---

title: "Short ABIs for Calldata Optimization" description: Optimizing smart contracts for Optimistic Rollups author: Ori Pomerantz lang: en tags: ["layer 2"] skill: intermediate published: 2022-04-01

Introduction {#introduction}

In this article, you learn about optimistic rollups, the cost of transactions on them, and how that different cost structure requires us to optimize for different things than on the Ethereum Mainnet. You also learn how to implement this optimization.

Full disclosure {#full-disclosure}

I'm a full time Optimism employee, so examples in this article will run on Optimism. However, the technique explained here should work just as well for other rollups.

Terminology {#terminology}

When discussing rollups, the term 'layer 1' (L1) is used for Mainnet, the production Ethereum network. The term 'layer 2' (L2) is used for the rollup or any other system that relies on L1 for security but does most of its processing off-chain.

How can we further reduce the cost of L2 transactions? {#how-can-we-further-reduce-the-cost-of-L2-transactions}

Optimistic rollups have to preserve a record of every historical transaction so that anybody will be able to go through them and verify that the current state is correct. The cheapest way to get data into the Ethereum Mainnet is to write it as calldata. This solution was chosen by both Optimism and Arbitrum.

Cost of L2 transactions {#cost-of-l2-transactions}

The cost of L2 transactions is composed of two components:

1. L2 processing, which is usually extremely cheap
2. L1 storage, which is tied to Mainnet gas costs

As I'm writing this, on Optimism the cost of L2 gas is 0.001Gwei. The cost of L1 gas, on the other hand, is approximately 40 gwei. You can see the current prices here.

A byte of calldata costs either 4 gas (if it is zero) or 16 gas (if it is any other value). One of the most expensive operations on the EVM is writing to storage. The maximum cost of writing a 32-byte word to storage on L2 is 22100 gas. Currently, this is 22.1 gwei. So if we can save a single zero byte of calldata, we'll be able to write about 200 bytes to storage and still come out ahead.

The ABI {#the-abi}

The vast majority of transactions access a contract from an externally-owned account. Most contracts are written in Solidity and interpret their data field per the application binary interface (ABI).

However, the ABI was designed for L1, where a byte of calldata costs approximately the same as four arithmetic operations, not L2 where a byte of calldata costs more than a thousand arithmetic operations. For example, here is an ERC-20 transfer transaction. The calldata is divided like this:

| Section | Length | Bytes | Wasted bytes | Wasted gas | Necessary bytes | Necessary gas | | ------------------- | -----: | ----: | ------: | ---------: | ----------: | ----------: | | Function selector | 4 | 0-3 | 3 | 48 | 1 | 16 | | Zeroes | 12 | 4-15 | 12 | 48 | 0 | 0 | | Destination address | 20 | 16-35 | 0 | 0 | 20 | 320 | | Amount | 32 | 36-67 | 17 | 64 | 15 | 240 | | Total | 68 | | | 160 | | 576 |

Explanation:

- **Function selector**: The contract has less than 256 functions, so we can distinguish them with a single byte. These

bytes are typically non-zero and therefore cost sixteen gas.

- **Zeroes**: These bytes are always zero because a twenty-byte address does not require a thirty-two-byte word to hold it. Bytes that hold zero cost four gas (see the yellow paper, Appendix G, p. 27, the value for $G_{txdatazero}$).
- **Amount**: If we assume that in this contract `decimals` is eighteen (the normal value) and the maximum amount of tokens we transfer will be 10^{18}, we get a maximum amount of 10^{36}. $256^{15} > 10^{36}$, so fifteen bytes are enough.

A waste of 160 gas on L1 is normally negligible. A transaction costs at least 21,000 gas, so an extra 0.8% doesn't matter. However, on L2, things are different. Almost the entire cost of the transaction is writing it to L1. In addition to the transaction calldata, there are 109 bytes of transaction header (destination address, signature, etc.). The total cost is therefore `109*16+576+160=2480`, and we are wasting about 6.5% of that.

Reducing costs when you don't control the destination {#reducing-costs-when-you-dont-control-the-destination}

Assuming that you do not have control over the destination contract, you can still use a solution similar to this one. Let's go over the relevant files.

Token.sol {#token-sol}

This is the destination contract. It is a standard ERC-20 contract, with one additional feature. This `faucet` function lets any user get some token to use. It would make a production ERC-20 contract useless, but it makes life easier when an ERC-20 exists only to facilitate testing.

```
solidity /** * @dev Gives the caller 1000 tokens to play with */ function faucet() external { _mint(msg.sender, 1000); } // function faucet
```

You can see an example of this contract being deployed here

CalldataInterpreter.sol {#calldatainterpreter-sol}

This is the contract that transactions are supposed to call with shorter calldata Let's go over it line by line.

```solidity //SPDX-License-Identifier: Unlicense pragma solidity ^0.8.0;

import { OrisUselessToken } from "./Token.sol"; ```

We need the token function to know how to call it.

