Prefetching in a Blockchain Client using static analysis and speculative execution of Smart Contracts

Diploma Thesis
Panagiotis Gkonis

Ethereum clients

- Ethereum network participants run a client
- It downloads the entire blockchain and runs all contained transaction
- A very time consuming process: runs for hours or even days on modern hardware
- Final disk usage: a few TB

World State

- It’s a set of K-V pairs, identical on all clients, modified by Transactions
- Contains accounts:
  Key=address → Value=account
- and for each contract, its storage:
  Key=slot → Value=content
- Structured as a modified Merkle Patricia Trie (MPT)

Merkle Patricia Trie

- Functions as a KV store
- Tree-like data structure
- The key is the path from root to a given node
- That node contains the corresponding value

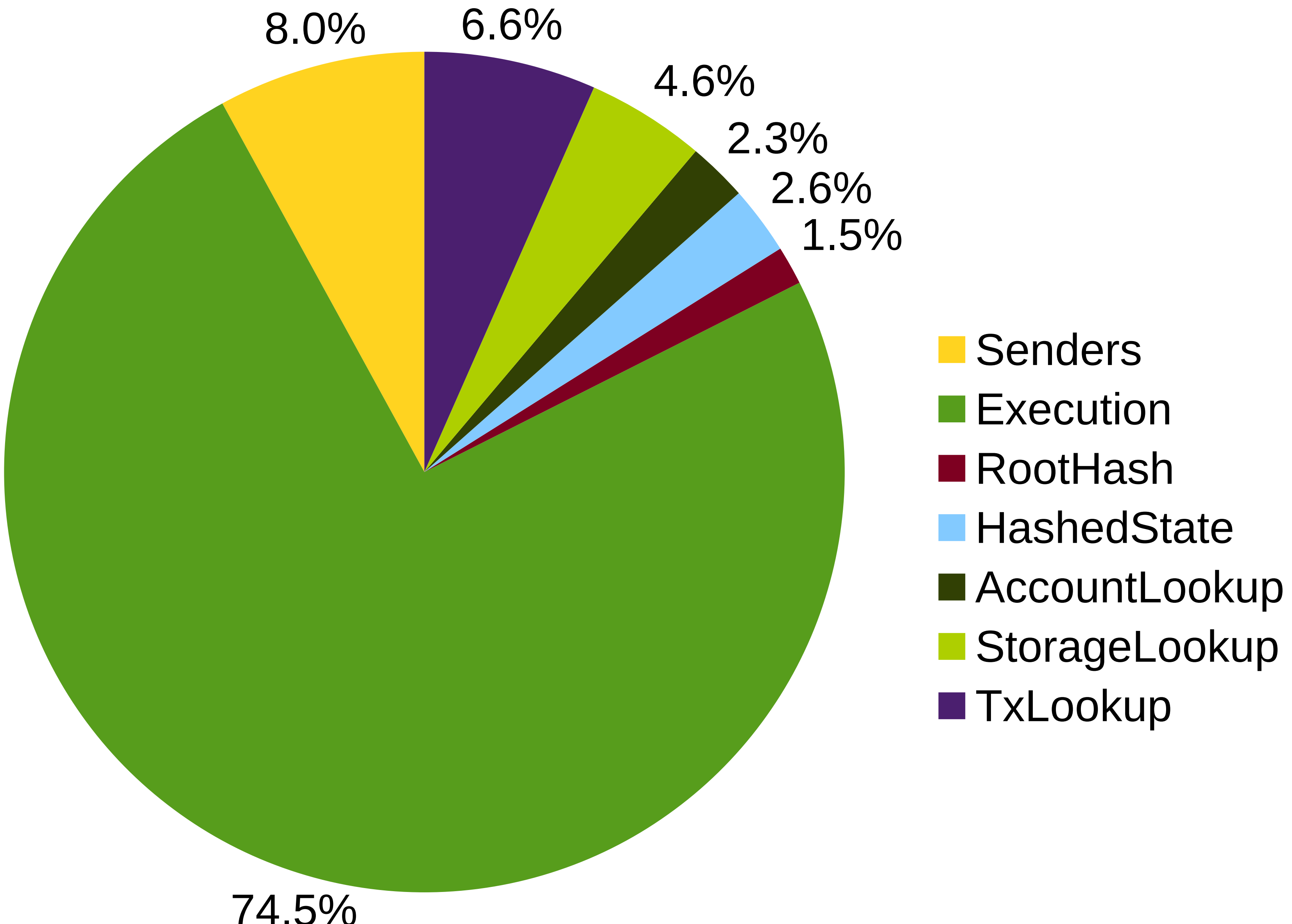
Go-ethereum

- Official client
- Internally implements a MPT
- which is stored in LevelDB
- Accessing historic state is trivial
- DB size of an archive node (2022): 9 TB

Erigon

- Fork of go-ethereum
- Works in stages, e.g. download, execute, …
- Does not use a Trie, instead storing in a “flat” data structure
- Database: MDBX (a fork of LMDB), mmap-ed
- Considerable improvement in space and time
- DB size of an archive node (2022): 1.5 TB

Stages (excl. network)



Execution Stage

- Our main focus: it executes blocks, transactions and smart contracts (SC)
- Sequential execution, due to unknown inter-transactional dependencies
- CPU and IO heavy
- Constant access to World State (DB), with time-consuming blocking reads
- If we used prefetching, the DB's pages would be cached in memory (FS cache), accelerating reads

→ precisely the purpose of this thesis

How much I/O;

- Test run of 30K blocks, measuring percentage of time spent in DB queries

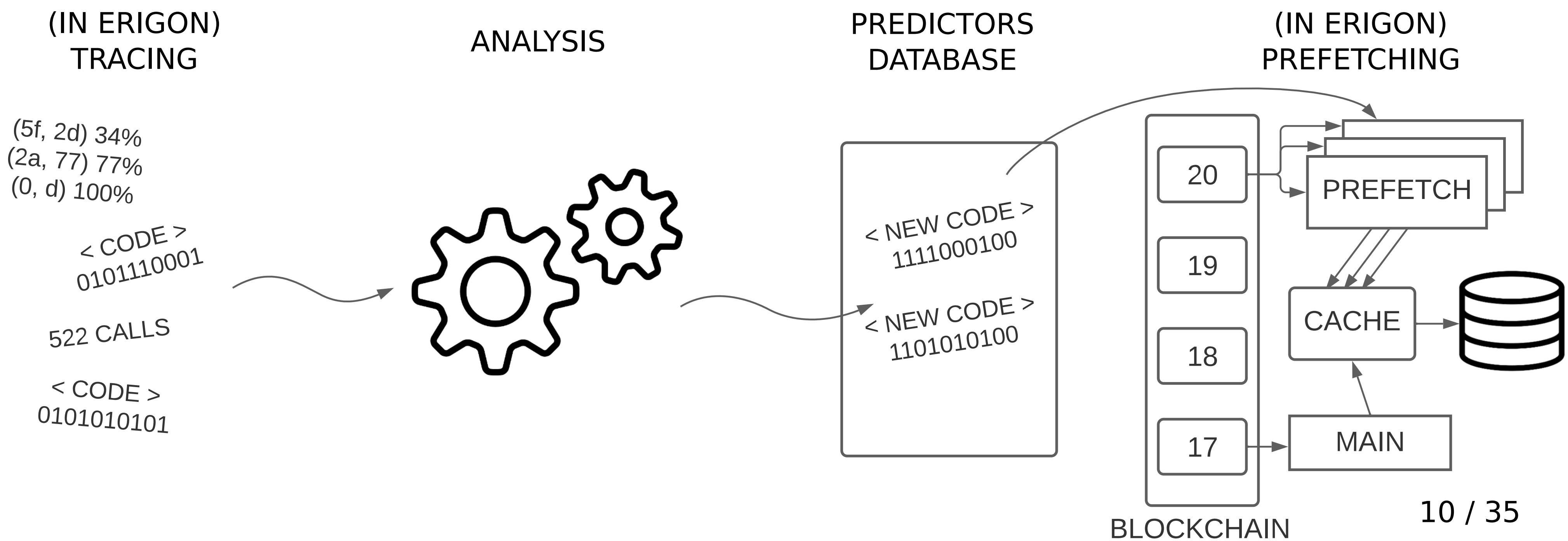
- ➔ With a fast NVME drive: 22 %
- ➔ With a slower sata SSD: 35 %

