MultiCall: a Transaction-batching Interpreter for Ethereum

William Hughes
University of Gothenburg
hughes@chalmers.se

Alejandro Russo
Chalmers University
russo@chalmers.se

Gerardo Schneider
University of Gothenburg
gersch@chalmers.se

ABSTRACT

Smart contracts are self-executing programs running in the blockchain allowing for decentralised storage and execution without a middleman. On-chain execution is expensive, with miners charging fees for distributed execution according to a cost model defined in the protocol. In particular, transactions have a high fixed cost. In this paper we present MultiCall, an interpreter that reduces the cost of smart contract execution by emulating sequences of transactions from multiple users in one transaction. We have implemented and integrated MultiCall into Ethereum. Our evaluation shows that using MultiCall provides a saving between 56.8% and 98.9% of the fixed per-transaction cost compared to the standard approach of sending transactions individually.

CCS CONCEPTS

• **Applied computing → Digital cash**; • **Computer systems organization → Peer-to-peer architectures**; • **Security and privacy → Public key encryption**; • **Software and its engineering → Interpreters**.

ACM Reference Format:
William Hughes, Alejandro Russo, and Gerardo Schneider. 2021. MultiCall: a Transaction-batching Interpreter for Ethereum. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI '21), June 7, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3457337.3457839

1 INTRODUCTION

Distributed ledger technologies and smart contracts provide exciting new capabilities, allowing mutually distrustful parties to transact and contract without a middleman. To prevent a denial of service attack, distributed ledger protocols require a cost model which limits the total computational cost of executing transactions. In the blockchain Ethereum [15], for instance, this is implemented via a unit of account called *gas*. Gas is charged for each transaction included in a block, each byte uploaded and EVM instruction executed. The gas used per block, and the rate of block creation, is limited by the protocol —currently to around 12 million and about once every 17 seconds, respectively [8]. On-chain processing capacity is valuable, expensive and strictly limited; miners charge users via transaction fees proportional to the gas used. This cost is significant, ranging from a few cents to several dollars for the fixed transaction cost alone [8].

BSCI '21, June 7, 2021, Virtual Event, Hong Kong
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8400-1/21/06.
https://doi.org/10.1145/3457337.3457839

```
contract Token {
 ...//Other variables
 mapping(address => uint)accounts;
function xfer(address recipient, uint amount) public {
  uint senderAmount = accounts[msg.sender];
  if (senderAmount < amount)revert();
  accounts[recipient] += amount;
  accounts[msg.sender] = senderAmount - amount;
 }
 ...//Other methods }
```

Figure 1: The smart contract Token

Let us consider a subset of the smart contract Token (see Fig. 1). The contract implements a ledger of tokens which depositors can transfer to each other. While simple, it resembles real Ethereum contracts implementing the popular ERC-20 [3] interface. Balances of the Token token are stored in the `accounts` object, a mapping from Ethereum addresses to 256-bit unsigned integers. The contract supports a `xfer` method, which depositors can use to pay other recipients integral amounts of the token. The standard way for users to interact with contracts today is to send an individual transaction corresponding to a method call. Profiling on a private chain shows making a `xfer` method call in this manner costs approximately 34000 gas.[1] Of that, 21000 gas, or about two-thirds, is the fixed per-transaction cost. Consider a user that wishes to transfer many tokens to different recipients, such as the operator of an exchange or mining pool. Reducing their gas costs by even a small proportion may provide significant monetary savings. Our solution provides more than a small cost reduction; for the example above it reduces the marginal cost of a token transfer by 67.4% to 11008 gas, or the total cost by 59.6% when making 10 transfers (the solution has an overhead of 26436 gas, which can be amortised by making more transfers).

In the example above one can see that a large part of the execution cost comes from the fixed per-transaction cost. Our aim in this paper is to find a practical and systematic way to reduce smart contract gas consumption. We do so by proposing an architecture to do transaction batching, and we provide a proof-of-concept implementation for the blockchain Ethereum.

The key module of our approach is *MultiCall*, an Ethereum smart contract which reduces the gas cost of on-chain execution by emulating multiple transactions using a single one (a process known as *batching*). Batching is not novel; existing batchers emulate multiple calls from a single sender using Solidity, the most popular Ethereum smart contract programming language.

[1] 33817 to be exact, but it could cost less if the address of the recipient contains zero bytes; the addresses we used did not.

MultiCall differs from prior batchers in that it is a proper multi-instruction interpreter, and its instruction set can emulate functionality equivalent to an arbitrary block of transactions in a single transaction.

Our interpreter is quite efficient: the volatile memory accesses, arithmetic and branching required for interpretation are much cheaper in the Ethereum cost model than accessing the ledger state or verifying signatures. The overhead of interpretation when batching transactions is therefore manageable.

More concretely, the main contributions of this paper are:

i) An architecture to improve gas consumption based on a middleware (MultiCall and its off-chain API code) that can emulate arbitrary sequences of transactions. (Section 3)
ii) A proof-of-concept implementation of the MultiCall Ethereum smart contract. (Section 4)
iii) An evaluation of our approach to show its feasibility and advantages. Our evaluation of MultiCall's performance relative to unbatched transactions shows a saving of between 57% and 99% of the fixed per-transaction cost compared to sending individual transactions. We also compare MultiCall's performance to an existing batcher, with favourable results. (Section 5)

In section 2 we present some preliminaries on Ethereum and Solidity. In section 3 and 4, we describe the design decisions underlying the MultiCall smart contract and its implementation details respectively. In section 5, we evaluate the contract's performance; security issues and avenues for future work are discussed in section 6. Related work and our conclusion are presented in section 7 and 8.

2 BACKGROUND

The fundamental purpose of the Ethereum ecosystem is to enable parties to transact: to enter into contracts, to interact with those contracts (for example by making choices or executing clauses) and make payments. The contracts entered into using distributed ledger technology are computerised and self-enforcing; such contracts are termed *smart contracts*.[2]

Like many computer systems, the Ethereum ecosystem can be thought of as implemented in multiple layers of abstraction (see Table 1). The highest is what might be called the abstract layer, which consists of payments and contracts that the parties wish to make. It is implementation-agnostic so it could for instance be implemented as a centralised ledger or using scalable decentralised solutions such as state channels [14].

The most popular means of implementing abstract contracts on the Ethereum platform is using the smart contract programming language Solidity [4]. Solidity is a statically typed object-oriented language which lets the user write Ethereum smart contracts as objects which expose methods and contain persistent state. Abstract arrangements, such as an escrow agreement or a new issue of tokens, then correspond to one or more Solidity contract objects (or state within such objects). Users offer new arrangements to counterparties by creating Solidity contracts, and interact with those contracts using method calls. Ether payments are treated as a special `.transfer` method.

The Solidity layer is in turn translated into primitive Ethereum transactions. Transactions are an indivisible unit of interaction with the blockchain; each block contains a sequence of transactions. There are two Ethereum transaction types, `create` and `call`. Solidity constructor calls (which instantiate a new contract) are translated into `create` transactions, and method calls are translated into `call` transactions. When appended to the blockchain, transactions modify the distributed ledger by performing a call or create action respectively from the signatory key's account.

