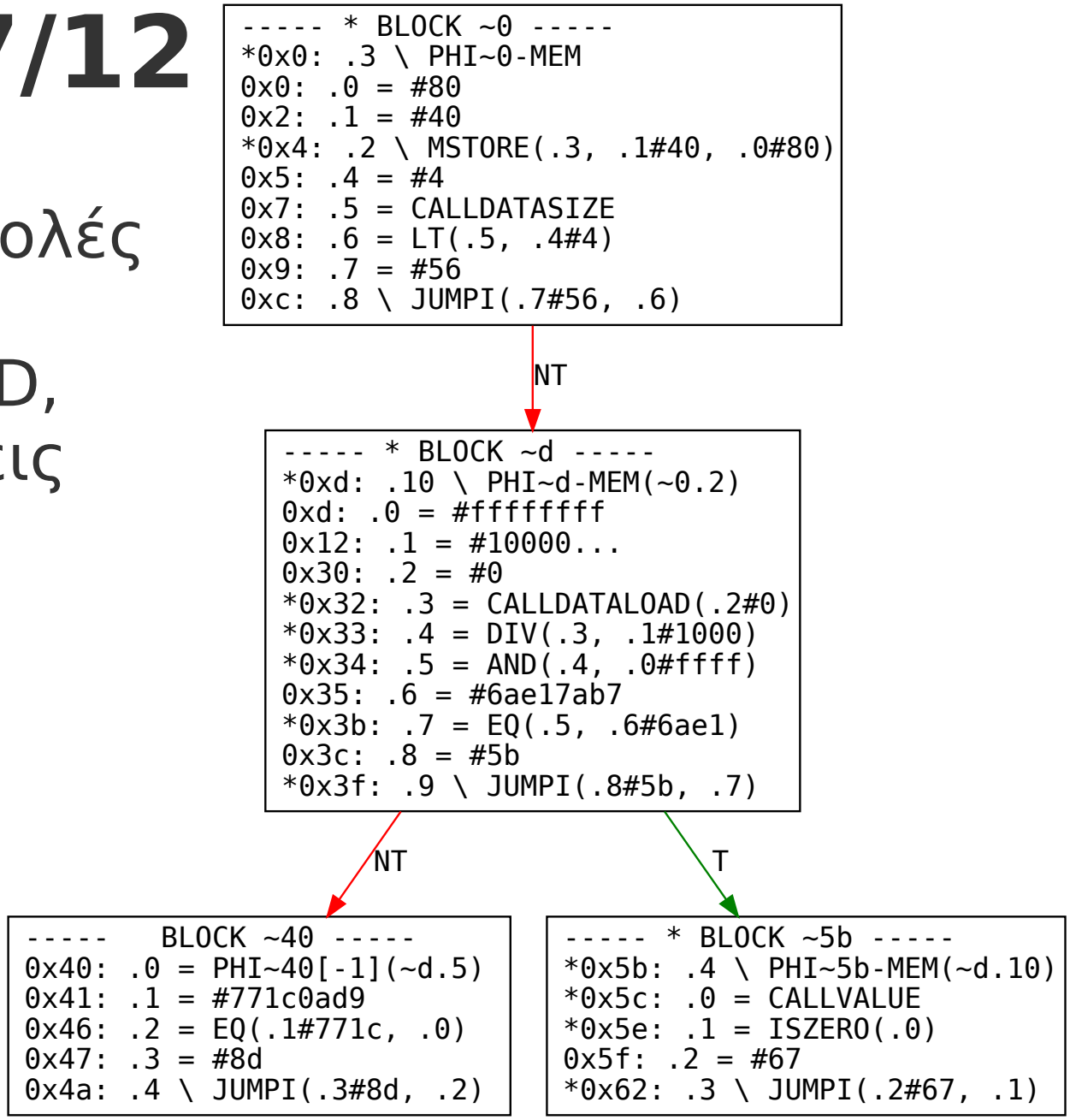
# Ανάλυση 6/12

- Κάνει βελτιστοποιήσεις, με τον αλγόριθμο worklist



# Ανάλυση 7/12

- Επιλέγονται οι εντολές που θα μπουν στον predictor, πχ SLOAD, CALL και εξαρτήσεις



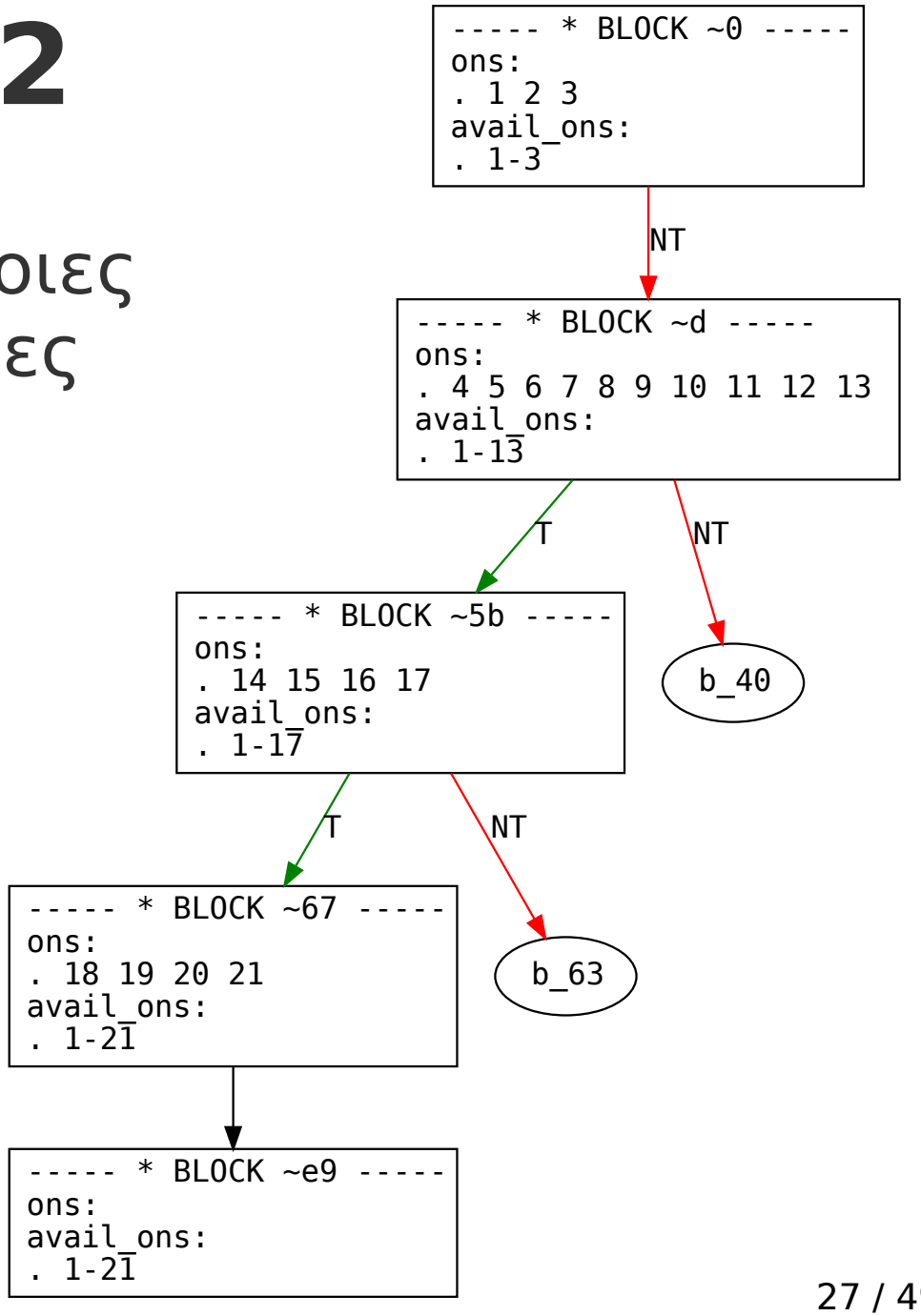
# Ανάλυση 8/12

- Απαριθμεί τις τιμές των επιλεγμένων εντολών

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

# Ανάλυση 9/12

- Βρίσκει σε κάθε block ποιες είναι διαθέσιμες και ποιες πρέπει να υπολογιστούν



# Ανάλυση 10/12

- Υπολογίζει τις νέες εκφράσεις τους

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

# Ανάλυση 11/12

- Συνθέτει τον κώδικα του predictor

```
~0 | ENTRY
     1 = #40
     2 = #80
     3 = MSTORE 0 1 2
~d | ~0
     4 = #10000...
     5 = #0
     6 = #6ae17ab7
     7 = #5b
     8 = #ffffffff
     9 = CALLDATALOAD 5
    10 = DIV 9 4
    11 = AND 10 8
    12 = EQ 11 6
    13 = JUMPI 7 12
....
```