```solidity contract CalldataInterpreter {

OrisUselessToken public immutable token;

```

The address of the token for which we are a proxy.

```solidity

/**
 * @dev Specify the token address
 * @param tokenAddr_ ERC-20 contract address
 */
constructor(
    address tokenAddr_
)  {
    token = OrisUselessToken(tokenAddr_);
}   // constructor

```

The token address is the only parameter we need to specify.

```
solidity function calldataVal(uint startByte, uint length) private pure returns (uint) {
```
```

Read a value from the calldata.

```solidity uint _retVal;

    require(length < 0x21,
        "calldataVal length limit is 32 bytes");

    require(length + startByte <= msg.data.length,
        "calldataVal trying to read beyond calldatasize");

```

We are going to load a single 32-byte (256-bit) word to memory and remove the bytes that aren't part of the field we want. This algorithm doesn't work for values longer than 32 bytes, and of course we can't read past the end of the calldata. On L1 it might be necessary to skip these tests to save on gas, but on L2 gas is extremely cheap, which enables whatever sanity checks we can think of.

`solidity assembly { _retVal := calldataload(startByte) }`

We could have copied the data from the call to `fallback()` (see below), but it is easier to use Yul, the assembly language of the EVM.

Here we use the CALLDATALOAD opcode to read bytes `startByte` to `startByte+31` into the stack. In general, the syntax of an opcode in Yul is `<opcode name>(<first stack value, if any>,<second stack value, if any>...)`.

```solidity

    _retVal = _retVal >> (256-length*8);

```

Only the most significant `length` bytes are part of the field, so we right-shift to get rid of the other values. This has the added advantage of moving the value to the right of the field, so it is the value itself rather than the value times $256^{\text{something}}$.

```solidity

    return _retVal;
}


fallback() external {

```

When a call to a Solidity contract does not match any of the function signatures, it calls the `fallback()` function (assuming there is one). In the case of `CalldataInterpreter`, *any* call gets here because there are no other `external` or `public` functions.

```solidity uint _func;

    _func = calldataVal(0, 1);

```

Read the first byte of the calldata, which tells us the function. There are two reasons why a function would not be available here:

1. Functions that are `pure` or `view` don't change the state and don't cost gas (when called off-chain). It makes no sense to try to reduce their gas cost.
2. Functions that rely on `msg.sender`. The value of `msg.sender` is going to be `CalldataInterpreter`'s address, not the caller.

Unfortunately, looking at the ERC-20 specifications, this leaves only one function, `transfer`. This leaves us with only two functions: `transfer` (because we can call `transferFrom`) and `faucet` (because we can transfer the tokens back to whoever called us).

```solidity

    // Call the state changing methods of token using
    // information from the calldata

    // faucet
    if (_func == 1) {

```

A call to `faucet()`, which doesn't have parameters.

`solidity token.faucet(); token.transfer(msg.sender, token.balanceOf(address(this))); }`

After we call `token.faucet()` we get tokens. However, as the proxy contract, we do not**need** tokens. The EOA (externally owned account) or contract that called us does. So we transfer all of our tokens to whoever called us.

`solidity // transfer (assume we have an allowance for it) if (_func == 2) {`

Transferring tokens requires two parameters: the destination address and the amount.

`solidity token.transferFrom( msg.sender,`

We only allow callers to transfer tokens they own

`solidity address(uint160(calldataVal(1, 20))),`

The destination address starts at byte #1 (byte #0 is the function). As an address, it is 20-bytes long.

`solidity calldataVal(21, 2)`

For this particular contract we assume that the maximum number of tokens anybody would want to transfer fits in two bytes (less than 65536).

`solidity ); }`

Overall, a transfer takes 35 bytes of calldata:

| Section | Length | Bytes | | ---------- | -----: | ----: | | Function selector | 1 | 0 | | Destination address | 32 | 1-32 | | Amount | 2 | 33-34 |

```solidity } // fallback

} // contract CalldataInterpreter ```

## test.js {#test-js}

This JavaScript unit test shows us how to use this mechanism (and how to verify it works correctly). I am going to assume you understand chai and ethers and only explain the parts that specifically apply to the contract.

