Mar 27, 2023 6 min read

EIP-2930 - Ethereum access list

Updated: 3 days ago

Introduction

An Ethereum access list transaction enables saving gas on cross-contract calls by declaring in advance which contract and storage slots will be accessed. Up to 100 gas can be saved per accessed storage slot.

The motivation for introducing this EIP was to mitigate breaking changes in EIP 2929, which raised the cost of cold storage access. EIP 2929 corrected underpriced storage access operations, which could lead to denial of service attacks. However, increasing the cold storage access cost broke some smart contracts, so EIP 2930: Optional Access Lists was introduced to mitigate this.

To unbrick these contracts, EIP 2930 was introduced, enabling the storage slots to be “pre-warmed.” It is not a coincidence that EIP 2929 and EIP 2930 are contiguous.

Authorship

This article was co-written by Jesse Raymond (LinkedIn, Twitter) a blockchain researcher at RareSkills. To support free high-quality articles like this, and to learn more advanced Ethereum development concepts, please see our Solidity Bootcamp.

How it works

An EIP-2930 transaction is carried out the same way as any other transaction, except that the cold storage cost is paid upfront with a discount, rather during the execution of the SLOAD operation.

It does not require any modifications to the Solidity code and is purely specified client-side.

The fee prepays the cold access of the storage slot so that during the actual execution, only the warm fee is paid. When the storage keys are known in advance, Ethereum node clients can pre-fetch storage values, effectively lowering the computational resource overhead.

EIP-2930 does not prevent storage access outside the access list; putting an address-storage combination in the access list is not a commitment to use it. However, the result would be prepaying the cold storage load for no purpose.

Charging less for access

Per EIP 2930, the Berlin hard fork raised the “cold” cost of account access opcodes (such as BALANCE, all CALL(s), and EXT*) to 2600 and raised the “cold” cost of state access opcode (SLOAD)

from 800 to 2100 while lowering the "warm" cost for both to 100.

However, EIP-2930 has the added benefit of lowering transaction costs due to the transaction's 200 gas discount.

As a result, instead of paying 2600 and 2100 gas for a "CALL" and "SLOAD" respectively, the transaction only requires 2400 and 1900 gas for cold access, and subsequent warm access will only cost 100 gas.

Implementing an access list transaction

In this section, we will implement an access list, compare a typical transaction to an EIP-2930 transaction, and provide some gas benchmarks.
Let's take a look at the contract that we'll be calling.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract Calculator {
    uint public x = 20;
    uint public y = 20;

    function getSum() public view returns (uint256) {
        return x + y;
    }
}

contract Caller {
    Calculator calculator;

    constructor(address _calc) {
        calculator = Calculator(_calc);
    }

    // call the getSum function in the calculator contract
    function callCalculator() public view returns (uint sum) {
        sum = calculator.getSum();
    }
}
```

We will deploy and interact with the contracts on the local hardhat node with the following script.

```
import { ethers } from "hardhat";

async function main() {
  const [user] = await ethers.getSigners();
  const data = "0xf4acc7b5"; // function selector for `callCalculator()`

  const Calculator = await ethers.getContractFactory("Calculator");
  const calculator = await Calculator.deploy();
  await calculator.deployed();

  console.log(`Calc contract deployed to ${calculator.address}`);

  const Caller = await ethers.getContractFactory("Caller");
  const caller = await Caller.deploy(calculator.address);
  await caller.deployed();

  console.log(`Caller contract deployed to ${caller.address}`);

  const tx1 = {
    from: user.address,
    to: caller.address,
    data: data,
    value: 0,
    type: 1,
    accessList: [
      {
        address: calculator.address,
        storageKeys: [
          "0x0000000000000000000000000000000000000000000000000000000000000000",
          "0x0000000000000000000000000000000000000000000000000000000000000001",
        ],
      },
    ],
  };

  const tx2 = {
    from: user.address,
    to: caller.address,
    data: data,
    value: 0,
  };

  console.log("==============  transaction with access list ==============");
  const txCall = await user.sendTransaction(tx1);

  const receipt = await txCall.wait();

  console.log(
    `gas cost for tx with access list: ${receipt.gasUsed.toString()}`
  );

  console.log("==============  transaction without access list ==============");
  const txCallNA = await user.sendTransaction(tx2);

  const receiptNA = await txCallNA.wait();

  console.log(
    `gas cost for tx without access list: ${receiptNA.gasUsed.toString()}`
  );

}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

benchmark-eip-2930.ts hosted with ❤ by GitHub view raw

The "type" section with the value of "1" right above the access list specifies that the transaction is an access list transaction.

The "accessList" is an array of objects that contain the address and storage slots the transaction will access.

The storage slots or "storageKeys" as defined in the code must be a 32 bytes value; this is why we have a lot of leading zeros there.

We have 32 bytes values for zero and one as storage keys because the "getSum" function that we call through the "Caller" contract accesses these exact storage slots in the "Calculator" contract. Specifically, x is in storage slot zero and y is in storage slot one.