# Ανάλυση 12/12

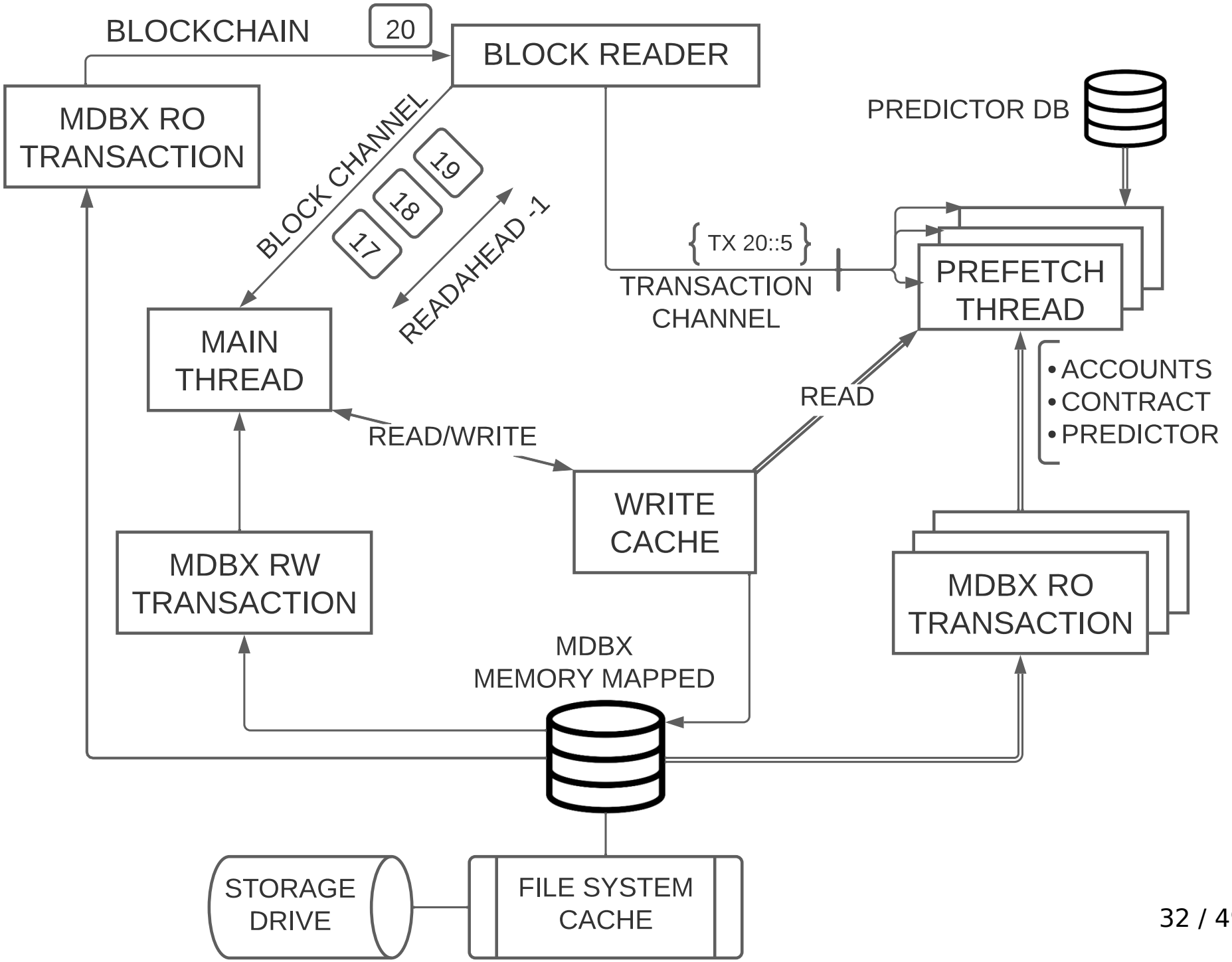
- Έξοδος σε binary encoding, εισάγεται στη DB

```
Key = 00a5e63813215d7783df9673e42ec7e1d2e5c0896f17e
96ef6f8d28f1e19f663 (original contract’s code hash)

Value = 6a000d00100000100005b00680000010d0067007c0
000015b00790098000010701e900cb00000016700f400d60000
01cd01fc00d9000001f4000501e4000001cd010701ec0000010
5013401f6000002e900fc00a30198010013401b701c3010001
a301cd01e4010001b701010100400102008090010020022 0d0
01d040001000000000000000000000000000000000000000
0000000000001050000040600...
```

# Prefetching

- Προσθήκη στον erigon, ξεχωριστό go package
- Βοηθητικά νήματα διαβάζουν από τη βάση όσο το κύριο νήμα κάνει πραγματική εκτέλεση
- Ένα νήμα διαβάζει μελλοντικά block και “ταΐζει” το κύριο
- Νήματα προανάγνωσης διαβάζουν accounts, κώδικα contract και εκτελούν τους predictor υποθετικά (όπου υπάρχουν)
- Τα νήματα προανάγνωσης παίρνουν transaction από το νήμα που διαβάζει block,
όποιο νήμα προλάβει κάθε φορά

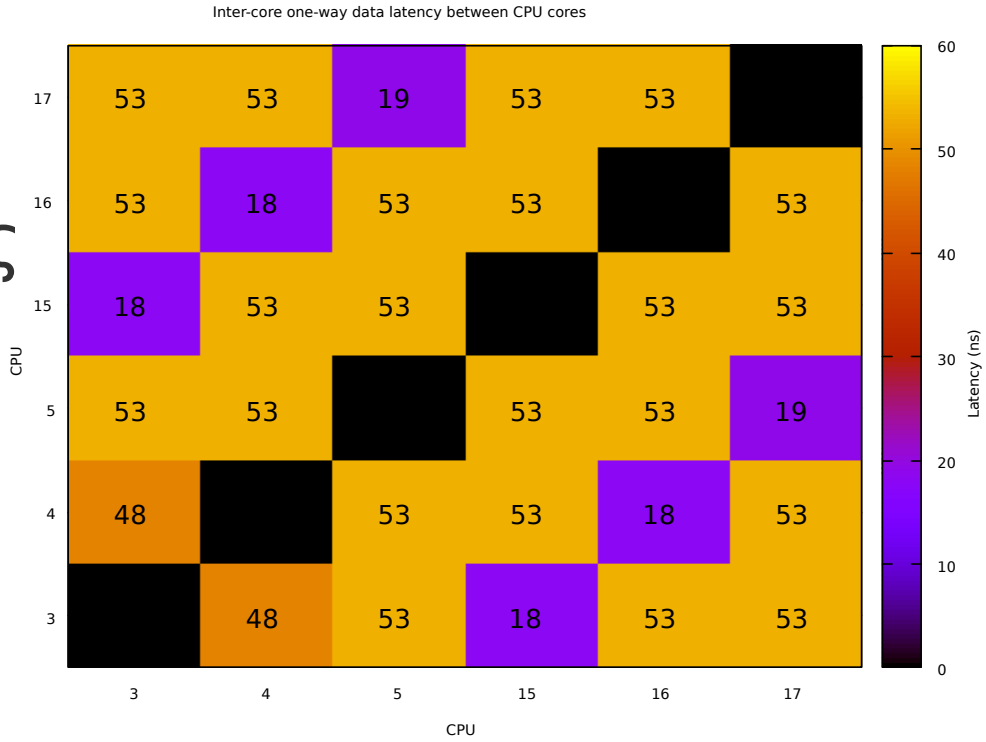

BLOCKCHAIN
20
BLOCK READER
MDBX RO
TRANSACTION
BLOCK CHANNEL
19
18
17
READAHEAD -1
PREDICTOR DB
TX 20::5
TRANSACTION
CHANNEL
PREFETCH
THREAD
MAIN
THREAD
READ
• ACCOUNTS
• CONTRACT
• PREDICTOR
READ/WRITE
WRITE
CACHE
MDBX RW
TRANSACTION
MDBX RO
TRANSACTION
MDBX
MEMORY MAPPED
STORAGE
DRIVE
FILE SYSTEM
CACHE