There are two types of account on the Ethereum ledger: *externally owned accounts* (EOAs) and *smart contracts*. EOAs are controlled by an Ethereum private key, and their address is the corresponding 160-bit public key. Transactions submitted to the blockchain effect calls or creates from an EOA. Calling an EOA transfers the Ether value specified in the call from the caller to the callee account. Smart contract accounts contain additional state: immutable bytecode and a mutable persistent storage space. When a smart contract is called, in addition to optionally effecting an Ether payment, the contract's bytecode is interpreted by the Ethereum Virtual Machine (EVM). The EVM is a Turing-complete stack machine with instructions specialised for the blockchain environment. In particular, the EVM supports instructions for querying the bytestring *calldata* sent in the call, querying the address of the caller, and performing calls and creates with equivalent effect to transactions. Solidity method identifiers and arguments are encoded as calldata, the `msg.sender` expression is compiled to the EVM `CALLER` instruction; method and constructor calls in the text of a contract compile to EVM `CALL` and `CREATE` instructions.

Payments in the native cryptocurrency Ether are a special case of call transactions with an empty calldata. Ether payments may be translated directly from the abstract layer to transactions, or may be viewed as the special `.transfer` Solidity method call. Payments in user-issued tokens such as those managed by ERC-20 contracts are translated to method calls to that contract.

Let us see how the above works through an example. When Alice wishes to perform the abstract action of giving Bob some `Token` tokens (see Fig.1), they specify an Ethereum public key `BobAddr` controlled by Bob and instruct their Ethereum client software to make a payment of `Token` to that address. The action is translated in their client software to a Solidity method call to the token contract's address `TokenAddr` and then to an Ethereum call transaction, signed by some private key controlled by Alice whose corresponding public key is `AliceAddr`. Finally the transaction is broadcast to the Ethereum network and ultimately mined and included in the blockchain. That modifies the distributed Ethereum ledger state, effecting a call action from the EOA at address `AliceAddr` to the smart contract at address `TokenAddr`. This process is illustrated in Fig. 2.

We are interested in reducing the gas cost of execution on the Ethereum blockchain. To optimise execution on the Ethereum ledger, one must first understand its capabilities and cost model. A detailed description of Ethereum's cost model and semantics can be found in the latest version of Ethereum Yellow Paper [15]. We will not present Ethereum's cost model in detail but rather what is

[2]The term refers both to the entirety of self-enforcing arrangements, including off-chain components, and to individual on-chain program objects on the blockchain. The relation between smart contracts in the general and narrow sense is analogous to that between programs consisting of multiple OS processes and each individual OS process, which is also a program.

Table 1: Ethereum's layers of abstraction

Layer	Identities	Payments	Contract initiation	Contract execution
Abstract	Users	Payment	Offer to counterparties	Choice, complaint etc
Solidity	Addresses	.transfer method	Constructor call	Method call
Primitive	Addresses	Call txn/instruction	Create txn/instruction	Call txn/instruction

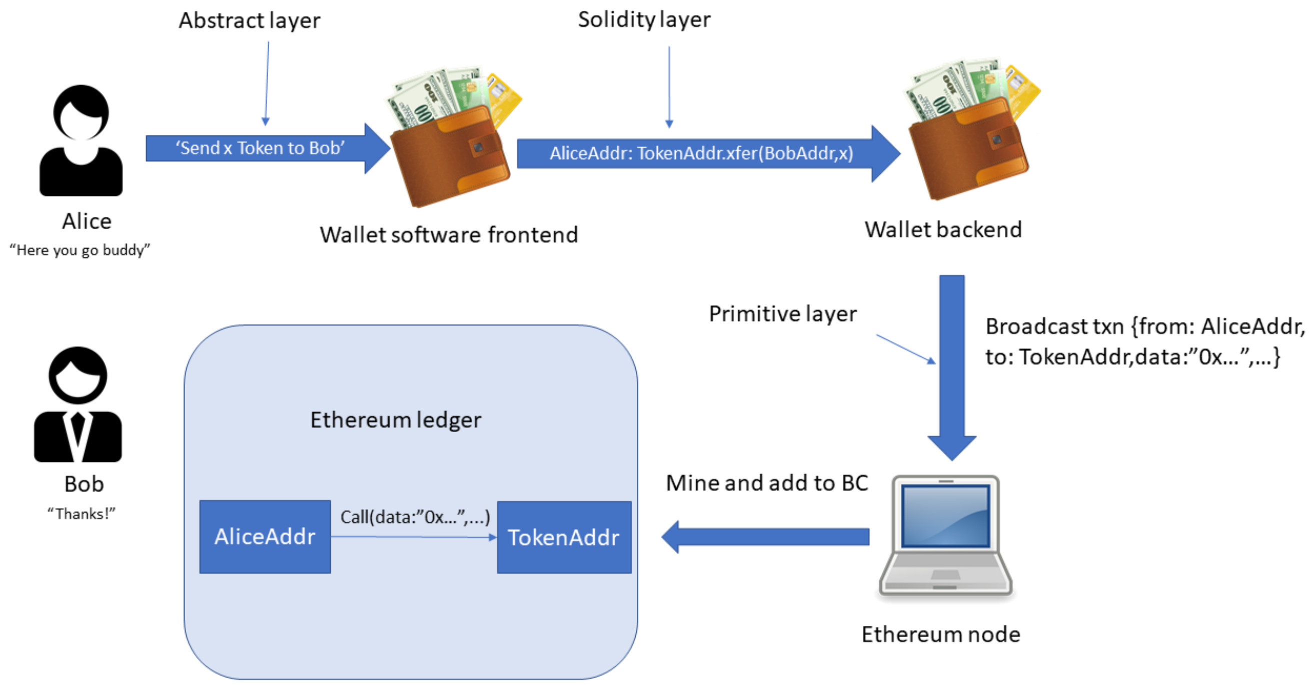

Figure 2: An illustrated payment of a user-issued token.

relevant to understand how MultiCall works, namely the following key facts:

(1) Call transactions cost 21000 gas, not including the cost of contract execution.
(2) Call instructions executed by a contract (which have a practically equivalent effect) cost 700 or 7400 gas depending on whether they involve an Ether payment, not including the cost of contract execution.
(3) Create transactions cost 53000 gas: the fixed transaction cost and an additional creation cost of 32000 gas.
(4) Create instructions executed by a contract (which have a practically equivalent effect) cost only 32000 gas.
(5) Compared to instructions which modify the persistent ledger state (such as calls and storage writes), the arithmetic and control flow EVM instructions needed for interpretation is very cheap.
(6) The cost of uploading data (such as scripts) is also relatively low.