- There is an oportunity for considerable improvement

(no test was performed with a hard drive, as it would be 1-2 orders of magnitude slower)

System overview

- Tracing: collecting metrics and SC code
- Anslysis of collected SC and synthesis of new micro-programs (predictors)
- Prefetching and speculative execution of predictors

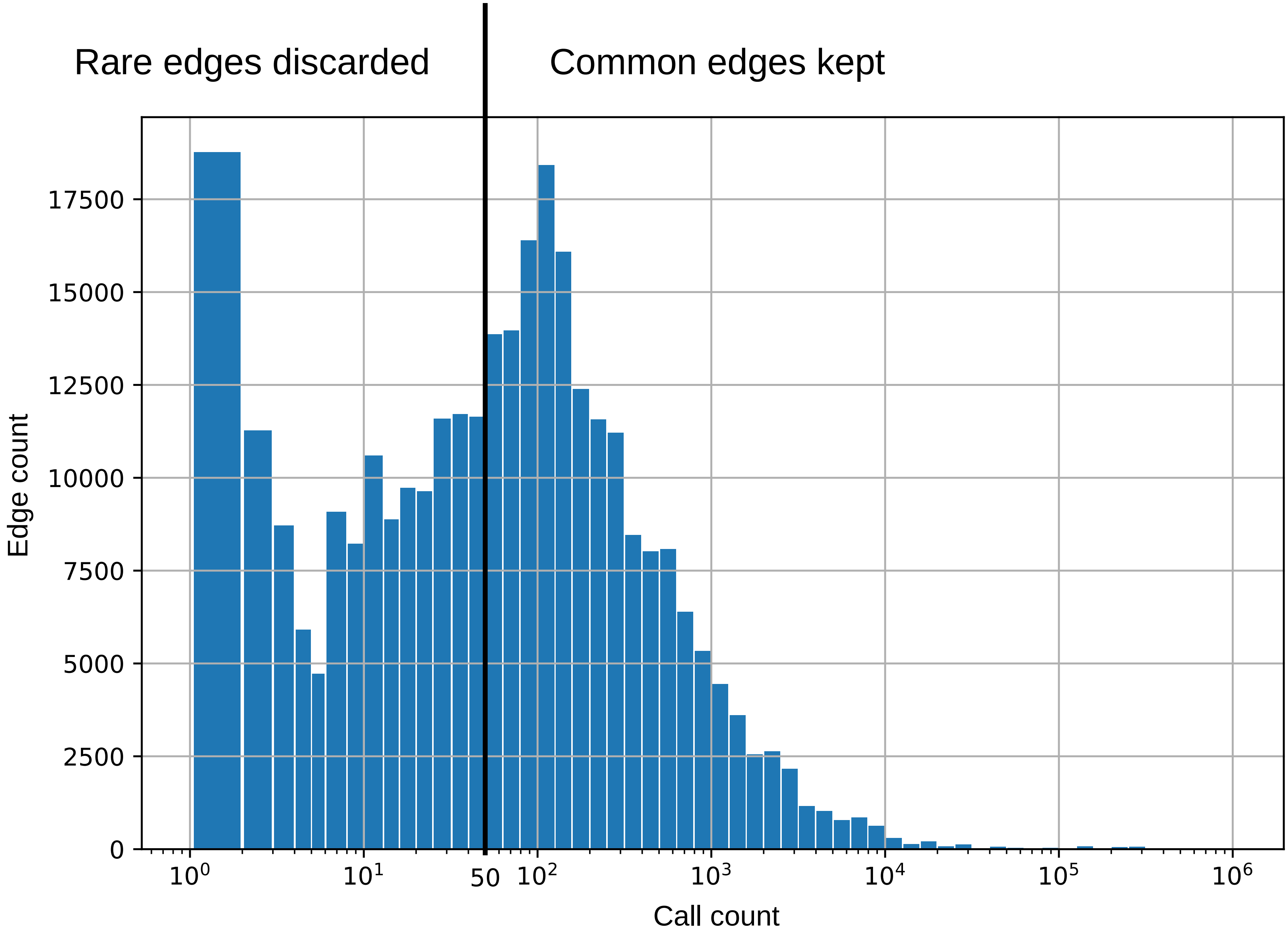


Tracing

- During the execution stage*, the SC code is collected and the number of calls to each one
- Frequent SCs are saved for analysis
- Also collected, for each JUMP, the PC before and after, as well as the unique ID of the call being executed
- These JUMP address pairs (CFG edges) with “many” calls are given as hints to the analyzer

**here realized with a transaction sample, offline, though in a real implementation would be online*

Tracing


Rare edges discarded
Common edges kept
Edge count
17500
15000
12500
10000
7500
5000
2500
0
10^0
10^1
50
10^2
10^3
10^4
10^5
10^6
Call count

Analysis

- Separate process, in Python
- Input is SC code, output is micro-programs (predictors) with similar code
- Predictors contain only “useful” instructions: storage access, calls, etc.
- Behavior can differ from the original code
- Opportunities for unsafe optimization, e.g. removing rare instructions

Analysis 1/12

- Takes as input the code of a SC, which is in EVM bytecode

608060405260043610610056576 3ffffffff7c010000000000000000000
0000000000000000000000000000000000000600035041663 6ae17a
b7811461005b578063771c0ad91461008d578063e80db5db146100a
b575b600080fd5b348015610067576 00080f...

Analysis 2/12

- It disassembles

```
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 0x10000000000000...
00000030: PUSH1 0x0
00000032: CALLDATALOAD
00000033: DIV
00000034: AND
....
```