# Concurrency Control

- Ο erigon έχει μια write-back write-cache, in-memory βασισμένη σε β-δέντρα
- Έλεγχος με απλά spinlock, πολλοί readers ή ένας exclusive writer
- Αποτυχίες πάνε στη βάση
- Η βάση ακολουθεί MVCC, τα read-only transactions (βοηθητικά νήματα) δεν μπλοκάρουν με το ένα read-write (κύριο νήμα, αν και μόνο αναγνώσεις κάνει)

# Πειράματα

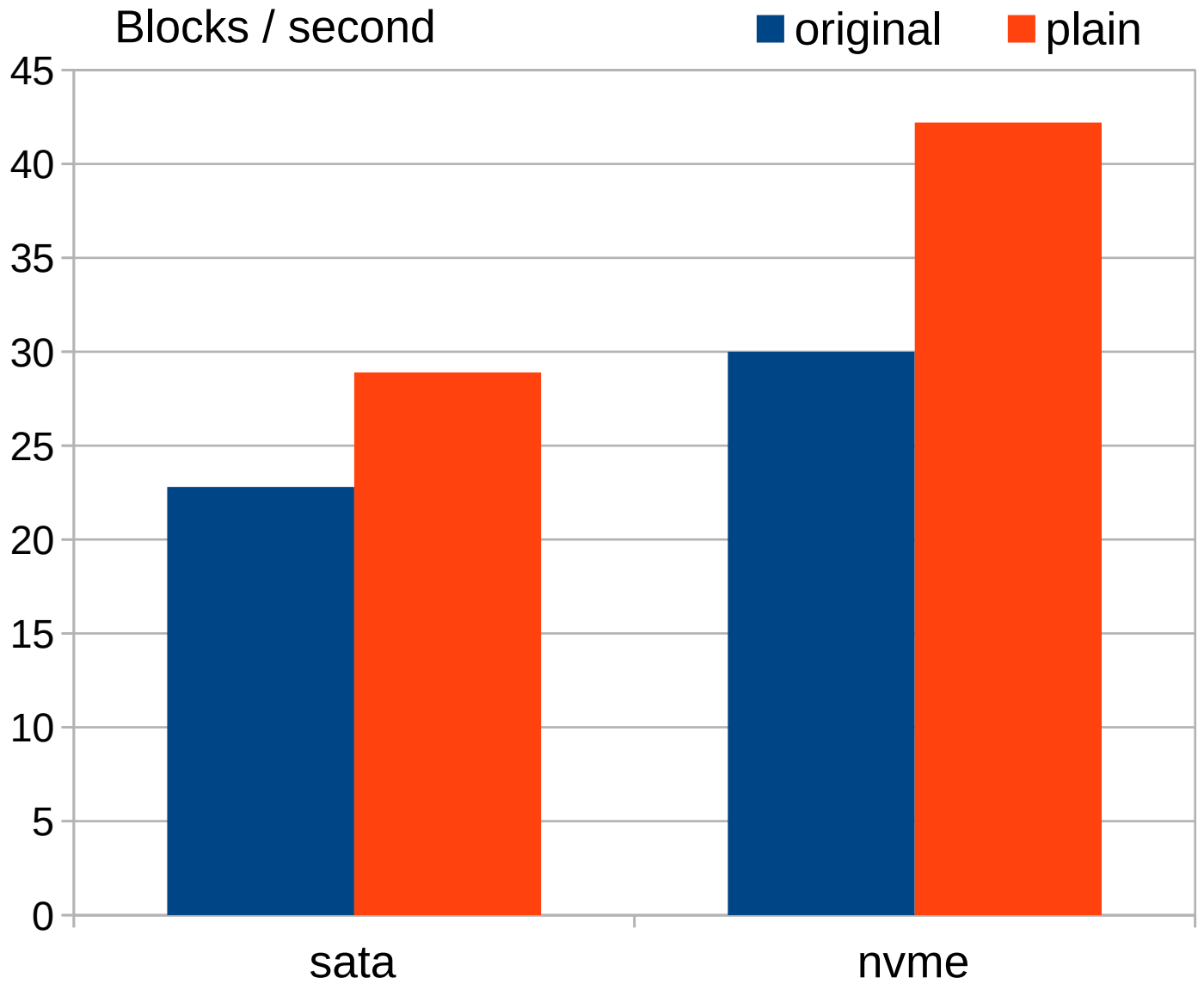
- Όλα στο ίδιο μηχάνημα, αλλάζει μόνο ο “δίσκος”
- CPU: zen 2, 12C/24T, 3.8 GHz, περιορίζουμε σε 1 CCX (3C/6T, κοινή L3)
- RAM: 64 GiB DDR4-3200 περιορίζουμε δεσμεύοντας με ξεχωριστό process
- Δίσκος 1: nvme tlc ssd
- Δίσκος 2: sata qlc ssd
- Θα αλλάζουν:
  Συχνότητα cpu, ram, χρονισμοί και διαθέσιμη ram, δίσκος, πλήθος νημάτων, readahead