3 MULTICALL DESIGN

Our solution provides value by batching transactions. To do so we need to modify the Ethereum transaction submission workflow in the wallets and on the Ethereum ledger. Instead of converting an action into a transaction and sending it immediately, it is converted to a MultiCall instruction and saved. Multiple instructions can be concatenated and signed to create a *metatransaction* [10]. A metatransaction is data which is not a valid Ethereum transaction but is signed by an Ethereum private key, then verified and executed in a smart contract. The MultiCall metatransactions can then be sent to a batching server, which sends their concatenation to MultiCall in a single transaction. MultiCall acts "under the hood" so the user need not be aware of it. As shown in Fig. 3, MultiCall and its associated off-chain code works as middleware between the wallet frontend and the backend, and adds some additional machinery to aggregate instructions before delivering them to Ethereum nodes.

One of the key features of MultiCall is the ability to batch transactions from multiple users in a single call, while providing each user a unique identity. Consider the Token example again: each user should be able to transfer tokens from their account, and only their

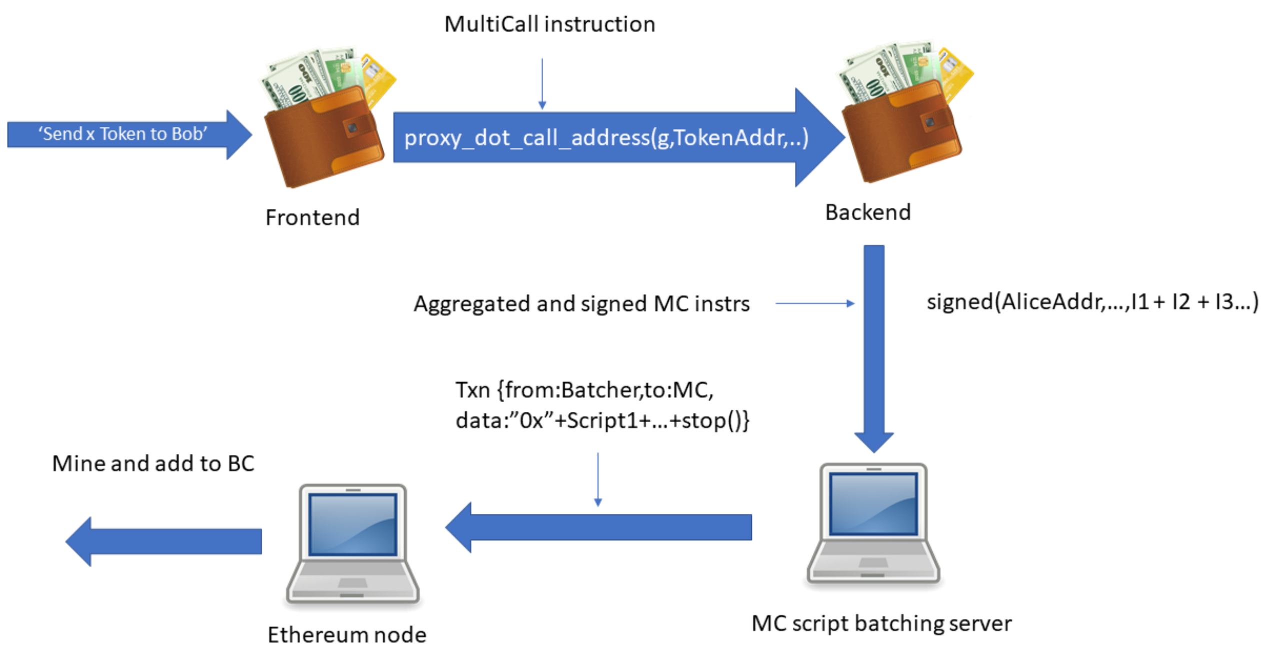
MultiCall instruction
'Send x Token to Bob'
proxy_dot_call_address(g,TokenAddr,..)
Frontend
Backend
Aggregated and signed MC instrs
signed(AliceAddr,...,I1 + I2 + I3...)
Txn {from:Batcher,to:MC, data:"0x"+Script1+...+stop()}
Mine and add to BC
Ethereum node
MC script batching server

Figure 3: An illustration of the modified workflow using MultiCall

account. This is done by querying the caller in the `xfer` method, `msg.sender` (see Fig. 1) and deducting from their account. Using the caller address to authenticate and identify users is a common smart contract pattern. Because smart contracts often deal with the transfer and use of assets, method calls which require user authentication are a key part of smart contract functionality. MultiCall would therefore not be very useful if it did not provide each user a unique identity; it does so through *proxies*. Proxies are a well-known Ethereum development pattern: puppet contracts which perform call and create actions when commanded to do so by their controller. That may be achieved by checking the address of the proxy's caller, or by using metatransactions. In the case of MultiCall, its proxies check that it is the caller. Each proxy belongs to a single MultiCall user (represented as an Ethereum address); MultiCall will only command a proxy to create a contract or call an address when executing on behalf of the address that owns it (we explain how that works in more detail later).

The MultiCall contract is written in a low-level EVM code-generating DSL rather than Solidity. That made it easier to achieve an efficient data layout for instruction arguments as we avoided the inefficient abstractions and calling convention of Solidity.

Instruction set. MultiCall is a smart contract implementing a specialised interpreter which, when called, interprets its bytestring argument as a sequence (or *script*) of MultiCall instructions and executes each sequentially. Its instruction set is designed purely for transaction batching, emulating the functionality of several transactions in one. Unlike interpreters such as the JVM or EVM, MultiCall is deliberately not Turing-complete in order to ease design and future formal verification. It has eight principal instructions of interest to the user (described below), not counting variants and admin-only instructions. MultiCall executes on behalf of a single Ethereum address at a time, charging any Ether costs incurred when executing instructions to the user's account.

(1) `call_address(gas,address,eth_value,data)` performs an EVM `CALL` with the given arguments directly from MultiCall; it is useful for Ether payments to users and method calls which do not require authentication.
(2) `proxy_dot_call_address(gas,address,eth_value,data)` instructs the user's proxy to perform the call instead.
(3) `create(eth_value,data)` creates a contract directly from MultiCall.
(4) `proxy_dot_create(eth_value,data)` creates a contract from the user's proxy.
(5) `deposit_address(address,eth_value)` credits the given address' account in MultiCall's persistent storage; it is a cheaper way of paying a MultiCall user Ether than making a call.
(6) `createProxy(eth_value)` creates a new proxy with the given Ether endowment.
(7) `signed(sig,len,deadline,eth_value,nonce,script)` checks the given signature against the other arguments, then executes the given script on behalf of the signatory if the signature is valid. The given `eth_value` is paid to the caller

to motivate them to include the script. If the time is greater than the deadline then the script becomes invalid; the EVM provides an instruction `TIMESTAMP` for querying the time.
(8) `stop()` terminates a signed script, or exits the interpreter if MultiCall is not executing a signed script.