Analysis 3/12

- Breaks it down into 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
```

Analysis 4/12

- Converts instructions to SSA form

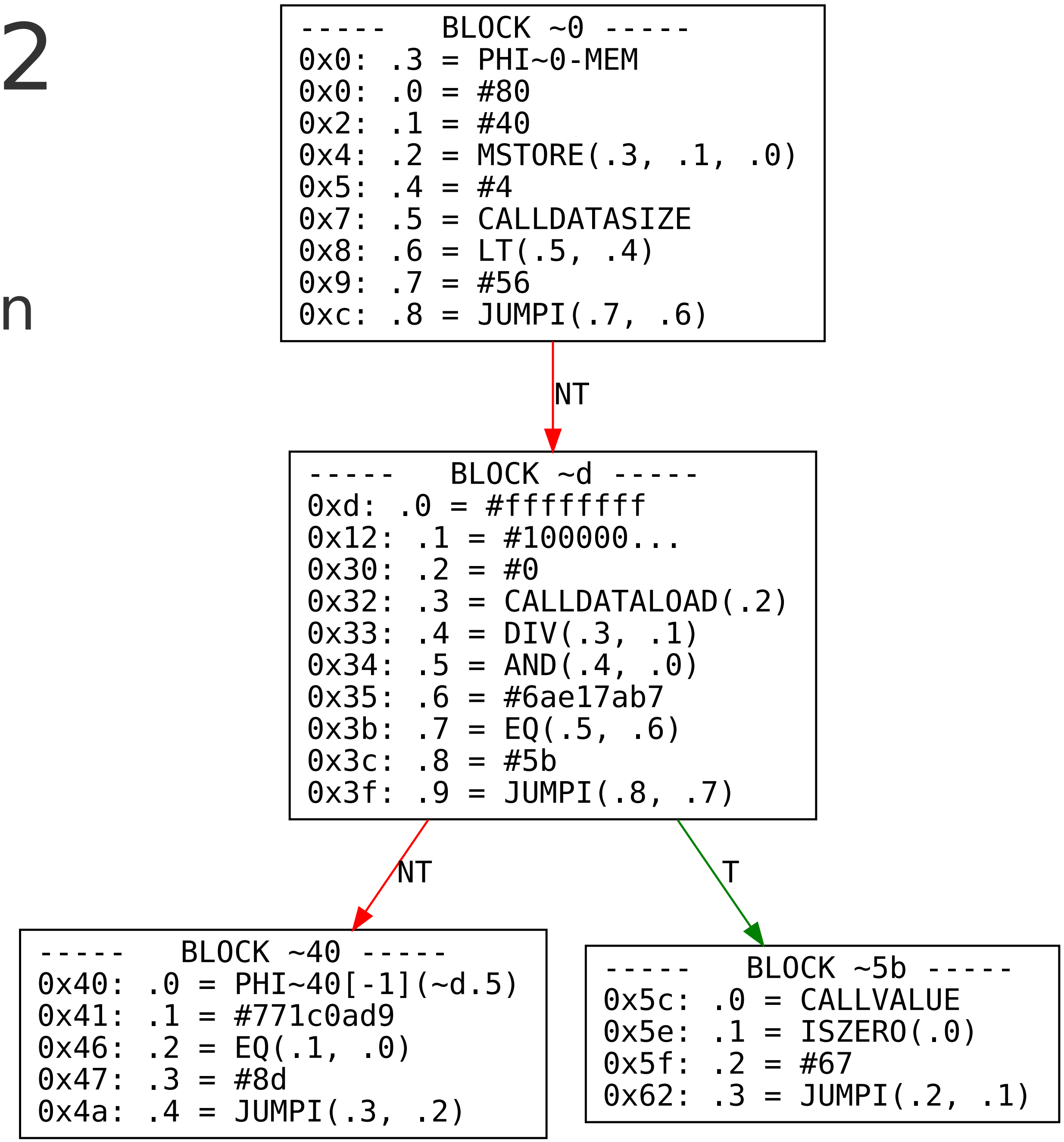
```
-----   BLOCK ~0 -----
0x0: .3 = PHI~0-MEM
0x0: .0 = #80
0x2: .1 = #40
0x4: .2 = MSTORE(.3, .1, .0)
0x5: .4 = #4
0x7: .5 = CALLDATASIZE
0x8: .6 = LT(.5, .4)
0x9: .7 = #56
0xc: .8 = JUMPI(.7, .6)
```