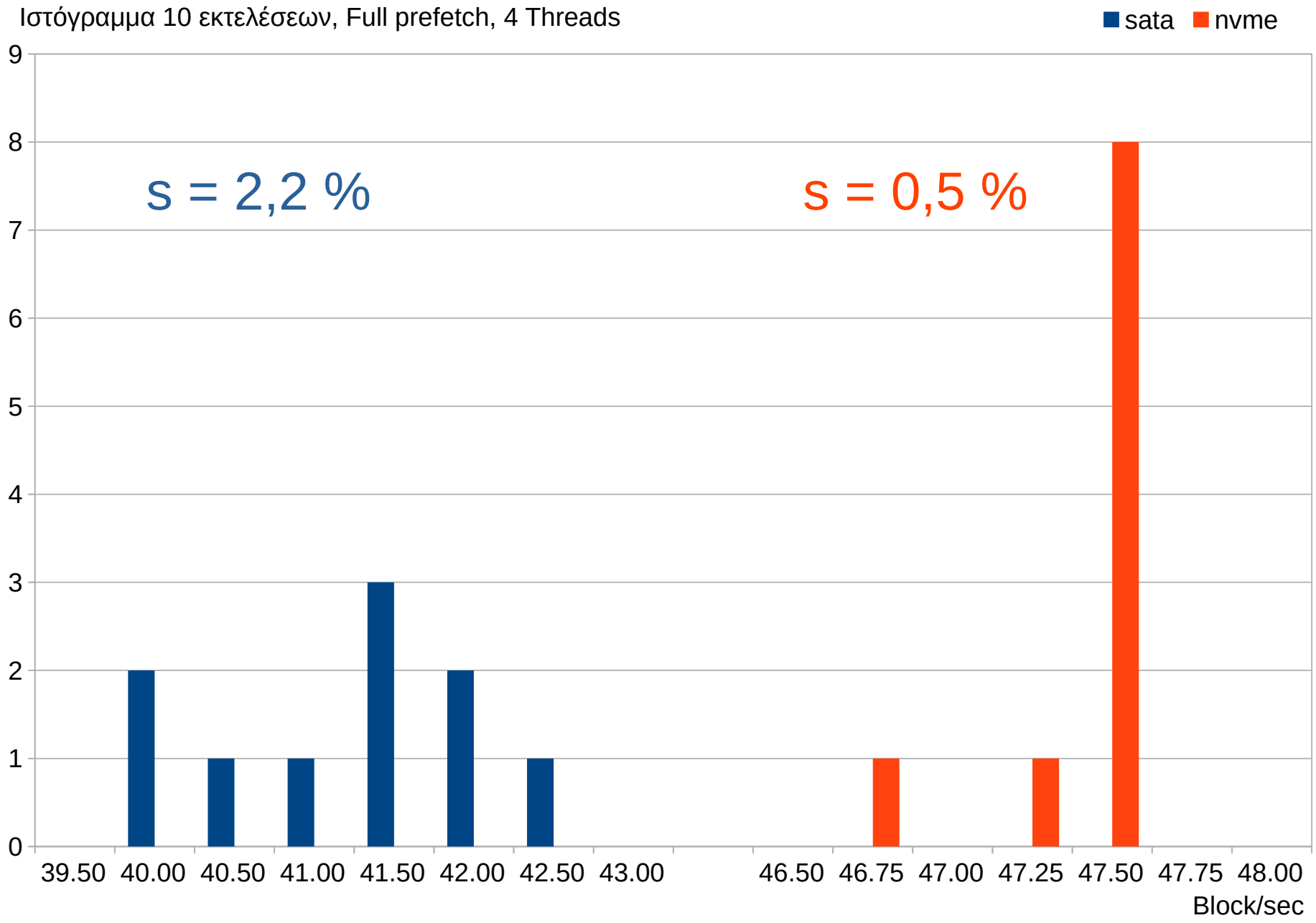
# Άλλες μικροβελτιώσεις

- Έχουν γίνει μικρές βελτιώσεις στον κώδικα του erigon και κάποια bug fixes
- Βελτιώνουν χρόνο εκτέλεσης, από πλευράς CPU
- πχ αφαίρεση εντολών debug
- Χωρίς tradeoff
- Ένα μεγάλο ποσοστό από feature του erigon που δεν ενεργοποιούταν λόγω bug
  (το code base είναι από τον Άυγουστο, μπορεί να έχει φτιαχτεί τώρα)
- Όλες οι επόμενες συγκρίσεις θα είναι ως προς τη βελτιωμένη έκδοση

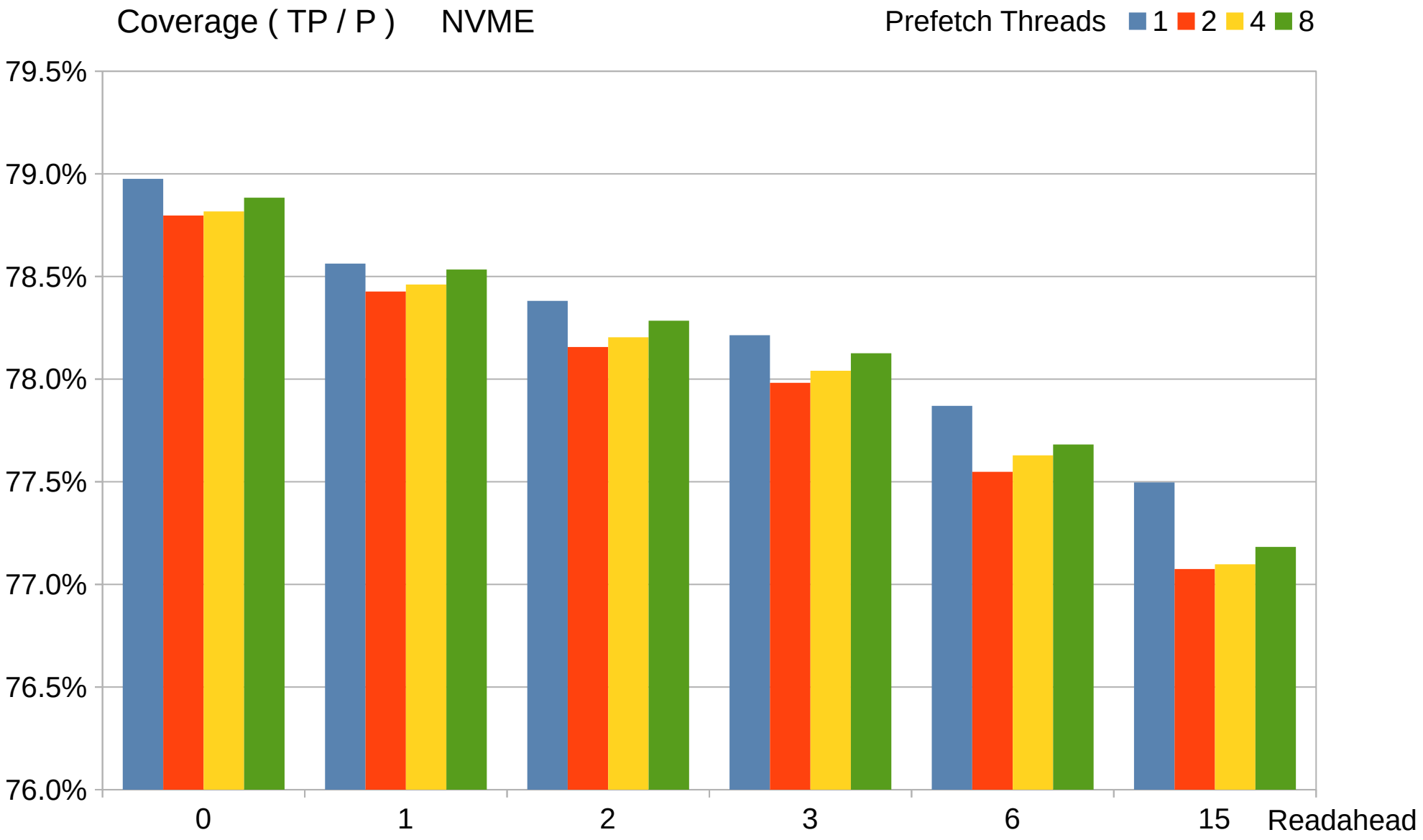
# Άλλες μικροβελτιώσεις



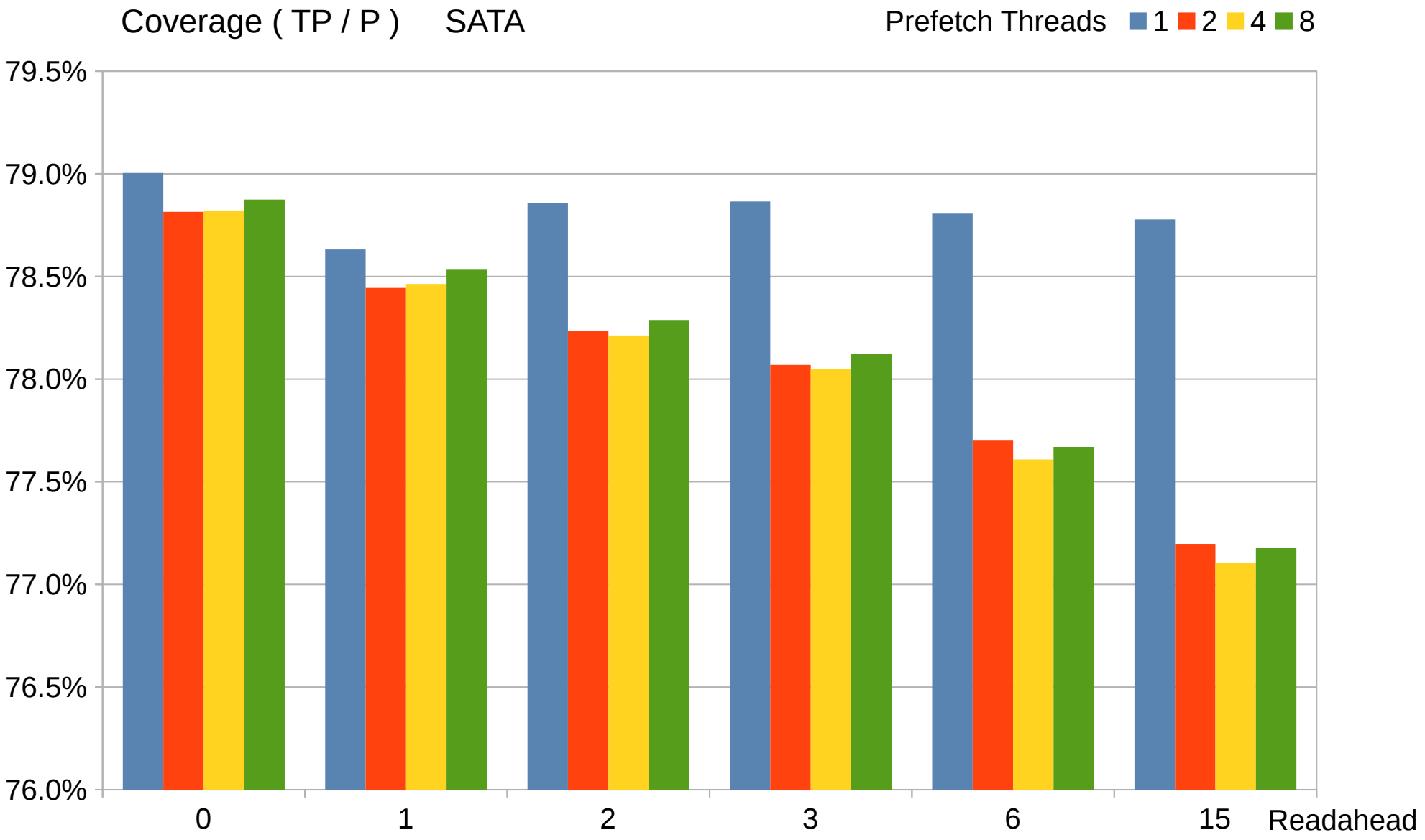
# Precision μετρήσεων ;