Intuitively, every Ethereum transaction conceived to create or call a contract can be mapped into one of MultiCall's instructions `create`, `proxy_dot_create` (if creator authentication is required), or `call_address`, `proxy_dot_call_address`, respectively. Furthermore, Ether payments may either be translated to call instructions or a `deposit_address` instruction. Multiple instructions may be concatenated into scripts before being signed and sent to a batching server. The batching server in turn collects multiple signed scripts and then broadcasts a call transaction (signed by its own key) which calls the MultiCall interpreter smart contract with the signed scripts as calldata. When mined, this transaction effects a call to MultiCall, which then performs the actions from the users, such as making method calls and payments. Like `Token`, MultiCall contains a mapping from Ethereum addresses to accounts in its persistent state; Ether payments performed on behalf of a user are deducted from the user's account.

Revisited example. Suppose Alice wishes to make payments of `Amount1` to `Amount10` respectively of the token tracked in the `Token` contract to 10 different recipients, `Bob1` to `Bob10`. Each payment is translated into a MultiCall instruction

```
proxy_dot_call(G,TokenAddr,0,C(BobN,AmountN))
```

where `G` is some reasonable gas limit chosen by Alice or the wallet and `C(BobN,AmountN)` is the calldata corresponding to the method `.xfer(BobN,AmountN)`. `BobN` and `AmountN` refer to one of the recipient addresses and the amount of tokens to pay them respectively. Note that a proxy call is appropriate because `Token`'s method `.xfer` requires authentication to spend from the caller's account. The wallet could at this point wait for more instructions to be added before signing them, but let's suppose it does not. It then selects an appropriate deadline and tip. A reasonable choice would be the current time plus one hour, and the cost of the given instructions at the current gas price. The wallet then signs the instructions and associated deadline `D` and tip `T`, and then constructs an instruction `signed(Sig,Script.length,D,T,Script)`. The expressions `Sig` and `Script` are the signature and the concatenated instructions respectively.

Another choice would be for Alice to send the instructions to MultiCall herself, without going through a batching server. In that case, profiling shows it costs 136516 gas, as opposed to 338170 to send the payments individually.[3] Considering the fixed cost of the transaction to send the script, 59.6% of the fixed transaction cost was eliminated relative to sending 10 transactions individually. Blocks may contain over 100 transactions (and when using MultiCall you could effectively make many more within the block gas limit), so fixed overheads would be negligible in practice if using a batching server.

Deployment. To use the MultiCall smart contract, an instance of it must first be uploaded to the blockchain with a create transaction. For each individual user to gain access to the full functionality of MultiCall, they must allocate an account and create a proxy. That is achieved by depositing Ether to MultiCall with a call transaction or receiving a deposit from another user, and by running the `createProxy` instruction respectively.

[3]The cost may vary slightly due to the number of zero bytes in addresses; a nonzero byte in calldata costs 16 gas, while a zero byte costs only 4.

4 IMPLEMENTATION

The implementation of the interpreter consists of five main components:[4]

- A *volatile state*;
- A *table of registered users* with their corresponding Ether balances and metatransaction nonces;
- A *jump table* of MultiCall instructions;
- *interpreter initialisation code* which sets the volatile state on entry, and
- an *instruction set* which defines the available instructions.

We briefly discuss each of these below.

Volatile state. When called, the MultiCall smart contract uses the following volatile state:

i) The program counter `pc` is used to track the next MultiCall instruction to be executed from the calldata; `pc` is stored on the stack, the rest in memory.
ii) The variable `balance` caches the credit of the current user, deducted for instructions which spend Ether such as calls, creates and deposits.
iii) Because MultiCall may batch transactions from many users in a single call, a mutable variable is required to track this: `signatory` tracks the Ethereum public key of the user on whose behalf MultiCall is currently executing.
iv) The variable `nonce` is used to protect from replay attacks when executing signed scripts: it is 0 when MultiCall is not executing a signed script.
v) The variable `stashedBalance` is used to remember the balance of the caller when it is required to execute a signed script, and MultiCall needs to temporarily change on whose behalf it is executing.

Registered users table. Each user (identified by an Ethereum address) has an entry in the table of registered users, which records a 48-bit balance of Ether denominated in gwei (a billion times the smallest unit of Ether, the wei) and a 16-bit nonce for protection from metatransaction replay attacks. This is implemented by an array of account structs in persistent storage.

Jump table. MultiCall's jump table consists of an array of EVM code entries, each of which is a JUMPDEST instruction marking a valid jump destination followed by a jump to a constant address (the address of the instruction code). The interpreter's one-byte opcodes are used as byte offsets into the table; the dispatch code simply jumps into the table using the opcode as a byte offset. The jump table is therefore only 256 bytes long, and because each entry is 5 bytes (one byte for JUMPDEST, 4 for a constant jump) it can fit at most 52 instructions. Thankfully that is sufficient for MultiCall's functionality, but future interpreters may require a different

[4]The source code can be made available via the PC chairs.

dispatch scheme. MultiCall's dispatching mechanism is efficient enough for our purposes, costing only 41 gas as compared to 3 gas for an add or push EVM instruction. Its cost is negligible compared to the cost of executing transaction-emulating instructions, as shown in Section 5.

Initialization code. When MultiCall is invoked, MultiCall sets the in-memory variables `signatory` (the address on whose behalf MultiCall is executing) to the caller, `balance` to the number of gwei deposited by the caller, and the stack variable `pc` (the program counter) to `0`. It then dispatches, entering the first instruction.

Instruction set. For efficiency there is no separate interpreter loop; each instruction dispatches to the next. Each MultiCall instruction consists of a contiguous block of EVM bytecode. MultiCall instructions modify the state of the interpreter and perform some side effects (such as internal state changes or EVM calls), until an instruction throws an exception or a `stop` instruction exits the interpreter.

Making payments. Paying instructions such as calls, creates and deposits perform a side effect which may cost Ether: performing an EVM call, creating a contract, and crediting another user's account respectively. Such instructions deduct the payment from the volatile `balance` variable, rather than directly from the account of the `signatory` on whose behalf MultiCall is executing. That saves gas since persistent storage writes are significantly more expensive than memory writes.

Stopping execution. The volatile balance is settled against the signatory's account in the `stop` instruction, which ends a signed script, or exits the interpreter if MultiCall is not in a signed script. Whether MultiCall is executing a signed script is detected by checking whether the variable `nonce` is `0`. It may seem unsafe to check whether the user can afford to make payments after the payments have been made. However, the EVM reverts the side effects of a call if it throws an exception. By throwing an exception when the user's account balance is insufficient, incorrect behaviour is prevented. The `stop` instruction is the only means of ending a call to MultiCall; if the script passed in the calldata does not contain a `stop`, then the interpreter will loop until it runs out of gas and throws an exception, reverting any desirable side effects.

Proxies. Each user controls a proxy contract, whose address can be computed from the user's address. Proxies are allocated with the `createProxy` instruction, which creates a proxy using the EVM instruction `CREATE2`. The instruction behaves like `CREATE`, except that the address of the created contract is deterministically computed from the creator address (in this case, the address of MultiCall), the initialization code and a salt. The salt used is the address of the signatory. That eliminates the need to store the proxy's address in the user's account, as the address can be recomputed when needed by the `proxy_dot_create` and `proxy_dot_call` MultiCall instructions.