```
-----   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)
```

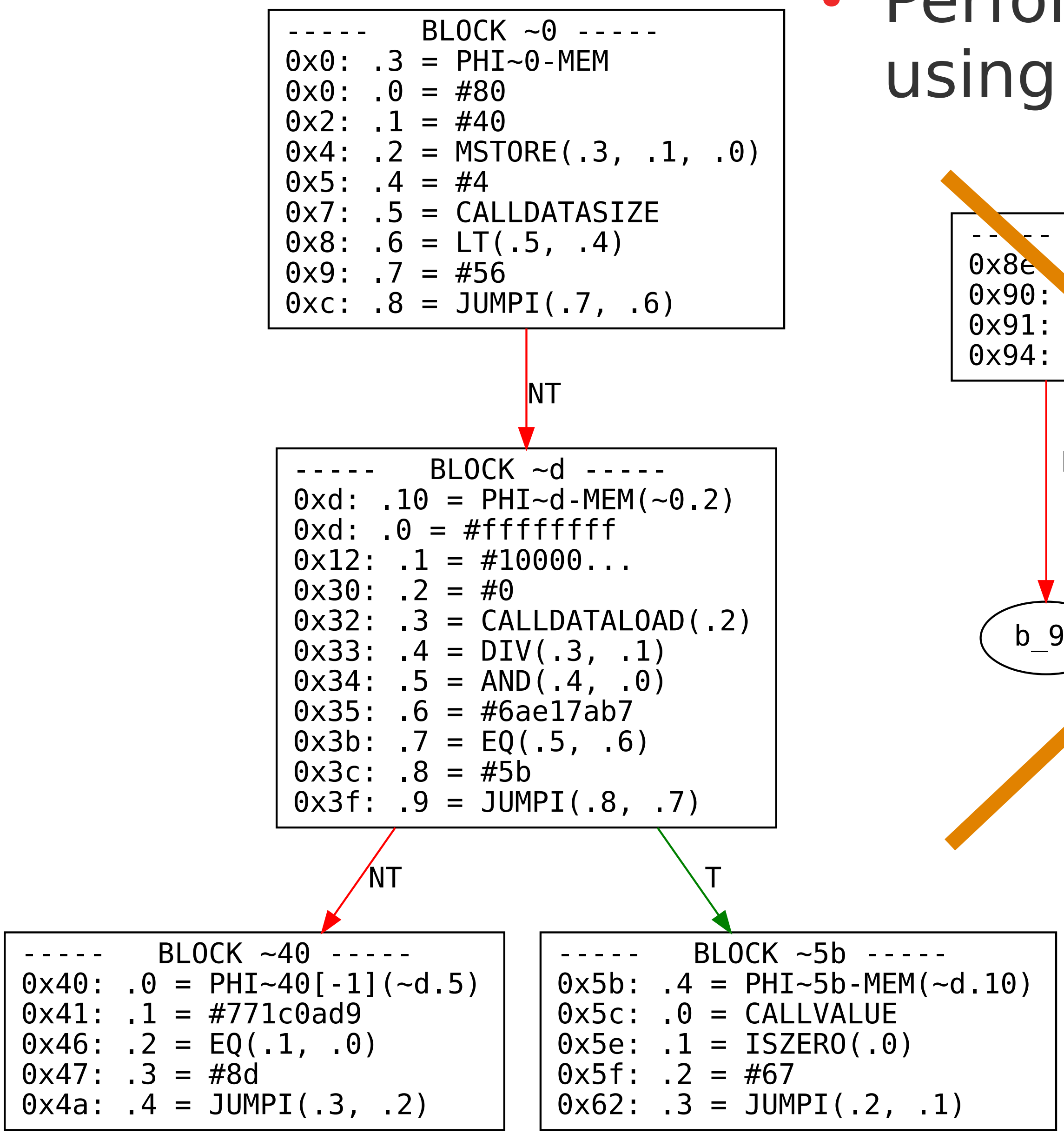
Analysis 5/12

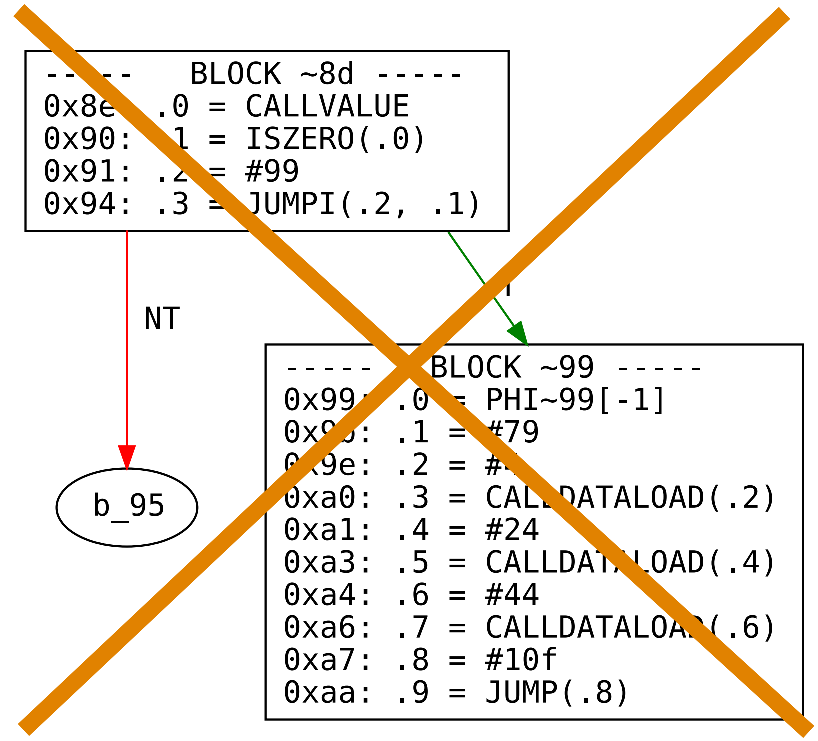
- Connects blocks, initial CFG estimation



Analysis 6/12

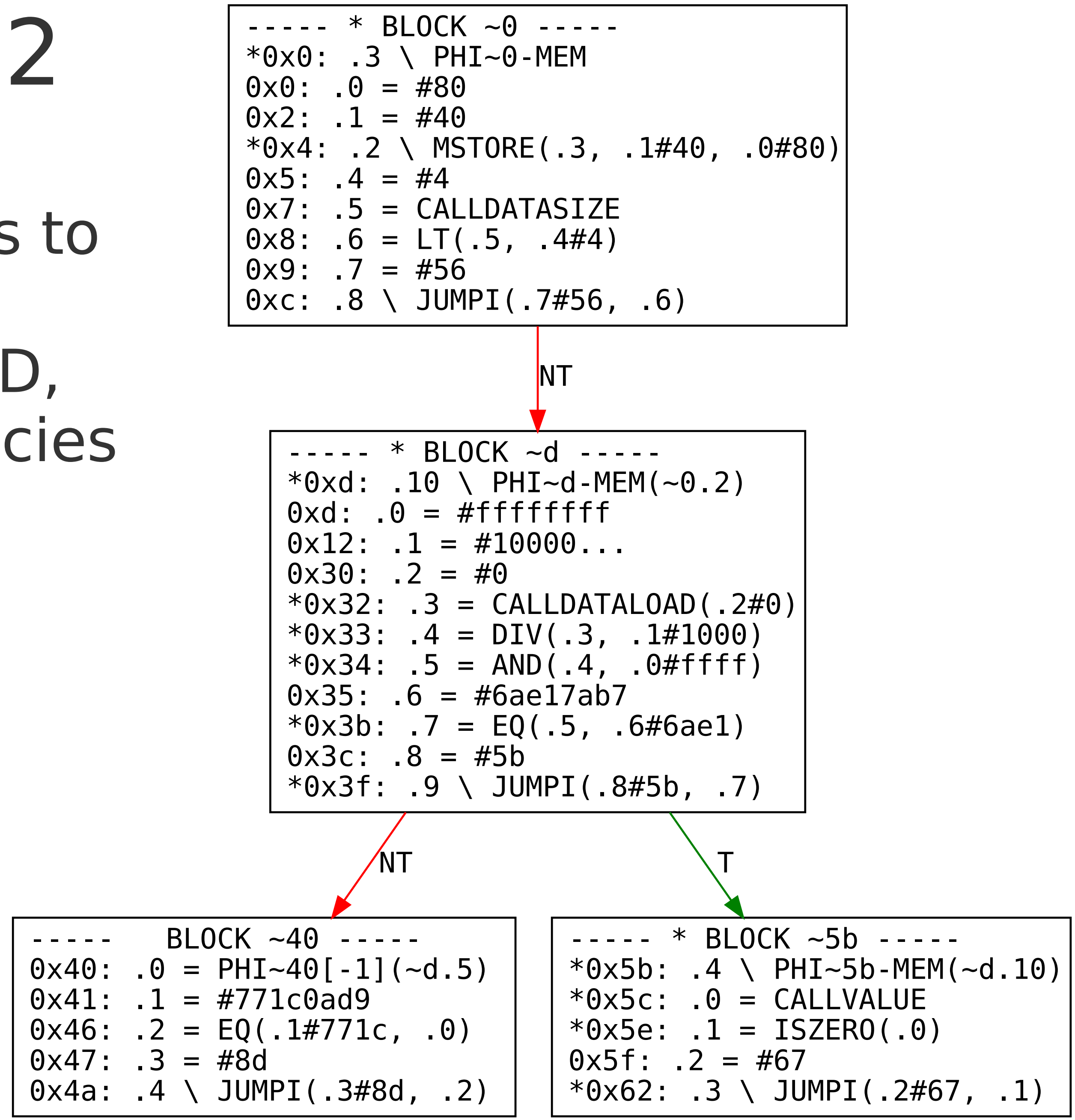
- Performs optimizations, using the worklist algorithm





Analysis 7/12

- Chooses instructions to be included in the predictor, e.g. SLOAD, CALL and dependencies



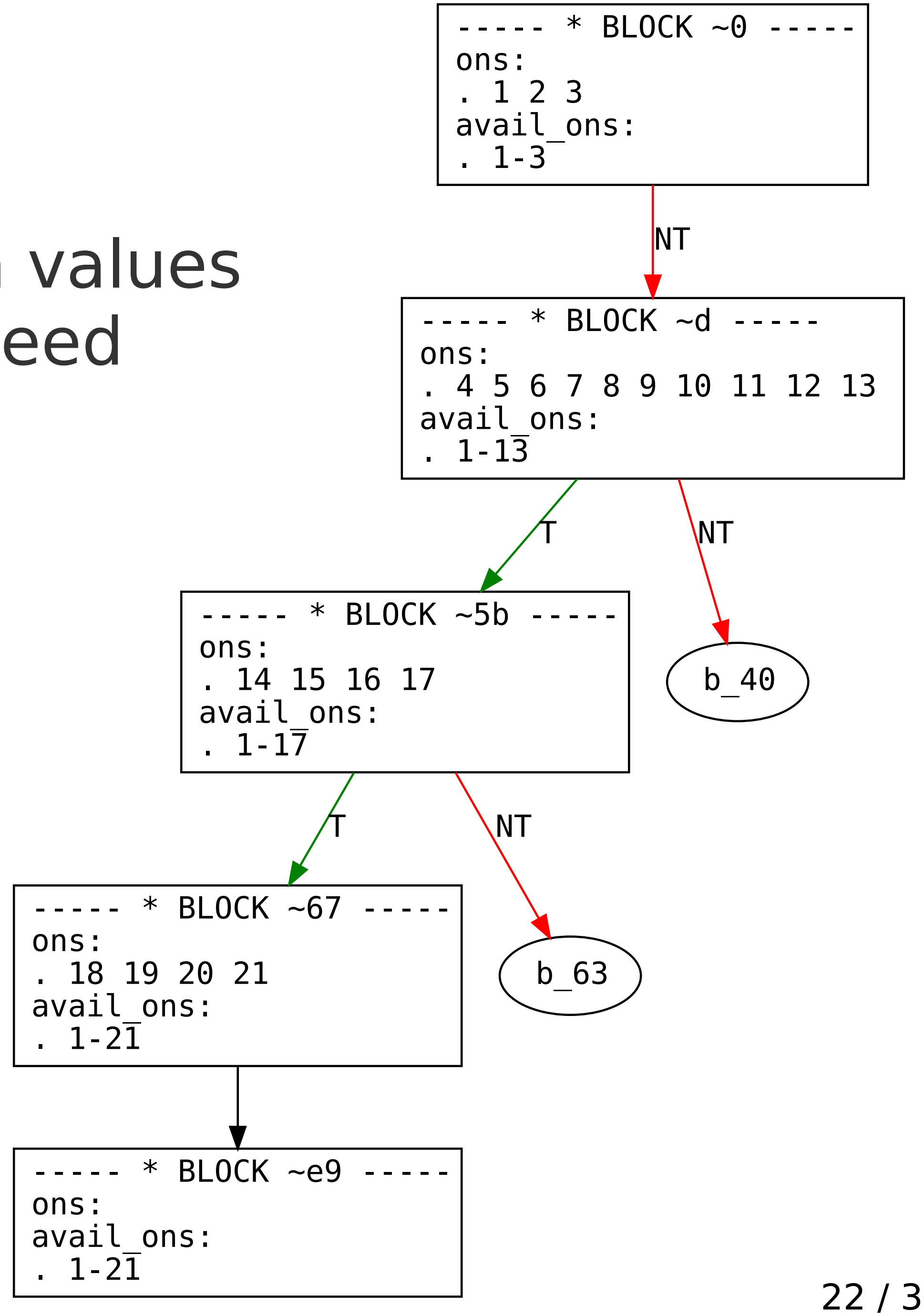
Analysis 8/12

- Enumerates values of chosen instructions

