Soldøgn extra slides

Benchmarking results

How we estimate gas costs for EIP-7904 and EIP-8038?

Methodology recap

- Per-client NNLS regression on benchmark data → per-op runtime (ms)
- Glue-opcode contribution subtracted from the slope
- Worst-case across tests → worst-case across clients
- Gas = `anchor_rate × runtime_ms / 1000`
  - **Osaka anchor**: 60 M gas/s
  - **Amsterdam anchor**: 100 M gas/s → 👩‍⚖️

Preliminary EIP-7904 numbers

EIP-7904 proposed gas - Osaka @ 60Mgas/s

Opcode	Param	Current	New	Change
BLAKE2F	constant	0	48	NA
BLAKE2F	num_rounds	1	1	0.0
ECADD	constant	150	382	+1.55
ECRECOVER	constant	3000	2812	−0.06
MOD / SDIV / SMOD	constant	5	6	+0.20
MULMOD	constant	8	12	+0.50
P256VERIFY	constant	6900	15958	+1.31
POINT_EVALUATION	constant	50000	84081	+0.68

Note: `ADDMOD`, `DIV`, and `KECCAK256` (constant) have no price changes. `BLS12_G1ADD` (375 → 324), `BLS12_G2ADD` (600 → 433), and `KECCAK256` (msg_size, 6 → 1) decrease. `ECPAIRING` had no good fit on this run.

Where the worst-case is driven by one client

Opcode	Worst client	Worst gas	2nd worst	2nd gas	Ratio
BLAKE2F	reth	48	besu	17	**2.8×**
ECADD	reth	382	erigon	183	**2.1×**
P256VERIFY	reth	15958	geth	4583	**3.5×**

- Three precompiles (BLAKE2F, ECADD, P256VERIFY) are **set by reth**:
  - 2–3.5× above the next client
  - Need to investigate if this can be optimized.
- All other operations have more stable costs across clients.

Osaka — what drives each worst-case

Opcode	Worst-case test
ECRECOVER	`test_ecrecover`
POINT_EVALUATION	`test_point_evaluation_uncachable`
P256VERIFY	`test_p256verify_uncachable`
BLS12_G1ADD / BLS12_G2ADD	`test_bls12_381_uncachable`
ECADD	`test_alt_bn128_uncachable`
KECCAK256	`test_keccak_diff_mem_msg_sizes`
BLAKE2F	`test_blake2f_uncachable`
ADDMOD / MULMOD	`test_mod_arithmetic`
DIV / SDIV	`test_arithmetic`
MOD / SMOD	`test_mod`

“ Test that drives the worst case across clients. Most clients agree, with these exceptions: **BLAKE2F** — besu, erigon, geth, nethermind use `test_blake2f_benchmark`; **P256VERIFY** — besu uses `test_p256verify`; **POINT_EVALUATION** — besu and geth use `test_point_evaluation`. ”

EIP-7904 — Amsterdam vs Osaka

Amsterdam vs Osaka — besu runtimes (ms)

Opcode (param)	Osaka	Amsterdam	Osaka/Amst
ECRECOVER	0.0408	0.0111	3.7×
POINT_EVALUATION	1.2667	0.3787	3.3×
KECCAK256 (const)	0.0005	0.0001	~3.4×
ECADD	0.0001	0.0000	—
ADDMOD / MULMOD / DIV	≤ 0.0002	≤ 0.0002	~1.5–1.9×

> Amsterdam runs **~3–4× faster** on heavy precompiles, but only **~1.5–2×** on cheap arithmetic.

Amsterdam vs Osaka — geth runtimes (ms)

Opcode (param)	Osaka	Amsterdam	Osaka/Amst
ECRECOVER	0.0461	0.0127	3.6×
POINT_EVALUATION	1.3487	0.3655	3.7×
ECADD	0.0029	0.00004	**80×**
KECCAK256 (const)	0.0004	0.0001	4.4×
MULMOD / MOD	≤ 0.0002	≤ 0.0000	~3.6–3.8×

> geth speed-up is uniform **~3.5–4×** across all ops (vs besu, which is only ~1.5–2× on cheap arithmetic). `ECADD` is an outlier.

Amsterdam vs Osaka — takeaways

- BAL-optimised runs are **~3–4× faster** on compute-heavy precompiles for both besu and geth.
- For these heavy ops, the +67% anchor rate does **not** compensate → Amsterdam worst-case gas is **~0.5×** Osaka's.
- On **cheap arithmetic opcodes**, besu's speed-up is only **~1.5–2×**, so the anchor bump roughly cancels out → gas stays **flat or slightly up**.

Preliminary EIP-8038 numbers

EIP-8038 proposed gas - Osaka @ 60Mgas/s

Parameter	Current	New	Change
ACCOUNT_CODE_ACCESS	2 600	21 457	+7.3×
ACCOUNT_NOCODE_ACCESS	2 600	10 591	+3.1×
ACCOUNT_WRITE	6 700	224 268	**+32.5×**
STORAGE_ACCESS	2 200	191 667	**+86×**
STORAGE_WRITE	2 800	149 032	**+52×**

> Increases are significant! But they are measured from bloatnet and don't take BALs into account...

Slow vs fast clients per parameter

Parameter	Slow group	Fast group	Slow/Fast
ACCOUNT_CODE_ACCESS	besu 21 457	reth, geth, nethermind, erigon 271 – 5 496	**~9×**
ACCOUNT_NOCODE_ACCESS	besu, nethermind 10 366 – 10 591	geth, erigon, reth 83 – 2 531	**~11×**
ACCOUNT_WRITE [1]	erigon, reth 117 838 – 224 268	geth, nethermind 4 550 – 5 056	**~36×**
STORAGE_ACCESS	erigon, reth 184 711 – 191 667	geth, nethermind, besu 9 925 – 12 722	**~17×**
STORAGE_WRITE [1]	erigon, reth 104 522 – 149 032	geth, nethermind 3 890 – 8 389	**~21×**

> Each parameter splits clients into a slow and fast group **~10–35× apart**. The slow group is **besu (± nethermind)** on account access and **erigon + reth** on writes / storage access.

[1] besu excluded (no statistically significant fit).

Worst-case config per client × parameter

All rows use **NO_CACHE**. `ACCOUNT_*` params from `test_account_access`; `STORAGE_*` params from `test_sstore_bloated` (SSTORE).

Parameter	besu	erigon	geth	nethermind	reth
ACCOUNT_CODE_ACCESS	CALLCODE contract	STATICCALL new	CALLCODE contract	CALLCODE new	CALLCODE contract
ACCOUNT_NOCODE_ACCESS	CALLCODE new	DELEGATECALL EOA	CALLCODE new	CALLCODE EOA	CALLCODE EOA
ACCOUNT_WRITE	CALL EOA [1]	CALL new	CALL contract	CALL new	CALL EOA
STORAGE_ACCESS	SSTORE existing slot	SSTORE fresh slot	SSTORE fresh slot	SSTORE existing slot	SSTORE fresh slot
STORAGE_WRITE	SSTORE existing slot [1]	SSTORE existing slot	SSTORE existing slot	SSTORE existing slot	SSTORE existing slot

Legend: `contract` = EXISTING_CONTRACT, `EOA` = EXISTING_EOA, `new` = NON_EXISTING_ACCOUNT; `existing/fresh slot` = `existing_slots` true/false. [1] no significant fit.

Thank you