All proxies created by the same instance of MultiCall have the same bytecode, which checks whether the caller is its creator (MultiCall) and the calldata is ended by the magic 32-bit number indicating a proxy call or create command. If that is the case, the proxy performs the commanded call or create.

The trivial way for a proxy to store its creator's address would be to place it in persistent storage and read it on each call, but that would be inefficient as storage reads are expensive. Instead, during contract creation the creator's address (which can then be compared to the `CALLER`) is written into a push instruction in the in-memory bytestring which is returned as the final code of the proxy. The end result is that fetching MultiCall's address in order to compare the caller's to it costs 3 gas, as opposed to 803.

Metatransactions. MultiCall metatransactions consist of a `signed` instruction, containing a number of MultiCall instructions terminated by a `stop` instruction. In section 3 we state that the signed instruction checks its signature immediate argument against the hash of its other arguments and begins executing "on behalf of" the signatory if the signature is valid. Signature verification is achieved by calling the primitive contract `ECRECOVER`, which recovers the public key of the signatory given a hash and an Ethereum signature of the hash. What executing "on behalf of" a public key means is that the `signatory` variable is set to that public key. The value of `signatory` is restored to the caller upon the next `stop` instruction. The balance of the caller is also saved to the `stashedBalance` variable, and the signatory executes with a new balance. The tip is deducted from the new balance and credited to the stashed balance. When signed execution stops (at the next `stop` instruction) then the signatory's balance is settled, the stashed balance is restored, and MultiCall reverts to executing on behalf of the caller. Since there is only one `stashedBalance` variable rather than a stack of balances and addresses, then nested metatransactions are disallowed.

Note that if the metatransaction is terminated with a stop prematurely, it is the caller that pays for subsequent instructions. If it is not terminated with a stop at all, any caller is free to append their own instructions when running the signed script, enabling the theft of any Ether in the account and any ERC-20 tokens controlled by the account's proxy contract.

An attractive feature of metatransaction scripts is that multiple user actions can be authorized with a single signature verification.

5 PERFORMANCE EVALUATION

We will showcase the cost savings provided by MultiCall with micro-benchmarks as well as by revisiting the example `Token` from Section 1, where MultiCall saves 59.6% of the gas required for token-transfer costs. We also compare the performance of MultiCall with a preexisting batcher, MultiSend.

5.1 Micro-benchmarks

To evaluate the gas cost savings provided by MultiCall, we ran a number of MultiCall instructions in sequence and measured their marginal cost. We uploaded MultiCall to a private test chain run using *ganache-cli* v6.10.2 [2] and tested it using *truffle* v5.1.43 [1], a development framework for Ethereum which can provide an interactive JavaScript console to the private chain. The savings provided by using transaction-emulating instructions relative to transactions as a proportion of the fixed transaction are shown in Table 2. To clarify: if an instruction which emulates a call transaction has a cost of X, its savings are reported as $(21000 - X)/21000$, rounded to the nearest tenth of a percentage point. To calculate the savings of instructions which emulate create transactions, we

Table 2: MultiCall gas costs and savings vs unbatched transactions

MultiCall action	Gas cost	Savings
Ether-paying call	8062	61.6%
Non-paying call	1360	93.5%
Ether-paying proxy call	9064	56.8%
Non-paying proxy call	2352	88.8%
Create	32233	98.9%
Proxy create	33289	93.9%
Deposit	6392	69.6%

first deduct the 32000 gas cost which is paid either way and then perform the above calculation.

The calldata used for calls and creates were empty; the contracts called were dummies which return immediately upon being called, in order to eliminate any gas cost confounding from contract execution. There is a negligible overhead of approximately 3 gas per word for copying calldata into memory when batching, rather than sending transactions directly. The fact that MultiCall create instructions cost more than the fixed transaction cost of 21000 gas and yet still provide savings may be surprising; note that the 32000 contract creation cost is paid by create transactions as well. Create transactions have a minimum gas cost of 53000, which includes both the 32000 gas cost of creation and the 21000 fixed transaction cost. The proportional cost saving of batching is computed as a fraction of the 21000 gas fixed transaction cost saved, not the 53000 gas cost which includes the create. We conclude that batching via MultiCall provides significant cost savings.

5.2 Metatransactions

MultiCall metatransactions use the `signed` instruction, which takes a script of MultiCall instructions and a signature of it (and some additional arguments) and executes the script on the signatory's behalf. The `signed` instruction can be used to allow multiple signatories to share a single call transaction for their batching. However, `signed` has some overhead - incrementing the signatory's nonce requires a storage write, and signature verification is an expensive operation. If the instruction cost more than 21000 gas, it would of course not be of any use - then users might as well send transactions separately.

To evaluate the `signed` instruction, it was profiled by sending varying numbers of metatransactions with empty scripts (containing only a stop instruction). Profiling is somewhat more complicated than for other instructions. Different signatories must be used for each `signed` in the call to avoid amortisation of the write to the signatory's account; repeated storage writes to the same index cost less in the gas cost model. Also, each time a new signed script is run for the same account, it must have a different nonce. For that reason, `signed` was profiled separately from the other instructions.

In short, the gas cost of each `signed` varies slightly due to variation in the number of zero bytes in the signature (which cost 12 gas less to upload in transaction calldata than nonzero bytes), but never exceeds 12000 gas. The end user may assume they save at least 9000 gas using a metatransaction to share a call compared to calling MultiCall themselves. A noteworthy implication is that it's cheaper to send a MultiCall metatransaction which contains only a single Ether transfer than use an unbatched transaction.

```
xfer = (to) => api.iset.proxy_dot_call(20480,T.address,0,
     "0x" + util.abi_method("xfer(address,uint256)")
        + util.abi_address(to) + api.abi_uint(1));
 web3.eth.sendTransaction({
 from: sender,
 to: MultiCall,
data: "0x" + xfer(recipient1) + ... + xfer(recipientN) +
     api.iset.stop(),
 gas: 1000000
 })
```

Figure 4: Batching contract calls using MultiCall

costs feb 15.png

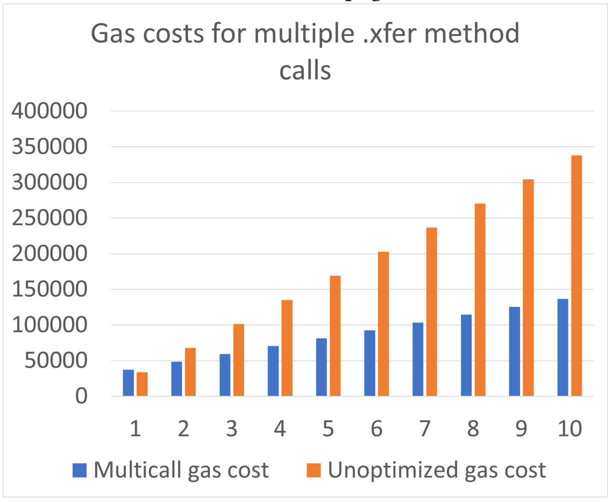

Figure 5: The gas cost of 1-10 `.xfer` method calls to the contract Token shown in Section 1, compared to the cost of unbatched EVM method calls.