```js const { expect } = require("chai");

describe("CalldataInterpreter", function () { it("Should let us use tokens", async function () { const Token = await ethers.getContractFactory("OrisUselessToken") const token = await Token.deploy() await token.deployed() console.log("Token addr:", token.address)

```
const Cdi = await ethers.getContractFactory("CalldataInterpreter")
const cdi = await Cdi.deploy(token.address)
await cdi.deployed()
console.log("CalldataInterpreter addr:", cdi.address)

const signer = await ethers.getSigner()

```

We start by deploying both contracts.

`javascript // Get tokens to play with const faucetTx = {`

We can't use the high-level functions we'd normally use (such as `token.faucet()`) to create transactions, because we do not follow the ABI. Instead, we have to build the transaction ourselves and then send it.

```
javascript to: cdi.address, data: "0x01"
```

There are two parameters we need to provide for the transaction:

1. `to`, the destination address. This is the calldata interpreter contract.
2. `data`, the calldata to send. In the case of a faucet call, the data is a single byte, `0x01`.

```javascript

}
await (await signer.sendTransaction(faucetTx)).wait()

```

We call the signer's `sendTransaction` method because we already specified the destination (`faucetTx.to`) and we need the transaction to be signed.

```
javascript // Check the faucet provides the tokens correctly expect(await
token.balanceOf(signer.address)).to.equal(1000)
```

Here we verify the balance. There is no need to save gas on `view` functions, so we just run them normally.

```
javascript // Give the CDI an allowance (approvals cannot be proxied) const approveTX = await
token.approve(cdi.address, 10000) await approveTX.wait() expect(await token.allowance(signer.address,
cdi.address)).to.equal(10000)
```

Give the calldata interpreter an allowance to be able to do transfers.

```
javascript // Transfer tokens const destAddr = "0xf5a6ead936fb47f342bb63e676479bddf26ebe1d" const transferTx = {
to: cdi.address, data: "0x02" + destAddr.slice(2, 42) + "0100", }
```

Create a transfer transaction. The first byte is "0x02", followed by the destination address, and finally the amount (0x0100, which is 256 in decimal).

```javascript await (await signer.sendTransaction(transferTx)).wait()

// Check that we have 256 tokens less
expect (await token.balanceOf(signer.address)).to.equal(1000-256)

// And that our destination got them
expect (await token.balanceOf(destAddr)).to.equal(256)
```

}) // it }) // describe ```

Example {#example}

If you want to see these files in action without running them yourself, follow these links:

1. Deployment of `OrisUselessToken` to address `0x950c753c0edbde44a74d3793db738a318e9c8ce8`.
2. Deployment of `CalldataInterpreter` to address `0x16617fea670aefe3b9051096c0eb4aeb4b3a5f55`.
3. Call to `faucet()`.
4. Call to `OrisUselessToken.approve()`. This call has to go directly to the token contract because the processing relies on `msg.sender`.
5. Call to `transfer()`.

Reducing the cost when you do control the destination contract {#reducing-the-cost-when-you-do-control-the-destination-contract}

If you do have control over the destination contract you can create functions that bypass the `msg.sender` checks because they trust the calldata interpreter. You can see an example of how this works here, in the `control-contract` branch.

If the contract were responding only to external transactions, we could get by with having just one contract. However, that

would break [composability]. It is much better to have a contract that responds to normal ERC-20 calls, and another contract that responds to transactions with short call data.

Token.sol {#token-sol-2}

In this example we can modify `Token.sol`. This lets us have a number of functions that only the proxy may call. Here are the new parts:

```solidity // The only address allowed to specify the CalldataInterpreter address address owner;

// The CalldataInterpreter address
address proxy = address(0);

```

The ERC-20 contract needs to know the identity of the authorized proxy. However, we cannot set this variable in the constructor, because we don't know the value yet. This contract is instantiated first because the proxy expects the token's address in its constructor.

```
solidity /** * @dev Calls the ERC20 constructor. */ constructor( ) ERC20("Oris useless token-2", "OUT-2") {
owner = msg.sender; }
```

The address of the creator (called `owner`) is stored here because that is the only address allowed to set the proxy.

```solidity /* * @dev set the address for the proxy (the CalldataInterpreter). * Can only be called once by the owner */ function setProxy(address _proxy) external { require(msg.sender == owner, "Can only be called by owner"); require(proxy == address(0), "Proxy is already set");

    proxy = _proxy;
}    // function setProxy

```

The proxy has privileged access, because it can bypass security checks. To make sure we can trust the proxy we only let `owner` call this function, and only once. Once `proxy` has a real value (not zero), that value cannot change, so even if the owner decides to become rogue, or the mnemonic for it is revealed, we are still safe.

```
solidity /** * @dev Some functions may only be called by the proxy. */ modifier onlyProxy {
```