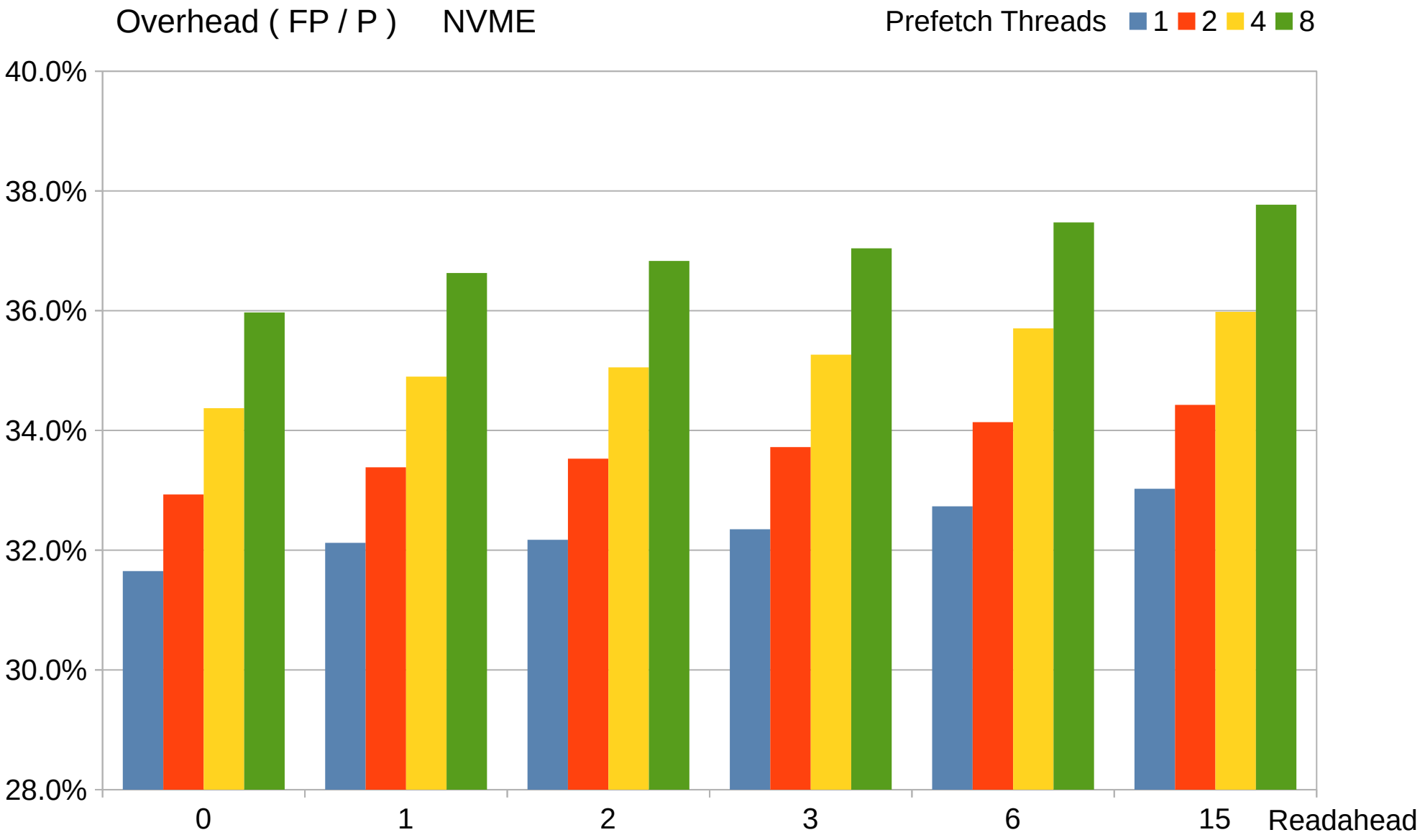
# Δοκιμή 1
# readahead και % επιτυχίας



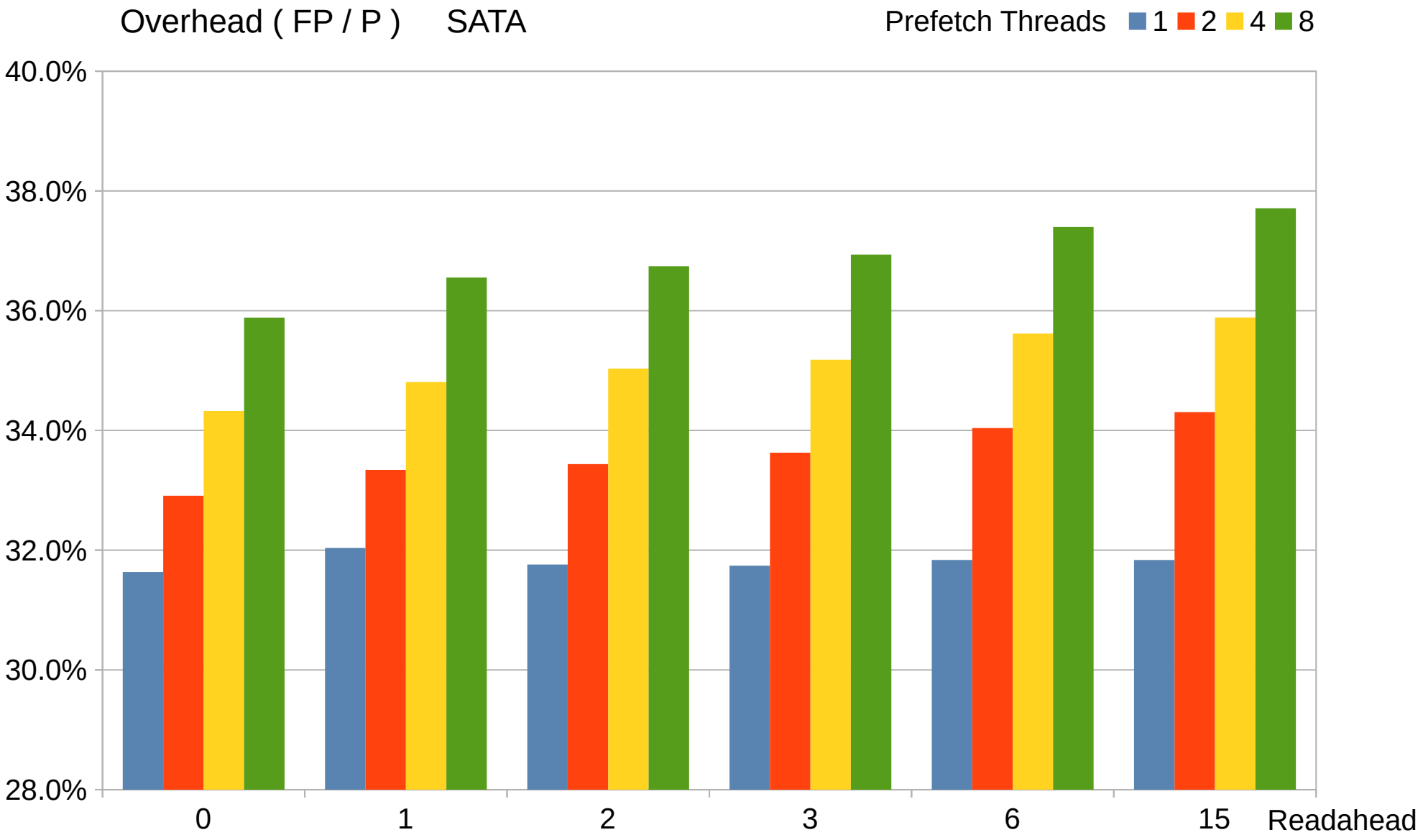
# Δοκιμή 1
# readahead και % επιτυχίας



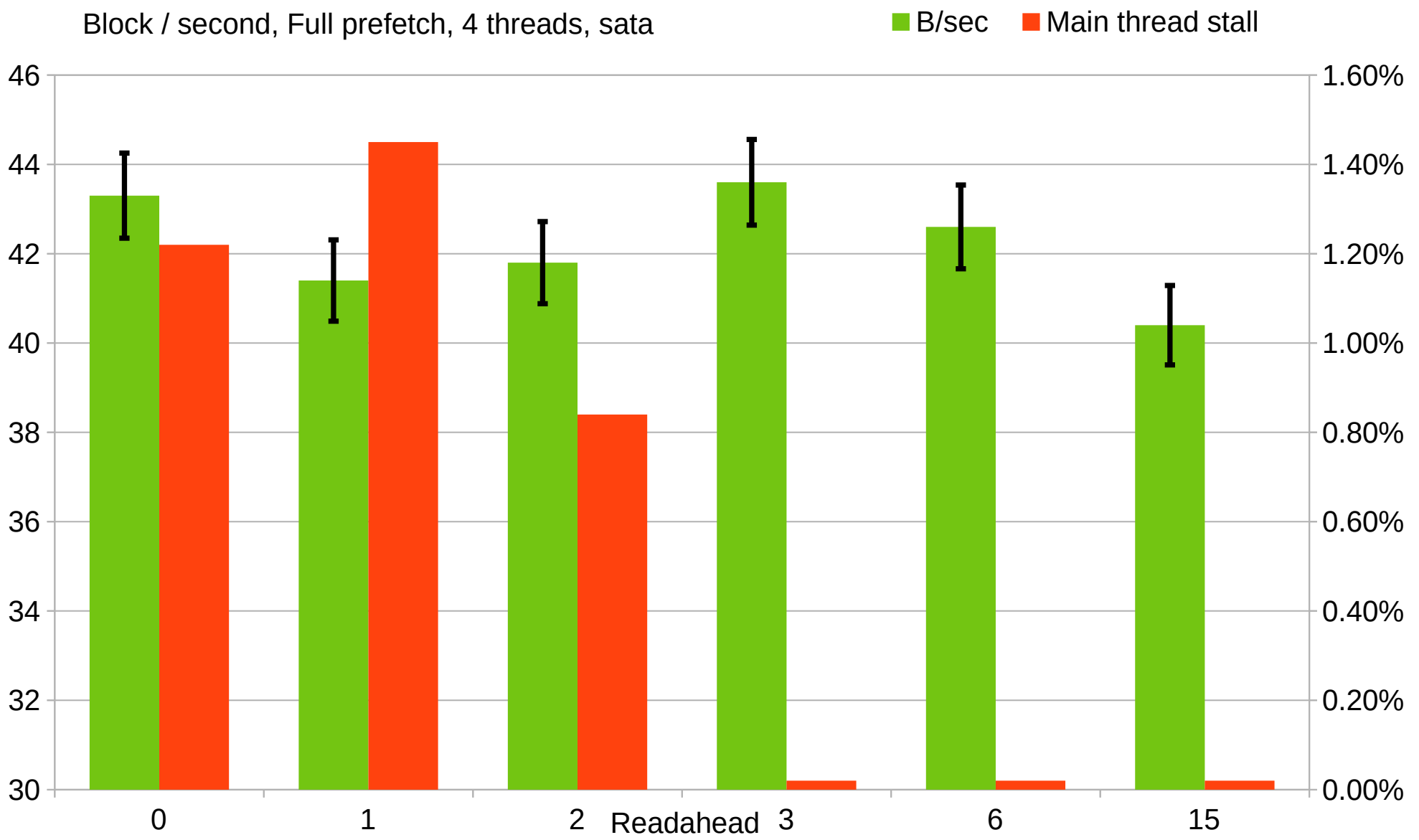
# Δοκιμή 1
# readahead και % επιτυχίας



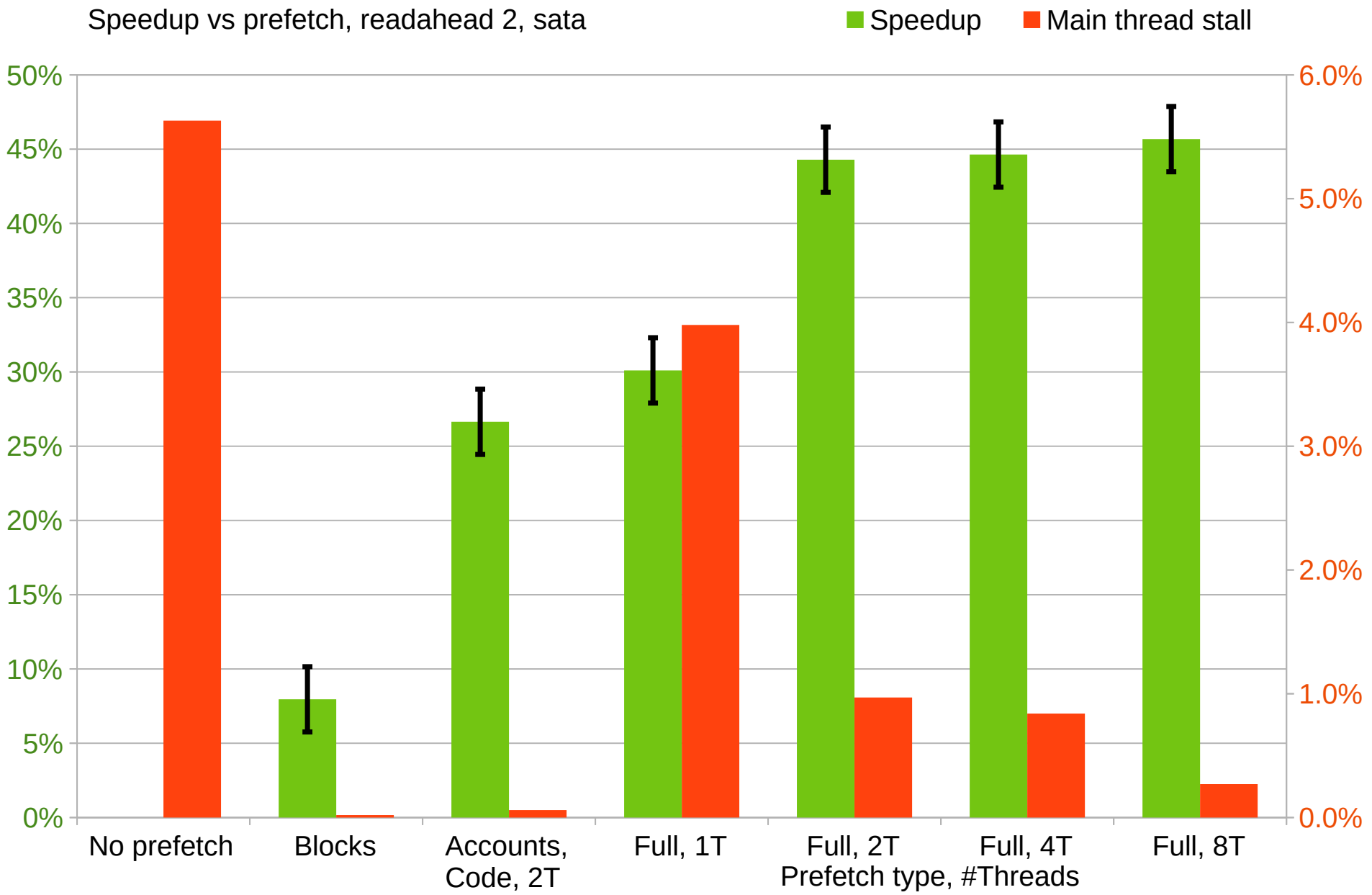