5.3 Token transfers

Consider once more the example of Alice making one token payment each to `N` recipients, using the method `.xfer(address,uint)` of the Solidity contract `Token`—recall Section 1. The gas cost of doing so in one transaction via MultiCall compared to sending `N` transactions individually is shown in Figure 5, for `N` ranging from 1 to 10. We can see in the figure that the savings are significant, i.e. up to 59.6% for 10 payments, and the proportional savings improve as the fixed overhead of MultiCall is amortised.

On the one hand, we launch `N` transfers with truffle by simply making `N` standalone JavaScript method calls `T.xfer(to,amt)`. Such calls are then translated into call transactions to the address `T.address`, with calldata containing the 32-bit method identifier for `xfer` followed by the 160-bit address argument `to` (left-padded to 32 bytes) and 256-bit token amount `amt`. When making `N` unbatched Solidity method calls, each transaction executes independently and costs the same amount of gas: 33817. The total cost is therefore `N*33817`.

In contrast, when making `N` payments using MultiCall, only a single call transaction to MultiCall is sent; the calldata consists of the concatenation of `N` `proxy_dot_call` instructions. Each proxy call instruction calls the user's proxy, which in turn calls the `Token` contract instance `T.address`, with the same calldata as used in a standalone `.xfer` transaction. The JavaScript code for making the call to MultiCall from truffle is shown in Fig. 4. The cost of making a single payment via MultiCall is 37444. After that the marginal cost of making an additional payment is always the same: 11008 gas! That's a 67.4% saving compared to sending token payment transactions individually. The shared fixed cost for the sequence of payments is 26436 gas, corresponding approximately to the fixed transaction cost and the cost of a storage write. Since batched `Token` payments are implemented using a proxy call which *does not make an Ether payment*, one would expect from the micro-benchmarking that approximately 2352 of the 11008 gas cost is the overhead of MultiCall, and the remaining 8656 is the `.xfer` method's execution cost—that is surprisingly low, since the method performs two storage increments, which typically cost 5800 gas each. However, repeated writes to the same storage index are cheaper in the cost model. We believe that since the proxy's balance is deducted repeatedly, the marginal cost of updating it is reduced. The ability to amortise the cost of storage writes made during contract execution makes batching transactions even more attractive than the micro-benchmarks would indicate.

5.4 Setup cost

Using MultiCall requires some initial on-chain setup work, which costs gas. There are four tasks which must be completed before an individual user can access the full features of the interpreter:

(1) An instance of MultiCall must be present on the Ethereum ledger.
(2) A user account must be allocated.
(3) A user proxy contract must be allocated.
(4) A call to MultiCall must be executed.

Uploading MultiCall is expensive, costing 1,751,894 gas.[5] However, the creation cost can be amortised over all users. Allocating an account can be done by depositing Ether as part of a call to MultiCall; the marginal cost of allocating the account is 22158. That is an upper bound on the cost; it can also be allocated by another party making a deposit to it, which costs only 21393 gas. Creating a proxy with the `createProxy` instruction costs 61358 gas. In total a user must pay 82749 gas to set up their account. Then follows the fixed overhead per invocation of MultiCall, which is up to 28158 gas assuming the caller spends ether from MultiCall, or 22215 if they do not. Benchmarking shows that at least 10000 gas can be saved per marginal batched transaction. Assuming pessimistically that users don't use signed scripts to amortise MultiCall's fixed call overhead, then if each user batches 5 transactions at a time, they will recoup the setup cost for their account after 25 batched transactions. If they batch a large number of transactions, they may recoup the cost after only 12 batched transactions. If each user makes 30 transactions in batches of 5, approximately 85 users and 2550 transactions in total would be required to recoup the cost of deploying MultiCall in terms of gas. Since Ethereum has many more than 85 users and hundreds of thousands of transactions are sent per day, that is not an insurmountable obstacle.

That shows that deploying at least one batcher on the blockchain is worthwhile considering the setup cost, but not that MultiCall itself is profitable to deploy - one must then compare the deployment cost to the cost of using existing batchers instead (in both gas and usage fees). That is more complex to estimate considering their different features, and is beyond the scope of this paper. In practice it would also be advisable to iterate further on the contract, adding features and optimisations to deploy once and for all; continuous deployment is costly on Ethereum. However, that does not diminish MultiCall's value as a research prototype.

5.5 MultiCall vs. MultiSend

In this section we will show that MultiCall is more performant than MultiSend [6], a popular batcher implemented in Solidity assembly with the purpose of reducing gas costs.

Before going into the details of our comparison, we summarise the main result. For the only comparable feature (call batching), MultiSend costs 200 more gas per batched call. Since the upload cost of each MultiCall call instruction is about 200 gas lower than its MultiSend equivalent, we speculate that the difference lies in MultiCall's highly optimised instruction argument packing and parsing, generated by a DSL combinator. Access to combinators which can be reused to generate efficient bytecode is an advantage of the DSL over Solidity assembly.

As explained earlier, transaction batching is an established technique. To evaluate MultiCall's performance, it is of interest not only to compare its overhead relative to the fixed transaction cost, but to compare it to an existing batcher. We have chosen to compare to the popular batcher MultiSend, because it is written in Solidity inline assembly to maximise performance. One might expect that would allow improved performance, especially since it allows one to avoid Solidity's space-inefficient ABI calling convention.

Profiling details. The cost of paying and non-paying MultiSend calls was evaluated the same way as MultiCall calls were, using calls with the same calldata (the empty bytestring) to the same contract (a dummy which stops immediately). Batches of 1 to 20 calls of each type were made, and the marginal cost of a call inferred by calculating the gas cost delta. Marginal gas costs varied in a range of 1464 to 1610 for non-paying calls, and 8176 to 8311 for paying calls, without a clear downward trend. On average, non-paying calls cost 1557 gas and paying calls 8268. Both are approximately 200 gas more expensive than their MultiCall equivalent, approximately 1% of the fixed transaction cost. Since the fixed transaction cost in real terms can rise to several U.S. dollars and the rate of transactions per second hovers around 10 to 15 [8], that can add up to a significant difference over time - 13.8 million dollars a year at the current transaction rate of 13.6 per second and gas price of 3.21 dollars per 21000 as of January 30, 2021 (assuming all current Ethereum transactions were instead batched).

[5]Future updates to MultiCall may affect the code size and therefore gas cost somewhat, but the order of magnitude is likely to remain the same.

[6]https://github.com/gnosis/safe-contracts/blob/8443cfaa410bfb197cc708b1c5e06ffa0c49c217/contracts/libraries/MultiSend.sol

Table 3: MultiCall gas cost vs MultiSend

Action	MultiCall cost	MultiSend cost (average)
Ether-paying call	8062	8268
Non-paying call	1360	1557

As MultiSend is a popular batcher written in assembly with the purpose of reducing gas costs compared to plain Solidity, that is an encouraging result.