This is a `modifier` function, it modifies the way other functions work.

```
solidity require(msg.sender == proxy);
```

First, verify we got called by the proxy and nobody else. If not, `revert`.

```
solidity _; }
```

If so, run the function which we modify.

```solidity /* Functions that allow the proxy to actually proxy for accounts*/

function transferProxy(address from, address to, uint256 amount)
    public virtual onlyProxy() returns (bool)
{
    _transfer(from, to, amount);
    return true;
}

function approveProxy(address from, address spender, uint256 amount)
    public virtual onlyProxy() returns (bool)
{
    _approve(from, spender, amount);
    return true;
}

function transferFromProxy(
    address spender,
    address from,
    address to,
    uint256 amount
```

```
) public virtual onlyProxy() returns (bool)
{
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

These are three operations that normally require the message to come directly from the entity transferring tokens or approving an allowance. Here we have a proxy version these operations which:

1. Is modified by `onlyProxy()` so nobody else is allowed to control them.
2. Gets the address that would normally be `msg.sender` as an extra parameter.

CalldataInterpreter.sol {#calldatainterpreter-sol-2}

The calldata interpreter is nearly identical to the one above, except that the proxied functions receive a `msg.sender` parameter and there is no need for an allowance for `transfer`.

```solidity // transfer (no need for allowance) if (_func == 2) { token.transferProxy( msg.sender, address(uint160(calldataVal(1, 20))), calldataVal(21, 2) ); }

```
    // approve
    if (_func == 3) {
        token.approveProxy(
            msg.sender,
            address(uint160(calldataVal(1, 20))),
            calldataVal(21, 2)
        );
    }

    // transferFrom
    if (_func == 4) {
        token.transferFromProxy(
            msg.sender,
            address(uint160(calldataVal(1, 20))),
            address(uint160(calldataVal(21, 20))),
            calldataVal(41, 2)
        );
    }
```

```

Test.js {#test-js-2}

There are a few changes between the previous testing code and this one.

```
js const Cdi = await ethers.getContractFactory("CalldataInterpreter") const cdi = await
Cdi.deploy(token.address) await cdi.deployed() await token.setProxy(cdi.address)
```

We need to tell the ERC-20 contract which proxy to trust

```js console.log("CalldataInterpreter addr:", cdi.address)

// Need two signers to verify allowances const signers = await ethers.getSigners() const signer = signers[0] const poorSigner = signers[1] ```

To check `approve()` and `transferFrom()` we need a second signer. We call it `poorSigner` because it does not get any of our tokens (it does need to have ETH, of course).

```
js // Transfer tokens const destAddr = "0xf5a6ead936fb47f342bb63e676479bddf26ebe1d" const transferTx = { to:
cdi.address, data: "0x02" + destAddr.slice(2, 42) + "0100", } await (await
signer.sendTransaction(transferTx)).wait()
```

Because the ERC-20 contract trusts the proxy (`cdi`), we don't need an allowance to relay transfers.

```js // approval and transferFrom const approveTx = { to: cdi.address, data: "0x03" + poorSigner.address.slice(2, 42) + "00FF", } await (await signer.sendTransaction(approveTx)).wait()

const destAddr2 = "0xE1165C689C0c3e9642cA7606F5287e708d846206"

const transferFromTx = { to: cdi.address, data: "0x04" + signer.address.slice(2, 42) + destAddr2.slice(2, 42) + "00FF", } await (await poorSigner.sendTransaction(transferFromTx)).wait()

// Check the approve / transferFrom combo was done correctly expect(await token.balanceOf(destAddr2)).to.equal(255) ```

Test the two new functions. Note that `transferFromTx` requires two address parameters: the giver of the allowance and the receiver.

### Example {#example-2}

If you want to see these files in action without running them yourself, follow these links:

1. Deployment of `OrisUselessToken-2` at address `0xb47c1f550d8af70b339970c673bbdb2594011696`.
2. Deployment of `CalldataInterpreter` at address `0x0dccfd03e3aaba2f8c4ea4008487fd0380815892`.
3. Call to `setProxy()`.
4. Call to `faucet()`.
5. Call to `transferProxy()`.
6. Call to `approveProxy()`.
7. Call to `transferFromProxy()`. Note that this call comes from a different address than the other ones, `poorSigner` instead of `signer`.

## Conclusion {#conclusion}

Both Optimism and Arbitrum are looking for ways to reduce the size of the calldata written to L1 and therefore the cost of transactions. However, as infrastructure providers looking for generic solutions, our abilities are limited. As the dapp developer, you have application-specific knowledge, which lets you optimize your calldata much better than we could in a generic solution. Hopefully, this article helps you find the ideal solution for your needs.