# Δοκιμή 1
# readahead και % επιτυχίας



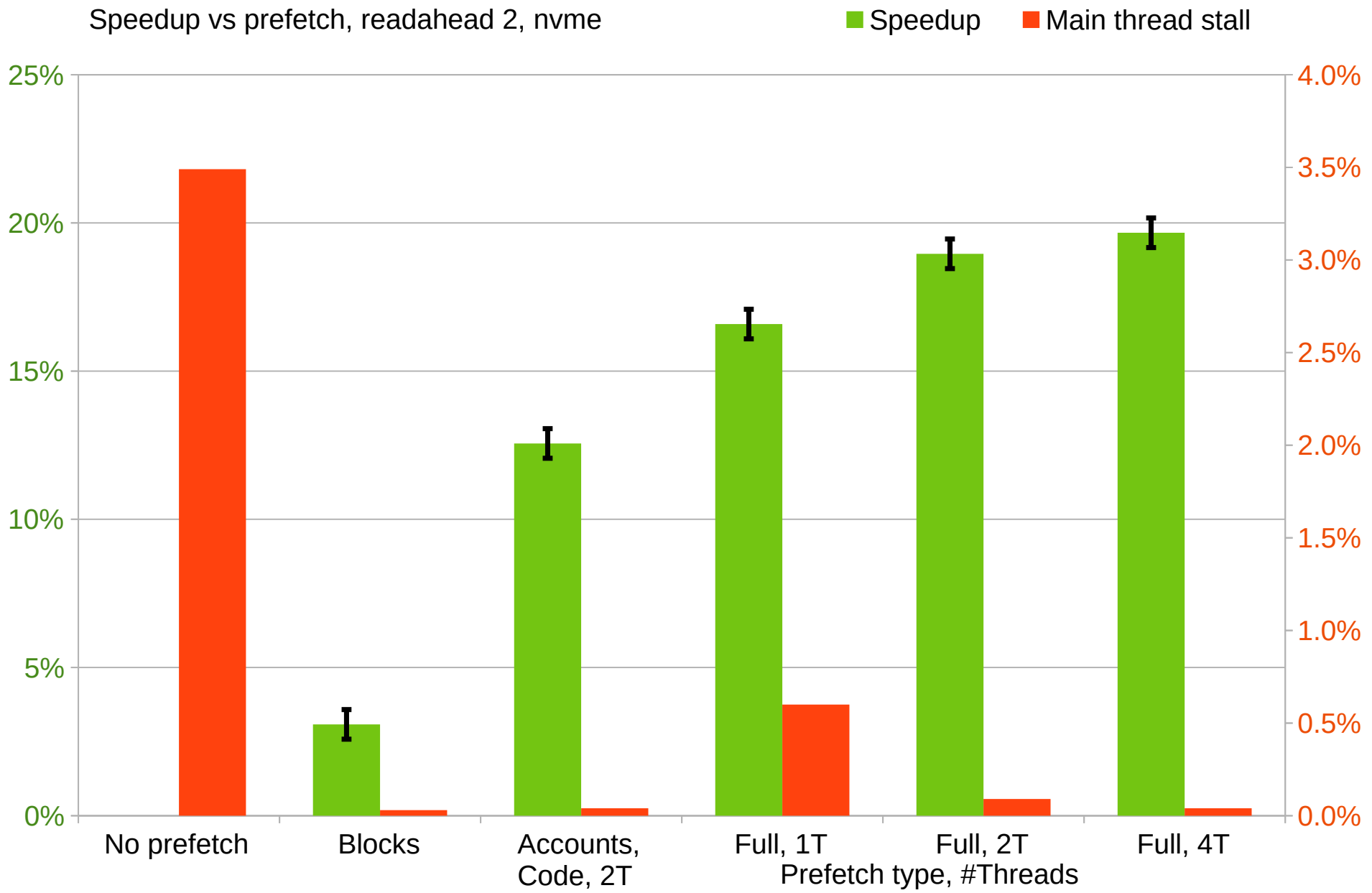
# Δοκιμή 2
# readahead και ταχύτητα



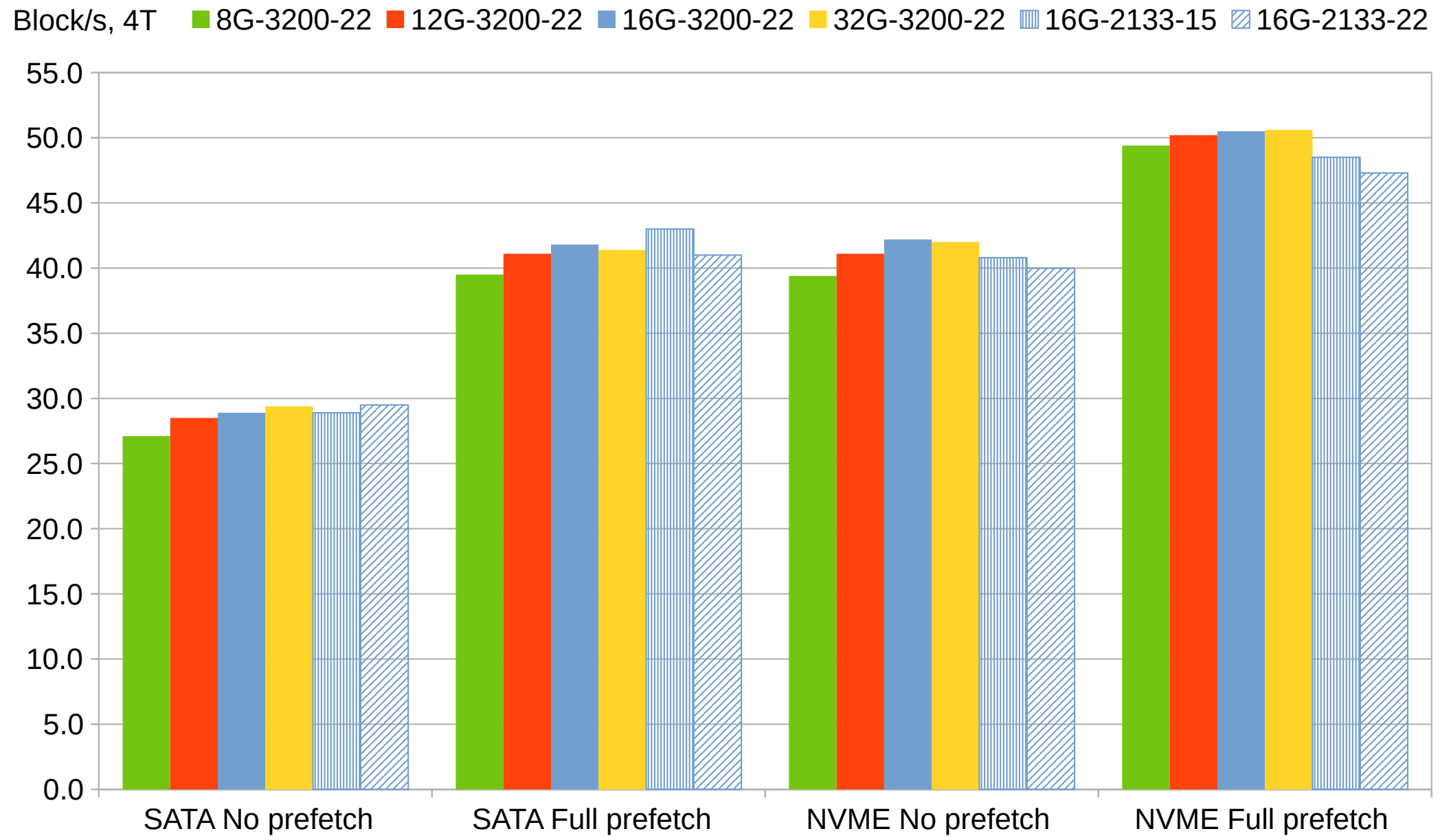
# Δοκιμή 3
# 16 GiB RAM, sata



# Δοκιμή 4
# 16 GiB RAM, nvme



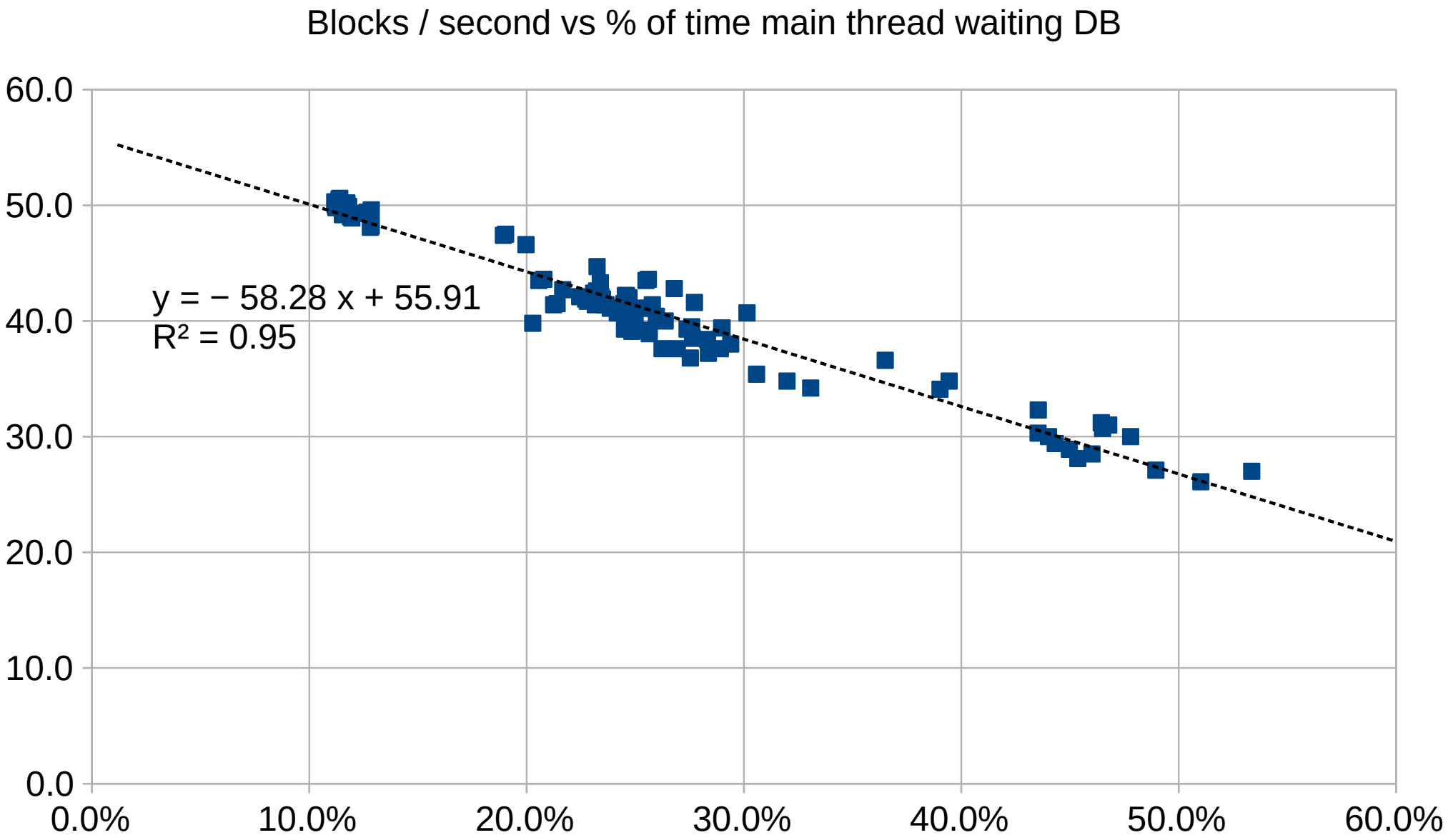