```
----- 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
```

Analysis 9/12

- In each block, finds which values are available and which need to be computed (common subexpression elimination)



Analysis 10/12

- Calculates the new expressions

```
----- 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
```

Analysis 11/12

- Synthesizes predictor code

```
~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
....
```

Analysis 12/12

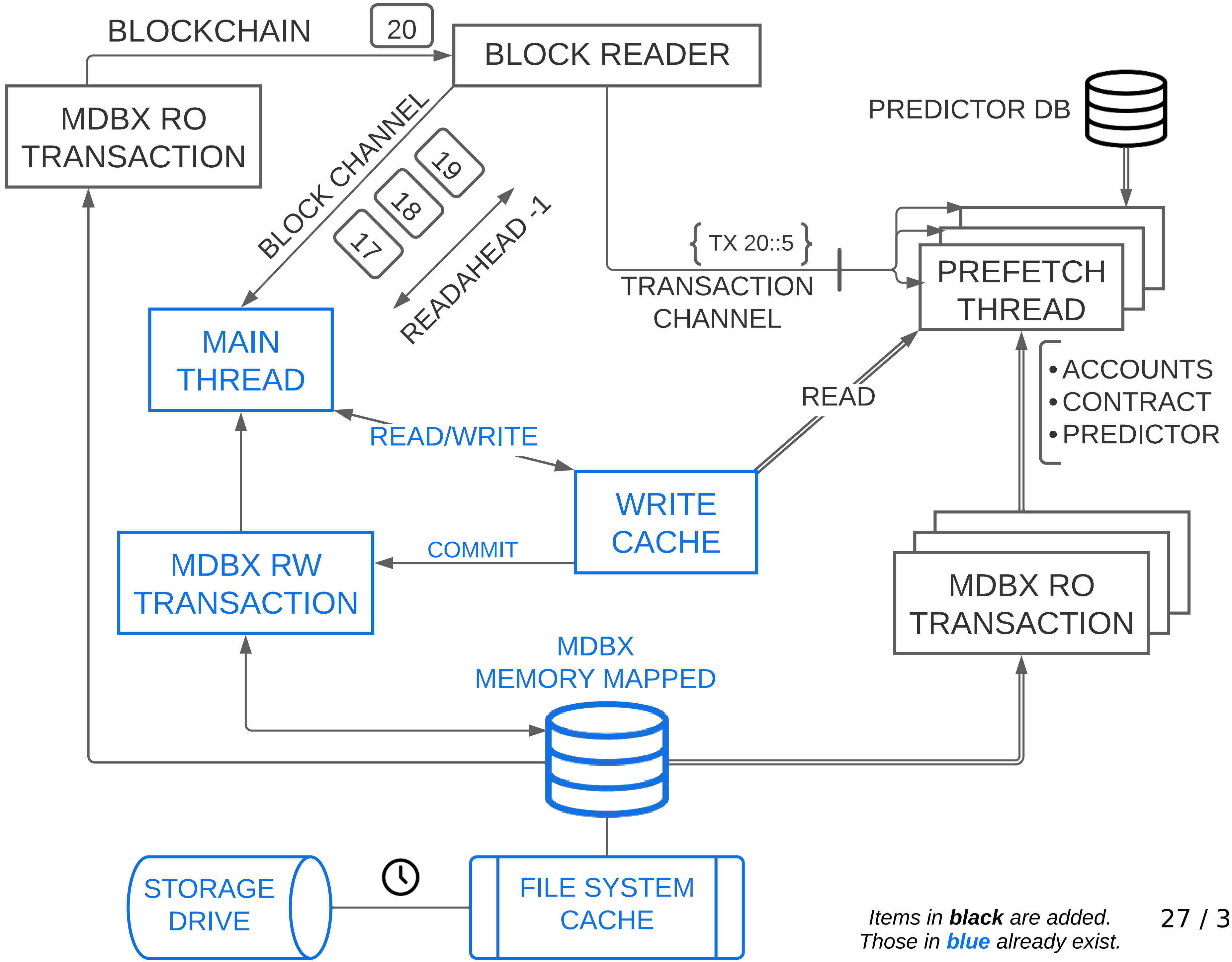
- Outputs in binary encoding, imported in the DB

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

Value = 6a000d0010000000100005b00680000010d0067007c0
000015b00790098000001070le900cb0000016700f400d60000
01cd01fc00d9000001f4000501e4000001cd010701ec0000010
5013401f6000002e900fc00a30198010001340lb701c3010001
a301cd01e4010001b70101010040010200809001000200220d0
01d040001000000000000000000000000000000000000000000
0000000000000000105000004060O...
```

