This post is version 1.1 of theoriginal proposal. The summary of changes are:

- Exceptional halts do not burn oil.
- ETH refund is based on the remaining oil exchanged at gasPrice

.

- Meta-transactions can still work with oil.

Introduction

This proposal for adding a second fuel source to the EVM called oil

that is unobservable and works in parallel with gas. The idea was originally devised byAlexey Akhunov, and the name of oil was suggested by Griffin Hotchkiss. Thanks to Martin Swende for the encouragement to pursue the idea.

Motivation

- Gas is currently being used for two different purposes:
- To pay for compute, memory, and storage resources
- To prevent re-entrancy by hardwiring the amount of gas a call can use.
- To pay for compute, memory, and storage resources
- To prevent re-entrancy by hardwiring the amount of gas a call can use.
- Adjusting the gas schedule to better reflect resource usage causes unintended consequences because contracts may be written in a way such that correctness depends on a specific gas schedule.
- Making an instruction cheaper may make a re-entrancy path feasible
- Making an instruction more expensive may make a call fail because the amount of gas hard-wired to it is now insufficient to execute the call
- Making an instruction cheaper may make a re-entrancy path feasible
- Making an instruction more expensive may make a call fail because the amount of gas hard-wired to it is now insufficient to execute the call
- Oil is a new fuel source that works very similarly to gas, but works in parallel to it.

Specification

- A transaction has a gasLimit

and gasPrice

.

- Currently, a transaction pays E

ether for allocating gasLimit

amount of gas to the transaction based on the gasPrice

.

- With oil, a transaction pays E

ether for allocating gasLimit

amount of gas to the transaction based on the gasPrice

, and additionally oilLimit

amount of oil to the transaction where oilLimit

is set equal to gasLimit

.

- A transaction still only specifies a gasLimit

. The EVM will internally set the oilLimit

to be the same as the gasLimit

specified by the transaction.

- Gas metering and gas semantics do not change.
- If the transaction runs out of oil at any point during execution, the transaction reverts. Unlike with gas, where out-of-gas reverts only the current frame, and lets the caller examine the result, out-of-oil always reverts the entire transaction (all frames).
- A caller contract cannot restrict how much oil a callee contract can use, unlike gas.
- The oil cost of all instructions is exactly the same as the gas cost, until further EIPs to modify oil schedule to reprice EVM operations.
- An OIL

instruction to read current oil will not be added, and this is intentional.

- The amount of ETH refunded for a transaction is now calculated based on the remaining oil, rather than remaining gas.
- If the transaction has an EVMC_SUCCESS

status code, the sender is refunded the amount of ETH that is the remaining oil exchanged at the gasPrice

.

- Similarly, if the transaction has an EVMC_REVERT

status code, the state is reverted as usual, and the sender is refunded the amount of ETH that is the remaining oil exchanged at the gasPrice

.

- If the transaction has an EVMC_SUCCESS

status code, the sender is refunded the amount of ETH that is the remaining oil exchanged at the gasPrice

.

- Similarly, if the transaction has an EVMC_REVERT

status code, the state is reverted as usual, and the sender is refunded the amount of ETH that is the remaining oil exchanged at the gasPrice

.

- The EVM handles exceptional halts by burning the remaining gas so the caller does not get a gas refund. An exception halt occurs when the EVM encounters an 1) an opcode which currently does not correspond to any instruction, 2) the INVALID

opcode, 3) stack underflow, or 4) jump to an invalidation destination. In contrast, oil is never burned even in exceptional halts. Thus, for some contracts invocations the oil cost will be less than gas cost.

Meta-transactions

- A meta-transaction is a payload that is sent to a relayer smart contract, which is then tasked with converting the payload into an actual transaction by funding it with gas and executing it. Systems like the GnosisSafe depend on this mechanism. Meta-transaction relayers need to be able to inspect the result of the relayed transaction in order to make that result conditional on certain payment to the relayer.
- One concern with oil is whether it breaks meta-transactions since an out-of-oil exception that occurs inside the relayed transaction deprives the relayer contract from the ability to extract payment for its service. This opens up a griefing

vulnerability where a malicious sender may cause the meta-transaction relayer run out of funds by paying for gas/oil, but not being able to receive any compensation. However, this vulnerability can be addressed by over-estimating how much oil the relayed transaction will need so that an out-of-oil exception does not occur.

- One can calculate a naive upper bound on the amount of oil needed for the relayed transaction by comparing the gas schedule and oil schedules, and finding the operation that has the largest ratio of oil-to-gas cost, maxR

. Then, the value gas*maxR

is a naive upper bound for oil consumption.

- Calculating a tighter upper bound is possible by leveraging additional information such as the graph of JUMPDEST

s and stack convergence.

Example

Consider the following two contracts where contract A

is stored at address a

and contract B

is stored at b

. Initially, let the gas cost of each instruction equal the oil cost of each instruction.

contract A { function set(B b) public { b.set(); } } contract B { uint public amount; function set() public { amount = address(this).balance; } }

Suppose a transaction TX_1

is sent to A

to invoke A.set

on B

with initial gas G_{init}

, where G_{init}

is set to exactly the gas cost of executing a

.set(

b

)

. Then, the initial oil O_{init}

would be equal to G_{init}

and the transaction would be accepted.

Now, suppose the oil cost of the BALANCE

opcode is increased and that a TX_2

is sent that is identical to TX_1

. This transaction TX_2

would get rejected with an out-of-oil error because the total oil cost would exceed O_{init}

.