Comparison discussion. It bears noting that comparing the cost of calls does not factor in the additional features required to use MultiSend that are built into MultiCall. Because MultiSend doesn't support metatransactions or have multiple account records for different users, each user is expected to separately delegate to it from a different contract. Payments between users of MultiSend also require at least one call, while multiple users could make deposits to other users without performing a call in MultiCall. Baking many features into the same contract reduces the overhead of context switching; interpreters like MultiCall are a promising avenue for doing so.

Each individual call instruction to MultiCall or MultiSend requires some data to be uploaded; comparing the upload costs of the different instructions, it appears that MultiCall's instructions are about 200 gas cheaper to upload. MultiCall's advantage likely originates in more efficient instruction packing and parsing. The DSL approach made it easy to write a code-generating combinator that parses a given list of argument types once, and then use it in many different instructions; Solidity assembly does not provide such code combinators. Since development effort is finite, more ergonomic development can lead to better performance.

In summary, the results of evaluation are promising and suggest greater adoption of batching would be advisable. MultiCall's interpreter design does not appear to impose unacceptable overheads, and could be reused in future smart contracts.

6 DISCUSSION

This work has focused primarily on the basic concept of a batching interpreter, and on a prototype on-chain implementation. While the gas usage of the prototype has been profiled with good results, a number of challenges stand between MultiCall and real-world adoption.

First of all, the question arises why batching has not yet achieved mass adoption, despite preexisting batchers which could also provide significant gas savings. We speculate that for individual users who don't frequently send many transactions at once, the effort of learning the user interface of an existing batching tool is not worth the money saved. MultiCall's ability to share calls between multiple users via metatransactions may ease adoption of batching, as it makes even single calls from one signatory cheaper to batch than send directly. However, changes to wallet user interface software would be required to make batcher use effortless for the end user. The problem of aligning incentives of wallet client providers and batcher developers is beyond the scope of this paper.

Security and incentive alignment of participants is a critical issue. Indeed, we are aware of two so-called transaction ordering-dependency vulnerabilities in MultiCall. Transaction ordering attacks occur when a malicious party observes transactions in flight (when they're broadcast but not yet added to the blockchain) and preempts them, or causes transactions to be mined in a harmful order [13]. The first attack on MultiCall works as follows: when a batching server Bob combines multiple metatransactions into a single call transaction to MultiCall, then the attacker runs one of them in a different transaction first. Since metatransactions are replay-protected, that will make one in the original transaction invalid. For efficiency, the validity check is only done after the metatransaction execution (alongside the settling of the user's balance based on their ether expenditure, calculated during execution), necessitating a revert if it's found to be invalid. Consequently, the attacker can waste all the gas used by Bob, costing them money. Since MultiCall is meant to fit an entire block's worth of transactions into a single one, the amount lost could be significant. The second attack is similar: a metatransaction can also be invalidated by its signatory by simply spending ether from the user's MultiCall account in another transaction, causing it to be insufficient when the metatransaction is run. A trivial mitigation for both attacks would be to announce the ether to be spent in the signed instruction, do the validity check in the beginning, and skip the metatransaction rather than throw an exception if it is invalid. However, the cost of the balance read and metatransaction upload is significant and would still be wasted. The problem would seem to arise from insufficient control of who may modify user accounts in MultiCall, enabling attempts to spend from them to be invalidated by attackers. The issue could be solved entirely by allowing users to specify a specific batching server which has the exclusive right to access their account promptly, thus allowing the server to batch the user's metatransaction without fear of tampering.

One potential objection to the approach of using an off-chain batching server would be that it imposes centralisation on a system designed to be decentralised, which might compromise censorship resistance. However, we argue allowing multiple competing batching servers would render them analogous to mining pools. While it is true that a system without more-central nodes such as mining pools or exchanges would be more decentralised, in practice the current arrangement is "decentralised enough" because all or most of them would have to collude to successfully impose transaction censorship.

Verification of the security of MultiCall with the help of existing tools such as [12] is a subject for future work. The off-chain component of batching merits deeper investigation, in particular from the perspective of preventing front-running and denial of service. Whether and to what degree batching increases the latency of batched transactions (especially urgent metatransactions batched into a large high gas price transaction) would be an interesting subject of an empirical study.

7 RELATED WORK

MultiCall is a technology intended to reduce the execution cost of on-chain contract calls and creation. Off-chain scaling solutions such as state channels [14] which save gas by avoiding on-chain execution entirely are of independent interest, but we choose to focus on on-chain optimisation solutions.

These can broadly be divided into two groups: micro-level and macro-level optimisation. Micro-level optimisations optimise individual contracts to reduce their creation and execution cost without changing their externally observable behaviour. Macro-level optimisations save gas by restructuring smart contracts, changing their API and potentially the transaction workflow. Essentially, they optimise systems of contracts. MultiCall and other batchers are of the latter sort. The approaches are complementary: micro-level optimisations can be applied to contracts after their structure and API have been designed. However, there is reason to think macro-optimisation can provide larger savings: making a particular contract use less gas with the same behaviour will not enable it to use fewer transactions, for example. Other expensive operations such as storage writes may also be easier to eliminate by varying the design of multi-contract systems than optimising single contracts.

7.1 Micro-optimisation

Chen et al. [6] developed GASPER, a tool which searches for inefficient patterns in EVM bytecode. Applied to all contracts on the blockchain as of 2016, it showed a significant proportion of contracts were under-optimised. One example inefficient pattern was fetching a storage word in a loop; that's optimised by fetching it once and caching it.

In a spiritual sequel to [6], Chen et al. introduce GasReducer [7], a tool which finds more inefficient patterns and performs bytecode-to-bytecode optimisation. GasReducer is evaluated by tracing the EVM code execution of all transactions as of 2017. The evaluation showed 9 billion gas was wasted to inefficient code patterns detected by GasReducer, vindicating the approach.

That Chen et al scan existing contracts and show there are significant savings (in monetary terms) to be made is interesting; not only does it show the value of gas optimisation, it's an inspiring approach to evaluating on-chain artefacts. Since the cost model is formalised and the actual transaction history is publicly available, obtaining real and accurate performance data is much easier than on physical machines. Augmenting evaluation of MultiCall with real transaction history could be a subject for future work.

Albert et al. created a super-optimising tool for the EVM, which finds the optimal code for straight-line segments containing arithmetic and bitwise instructions by exhaustive search [5]. Using a data set of transactions to the 128 most-called smart contracts, they obtain a potential gas optimisation of 0.59%. We suspect that the dominance of the cost of instructions which access the ledger state compared to arithmetic is responsible for the small saving relative to that provided by batchers. Nonetheless, any gas cost reduction is welcome.