Prefetching

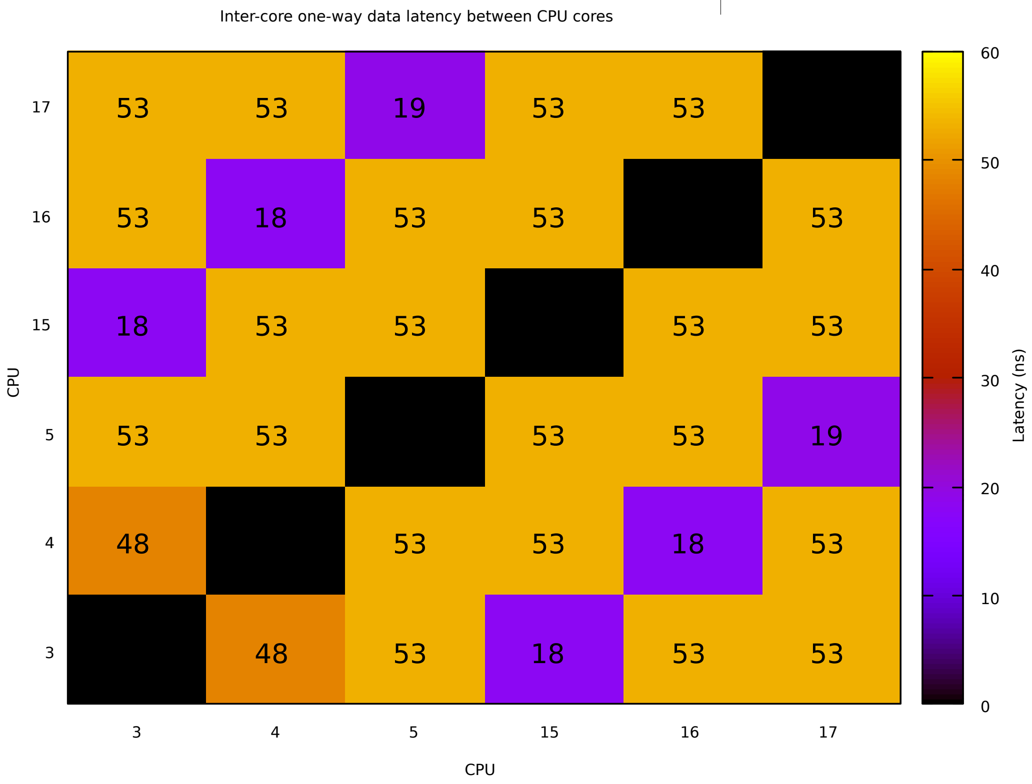
- Added to erigon, as a separate go package



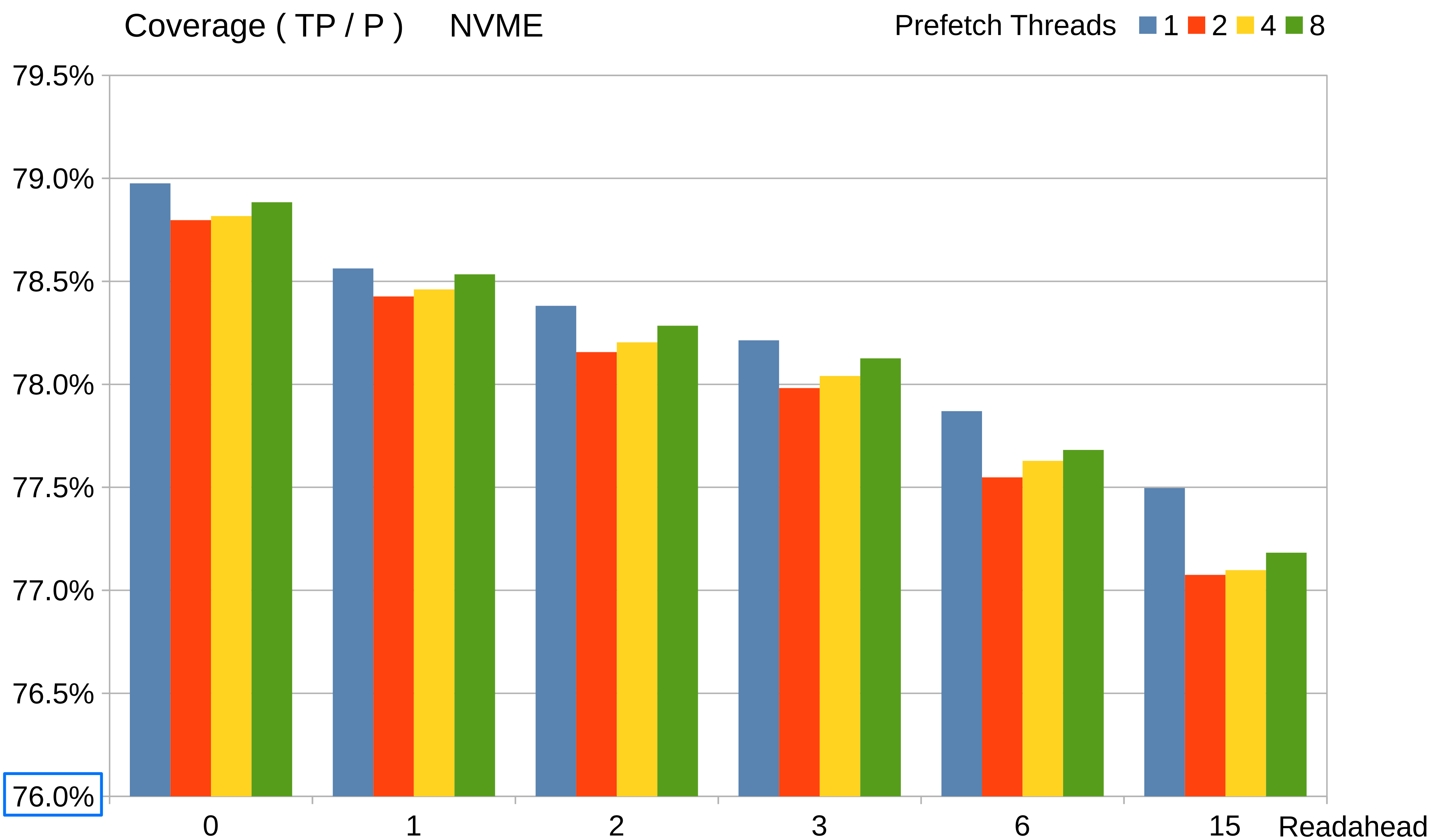
*Items in **black** are added.*
*Those in **blue** already exist.*