Results

We get the following output

```
Compiled 1 Solidity file successfully
Calc contract deployed to 0x5FbDB2315678afecb367f032d93F642f64180aa3
Caller contract deployed to 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
==============  transaction with access list ==============
gas cost for tx with access list: 30934
==============  transaction without access list ==============
gas cost for tx without access list: 31234
```

We can see we saved 300 gas (this will be true regardless of the optimizer setting).
The call to the external contract saved 200 gas, and the two storage accesses saved 200 each, leading to a potential savings of 600. However, the warm access must still be paid, and there is a warm access for the external call and the two storage variables, each of these three operations costing 100 gas each. Thus, the net savings is 300 gas.

To be specific, the formula works in our example works out as follows:

(2600 - 2400) + 2 × (2100 - 1900) + 3 × 100 = 300.

Obtaining the storage slots of an access list transaction

The Go-Ethereum (geth) client has the “eth_createAccessList” rpc method for conveniently determining the storage slot (see the web3.js api for example).
With the RPC method, the client determines the storage slots accessed and returns the access list.

We can also use this RPC method in foundry with the cast access-list command, which uses the “eth_createAccessList” in the background and returns the access list.

Let us try an example below; we will interact with the UniswapV2 factory contract (in the Göerli network) by calling the “allPairs” function, which returns a pair contract from an array based on the index passed.

We run the following command in a forked goerli testnet.

```
cast access-list 0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f
"allPairs(uint256)" 0
```

This will return the access list of the transaction, and it would look like this in our terminal if it were successful.

```
gas used: 27983 // amount of gas used by the transaction
access-list:
- address: 0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f // address of the
uniswapv2 factory
  keys:
    0xc2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b
// slot of the pair address
    0x0000000000000000000000000000000000000000000000000000000000000003
// slot of the array length
```

Example of wasting gas with access lists

If the storage slot is calculated incorrectly, then the transaction will pay the deposit for the access list and not get any benefit for it. In the following example, we will benchmark an incorrectly calculated

ethereum access list transaction.

The following benchmark will prepay for slot 1 when it is actually slot 0 that is used.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract Wrong {
    uint256 private x = 1;

    function getX() public view returns (uint256) {
        return x;
    }
}
```

Let us put this to the test. We will call the "getX()" function using an access list with a wrong storage slot and then compare it with a transaction with a normal transaction that does not specify an access list.

This is the script to deploy and run the contract in the local hardhat node.

```javascript
import { ethers } from "hardhat";

async function main() {
  const [user] = await ethers.getSigners();
  const data = "0x5197c7aa"; // function selector for the `getX` function

  const Slot = await ethers.getContractFactory("Wrong");
  const slot = await Slot.deploy();
  await slot.deployed();

  console.log(`Slot contract deployed to ${slot.address}`);

  const badtx = {
    from: user.address,
    // to: calculator.address,
    to: slot.address,
    data: data,
    value: 0,
    type: 1,
    accessList: [
      {
        address: slot.address,
        storageKeys: [
          "0x0000000000000000000000000000000000000000000000000000000000000001", // wrong slot number
        ],
      },
    ],
  };

  const badTxResult = await user.sendTransaction(badtx);
  const badTxReceipt = await badTxResult.wait();

  console.log(
    `gas cost for incorrect  access list: ${badTxReceipt.gasUsed.toString()}`
  );

  const normaltx = {
    from: user.address,
    // to: calculator.address,
    to: slot.address,
    data: data,
    value: 0,
  };

  const normalTxResult = await user.sendTransaction(normaltx);
  const normalTxReceipt = await normalTxResult.wait();

  console.log(
    `gas cost for tx without access list: ${normalTxReceipt.gasUsed.toString()}`
  );
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

eip-2930-wrong-slot.ts hosted with ❤ by GitHub view raw

The results are as follows

```
Slot contract deployed to 0x5FbDB2315678afecb367f032d93F642f64180aa3
gas cost for incorrect  access list: 27610
gas cost for tx without access list: 23310
```

The transaction went through even though we had the wrong storage slot, however it would have been cheaper to not use an access list rather than use an incorrectly calculated one.

Don’t use access lists when storage slots are not deterministic

The implication of the previous section is that access lists should not be used when the storage slots accessed are non-deterministic.

For example, if we use a storage slot number determined based on a certain block number, the storage slot will not generally be predictable.

Another example is storage slots that depend on when the transaction occurred. Some implementations of ERC-721 push owner addresses onto an array and use the array index to identify NFT ownership. As a result, the storage slot for a token depends on the order in which users minted and that can not be predicted.

When does the access list save gas?

Making a cross contract call incurs an additional 2600 gas, which is an opportunity for savings with an access list transaction. However, there is no “added fee” for directly calling a smart contract, it is included in the 21,000 gas all transactions must pay. Therefore, access lists don’t provide any benefit for transactions that only access one smart contract.

Conclusion

EIP-2930 Ethereum access list transactions are a quick way to save up to 200 gas per storage slot when the address and storage slot of a cross contract call can be predicted. It should not be used when no cross-contract calls are made or when the address and storage slot pair is not deterministic.

Learn more

For more advanced Solidity concepts, see our Solidity Bootcamp.