MultiCall consists of manually optimised EVM assembly and already uses techniques such as caching storage words, but it would be interesting to apply automatic optimisation to it. Optimising MultiCall was already a significant effort; manually optimising more complex interpreters with additional functionality may quickly become infeasible as they grow.

7.2 Macro-optimisation

The patterns used in MultiCall's design (batching, proxies and metatransactions) are well-established, but the manner in which they've been combined is novel.

MultiCall is to our knowledge unique in being expressly designed to batch a full block of transactions in one, and the first application of an interpreter smart contract to batching. Aside from its interpreter design and programming language used, the features provided by MultiCall differ in two main ways. First, it combines account structs in the batcher with metatransactions to allow multiple signatories to control a single batcher smart contract (MultiCall) in a single call. Secondly, MultiCall and the proxies it controls allow the batching of create transactions as well as calls (both directly from MultiCall and via a proxy).

Transaction batching, metatransactions and proxies. Transaction batching is a method of reducing on-chain transaction execution costs by emulating a number of transactions with a smaller number that have an equivalent effect. The concept is well-known to Ethereum developers. Different batching techniques used for airdrops (mass transfers of a token intended to boost adoption of it) were studied and compared in [9]. Several payment batching contracts on the blockchain[7] [8] allow the caller to make payments in a single currency to a number of recipients. The company Autherium also provides wallet proxy contracts written in Solidity which can batch calls on behalf of their owner[9]. Such contracts can receive an array of metatransactions (specifying a call to make) signed by the contract's owner, verify the signature of each and then execute them. MultiCall allows signature verification to be shared across a sequence of actions, which is more efficient. Storing batching logic in individual users' wallet contracts is also expensive, because code storage costs gas.

The batcher MultiSend[10] (mentioned in section 5) solves the code size issue by allowing user wallet contracts to delegate to a shared library using the EVM instruction `DELEGATECALL`, which executes the code of the callee in the storage context of the caller.

Existing batchers could be retrofitted with most of MultiCall's functionality by combining them: one batcher such as MultiSend can be used to send metatransactions to wallet contracts such as Authereum's which accept them. Wallets which can only perform calls can be retrofitted with the ability to create contracts by calling a simple contract which just creates a contract using the given calldata as its creation code. We do not consider this a threat to MultiCall's validity: as calls are costly, a monolithic approach is efficient. MultiCall is a prototype of that approach.

Transaction batching is also used in Bitcoin: transactions natively support sending to multiple outputs, which can be used to reduce transaction costs by up to 80% [11].

[7]https://multisender.app/
[8]https://etherscan.io/address/0x2f6321db2461f68676f42f396330a4dc4a8f49df#code
[9]https://github.com/authereum/contracts/blob/master/contracts/account/AuthKeyMetaTxAccount.sol
[10]https://github.com/gnosis/safe-contracts/blob/8443cfaa410bfb197cc708b1c5e06ffa0c49c217/contracts/libraries/MultiSend.sol

7.3 On-chain interpreters

Other smart contracts which implement interpreters have been developed for Ethereum in order to scale contract execution. The Optimistic Virtual Machine (OVM) is an interesting example, used by the Optimistic Rollup scaling solution[11]. Optimistic Rollup allows users to run another less-replicated and therefore cheaper blockchain linked to Ethereum. The rollup chain is secured by enabling users to verify its execution and prove fraud via a verifier contract on the Ethereum chain. To enable Ethereum smart contract execution on the rollup chain, the verifier contract implements an EVM code interpreter called OVM (Optimistic Virtual Machine). Its purpose is quite different: where MultiCall instructions are run to perform actions on the blockchain, OVM is used to verify off-chain computations in the event of dispute. To be useful, MultiCall *must* be run, whereas the OVM need only be *available* to be run in the optimistic case. Moving computation off-chain can yield significant savings, but because off-chain scaling still requires transactions on the main chain to be secure, on-chain gas optimisation is not obsolete. An interpreter for a simple virtual machine called Lanai is also used to verify off-chain computations[12] as part of the TrueBit scaling solution.

8 CONCLUSION

We have implemented MultiCall, an interpreter for the Ethereum blockchain whose instruction set is designed for batching transactions. We demonstrated significant savings for both micro-benchmarks as well as a typical token-transfer smart contract, and MultiCall's performance is compared favourably to a preexisting batcher that was also written at a low level of abstraction in order to maximise performance.

Our idea of deploying batching interpreter smart contracts with rich instruction sets has been shown to be an effective and systematic mechanism to substantially save gas. To the best of our knowledge this paper is the first work to demonstrate that.

MultiCall's metatransaction and deposit features allow multiple users to operate the batcher in a single call, and to interact with other users. It is an example of how allowing safe resource sharing between mutually distrustful parties (the signatories) can help reduce costs.

As future work, we plan to expand the MultiCall instruction set to permit more safe sharing of resources.

REFERENCES

[1] Truffle overview. Truffle Blockchain Group, https://www.trufflesuite.com/docs/truffle/overview, accessed: 2021-01-29

[2] trufflesuite/ganache-cli: Fast ethereum rpc client for testing and development. Truffle Blockchain Group, https://github.com/trufflesuite/ganache-cli, accessed: 2021-01-29

[3] EIP-20: ERC-20 Token Standard (2015), https://eips.ethereum.org/EIPS/eip-20, accessed: 2021-01-29

[4] Solidity documentation. The Ethereum Foundation (2021), https://solidity.readthedocs.io/en/v0.8.1/, accessed: 2021-01-29

[5] Albert, E., Gordillo, P., Rubio, A., Schett, M.A.: Synthesis of super-optimized smart contracts using max-smt. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 177–200. Springer International Publishing, Cham (2020)

[6] Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 442–446 (2017). https://doi.org/10.1109/SANER.2017.7884650

[7] Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards saving money in using smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER). pp. 81–84 (2018)

[8] Etherscan.io: Ethereum (ETH) Blockchain Explorer, https://etherscan.io/, accessed: 2021-01-29

[9] Fröwis, M., Böhme, R.: The operational cost of ethereum airdrops. In: Pérez-Solà, C., Navarro-Arribas, G., Biryukov, A., Garcia-Alfaro, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 255–270. Springer International Publishing, Cham (2019)

[10] Griffith, A.T.: Ethereum meta transactions (2018), https://medium.com/@austin_48503/ethereum-meta-transactions-90ccf0859e84

[11] Harding, D.A.: Saving up to 80% on bitcoin transaction fees by batching payments (2017), https://bitcointechtalk.com/saving-up-to-80-on-bitcoin-transaction-fees-by-batching-payments-4147ab7009fb

[12] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022

[13] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 254–269. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, https://doi.org/10.1145/2976749.2978309

[14] Poon, J., Dryja, T.: The bitcoin lightning network (2016), https://lightning.network/lightning-network-paper.pdf, accessed: 2021-01-29

[15] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2020), https://ethereum.github.io/yellowpaper/paper.pdf, accessed: 2021-01-29

[11] https://optimism.io

[12] https://github.com/TrueBitProject/lanai