Experiments

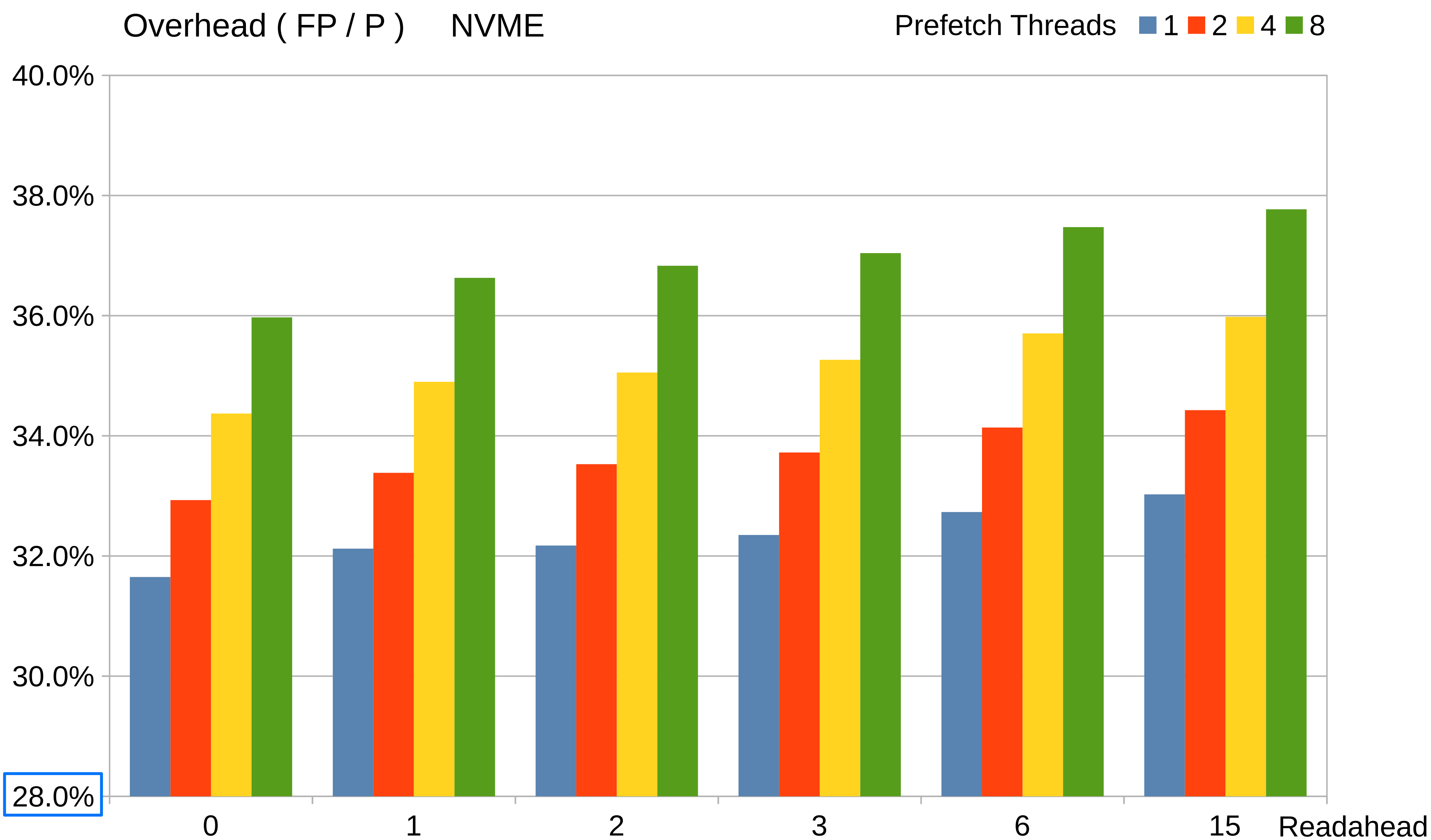
- All on the same machine
- CPU: zen 2, 3.8 GHz, limited to 1 CCX 3C/6T, shared L3
- RAM: limited by reserving using separate process
- Disk 1: nvme tlc ssd
- Disk 2: sata qlc ssd