# Δοκιμή 5, μνήμη



# Δοκιμή 7
# συχνότητα CPU

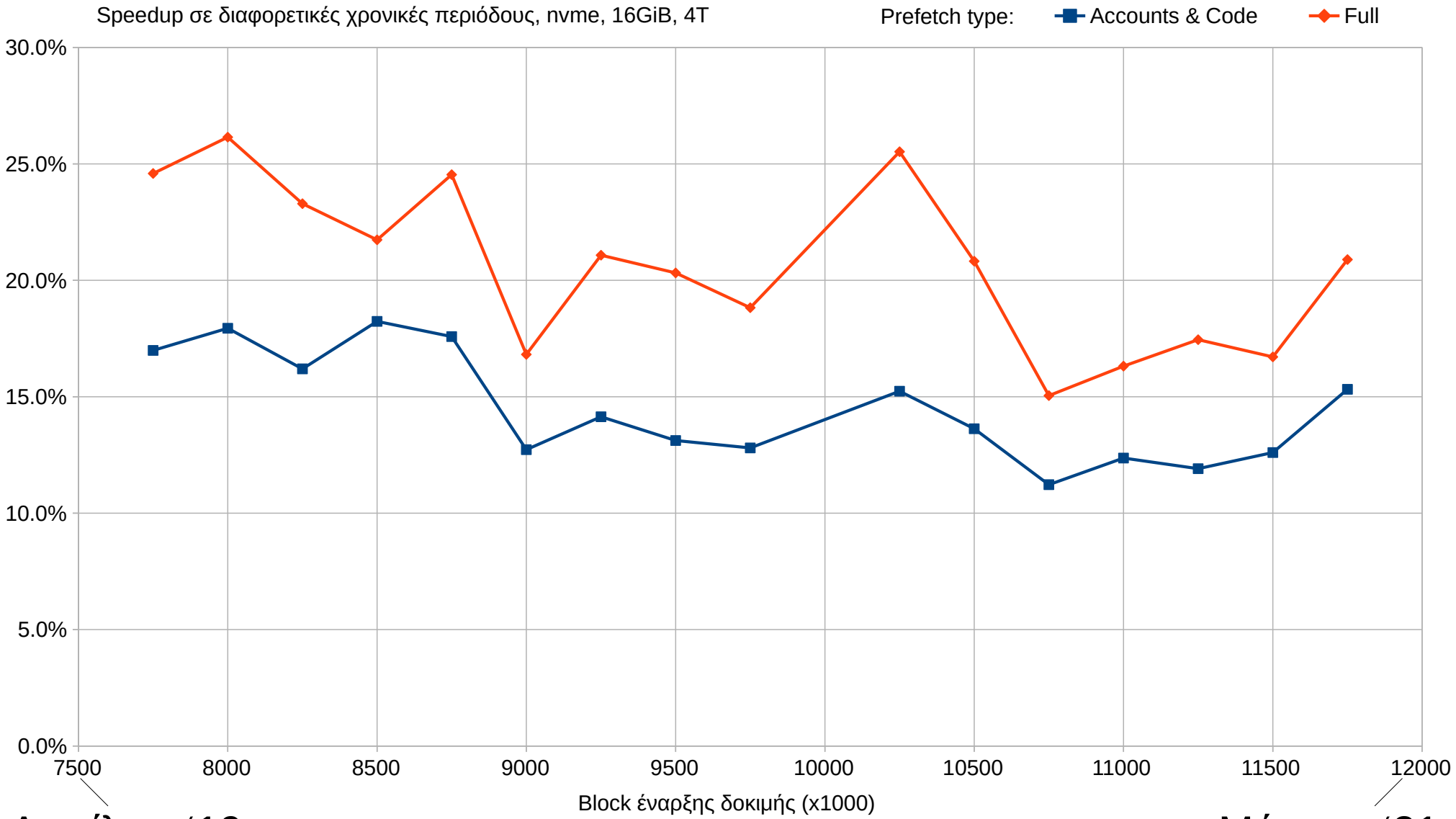
- Από 3.8 GHz → 2.8 GHz
- Καμία μετρήσιμη (πέραν σφάλματος) αλλαγή στην ταχύτητα εκτέλεσης
- Μπορεί να έγινε λανθασμένα η αλλαγή
- cpufreq-info δείχνει την επιθυμητή, governor "userspace", frequency asserted by call to hardware
- TODO: cpu-bound bench

# Scatter με όλες* τις δοκιμές




*εκτός από χρονισμών μνήμης και χρονικών περιόδων

# Χρονική εξέληξη




Απρίλιος ‘19

Μάρτιος ‘21

# Συμπεράσματα;