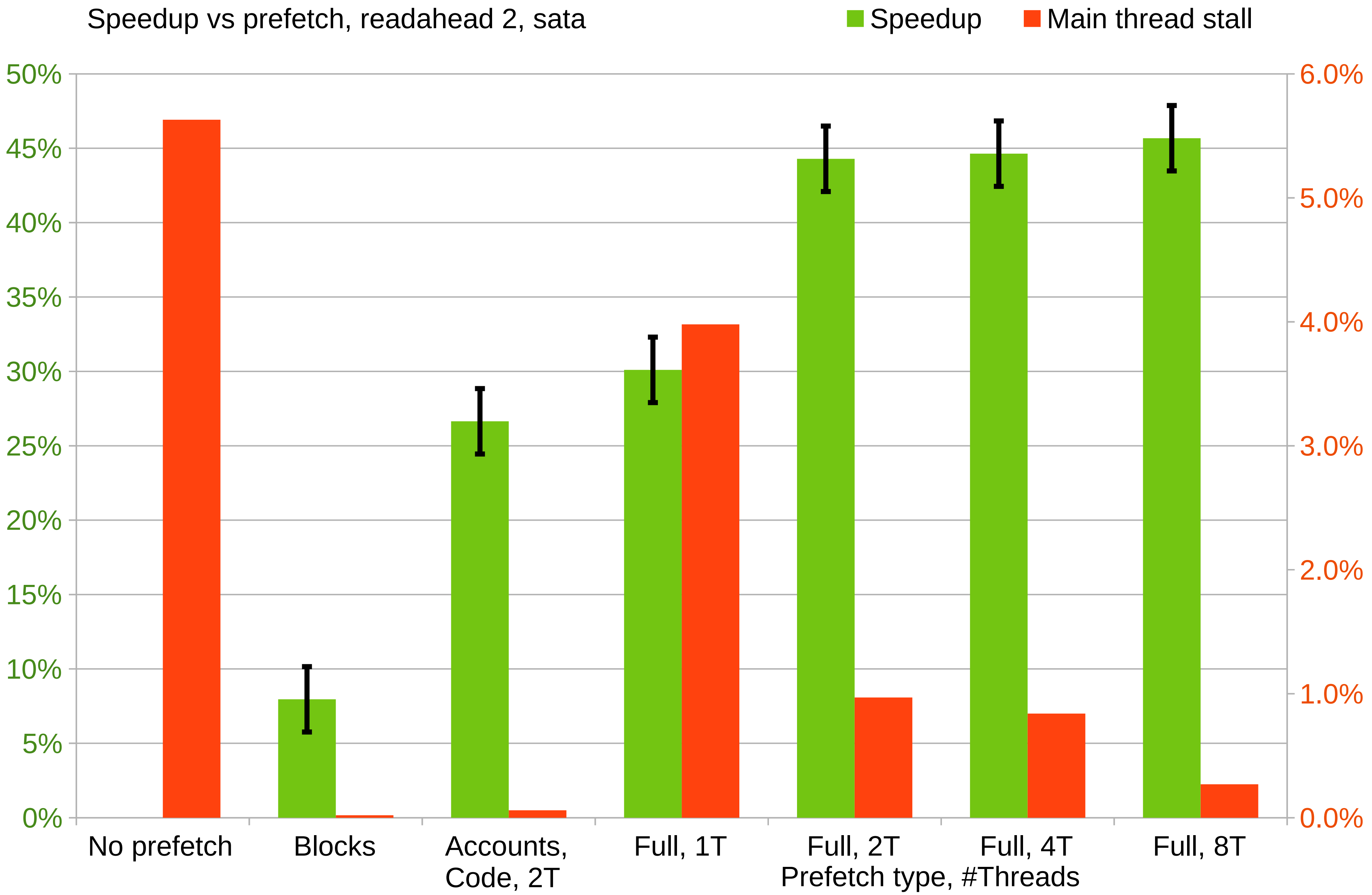
Success % vs readahead



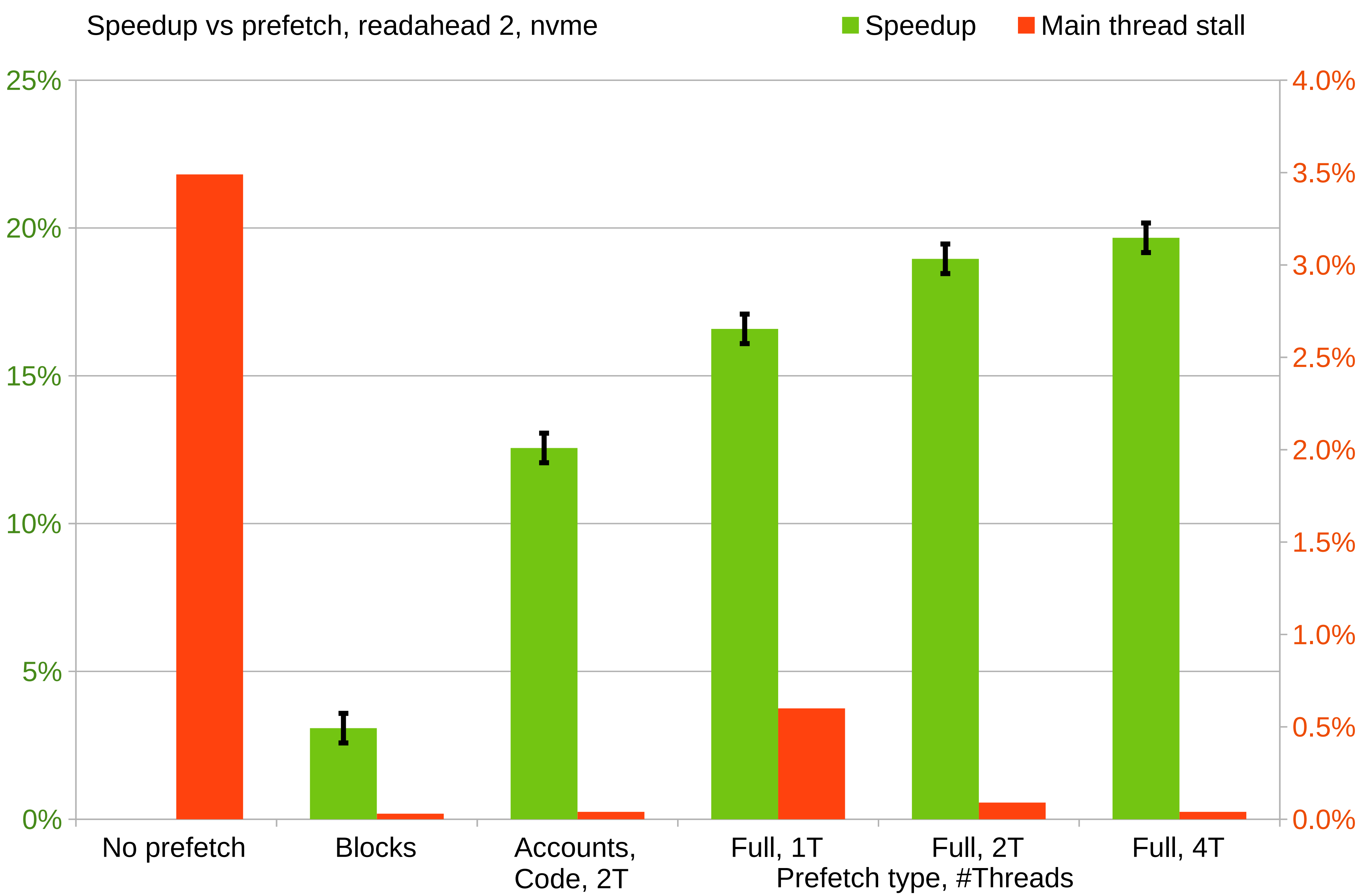
Overhead % vs readahead



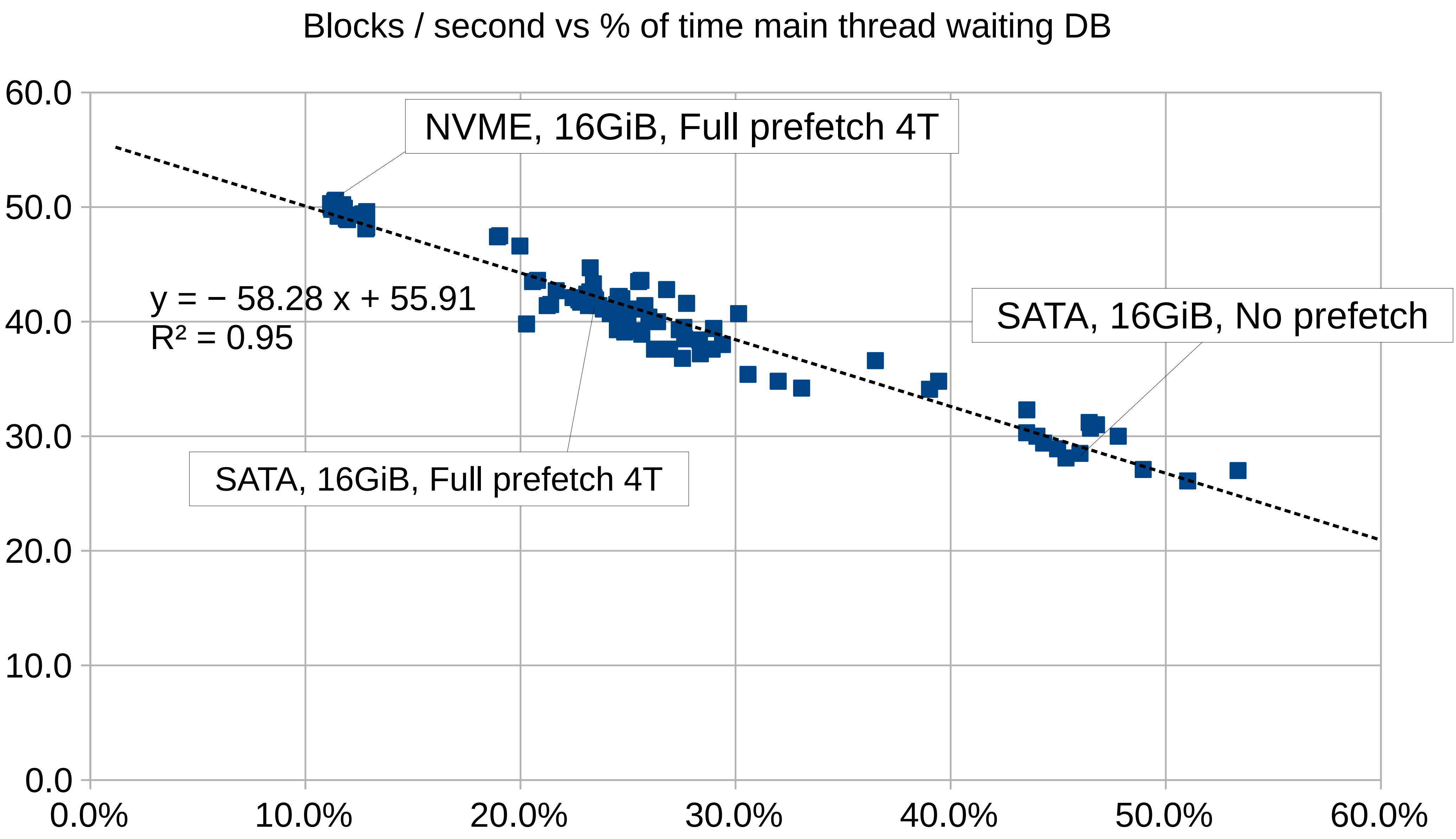
Speedup, sata



Speedup, nvme



Scatter with all test runs



Speedup vs block date

Conclusion

- Significant speedup in an already optimized client
- Close enough to the estimated maximum speedup for a perfect prefetch prediction
- Since it predicts the transactions' read/write set, it effectively determines their dependencies

→ New possibilities for future extensions,
by parallelizing the main execution