Thank you to @villanuevawill for his numerous contributions to this outline.

Recommended reading / watching:

- @vbuterin: Phase 2 Proposal 2
- @vbuterin: Phase 2 presentation at the Eth 2.0 workshop in Brooklyn
- @villanuevawill: A Journey Through Phase 2 (a great place to start if you're newer to Eth 2.0)
- ewasm EEI spec

The Ethereum execution environment will provide a smart contract platform similar to Ethereum 1.0. From here on, it will just be referred to as Ethereum. In order to achieve the same guarantees as Ethereum 1.0, the execution environment must maintain some context that is immutable from the smart contract execution (i.e. concepts like msg.sender

). The approach taken by this proposal is to offer smart contracts a wrapped version of the mutating methods from the EEI, while validating that the code submitted to Ethereum does not make use of the unwrapped EEI functions. Using this framework, we also describe how msg.value

can be implemented to support value transfers of arbitrary token types.

Out of scope:

- gas / fee payments
- account abstraction implementations
- stateless design

In general, the goal of this proposal is to present three novel concepts: key-value denominated context stored in the client, dynamically linked code, and arbitrary value transfers in transactions. The actual architecture of the final solution will likely be radically different than what we have outlined, but may include some of the novel concepts described here.

# Additions to EEI

### get_context() -> bytes

In some cases it may be valuable to store runtime context between calls that are managed by the client. In order to support this, we'll assume that the EEI methods which spawn new calls will take an additional ctx

parameter that will be pushed onto the context stack.

There are three special context keys, two of which cannot be set by a caller, that the client will need to manage:

- es_caller: uint256

– The index of the execution script that invoked the current frame of execution. It should default to 2**256 - 1

if the current frame is the transactions entry point.

- es_callee: uint256

– The index of the execution script that running in the current frame of execution.

- ext: { string: bytes }

– A map of externally defined functions where the key is the function name and the value is the code to execute. Upon entering a new frame of execution via execute_code

or execute_script

, each function defined in this map will bedynamically linked (or appended to call code) into the new runtime environment.

### dynamically_link_function(name, code)

Stores a function in context

that it is dynamically linked in child runtimes.

# Ethereum Objects

### Transaction

{ 'target': Address, 'data': bytes, 'state': [2**256; State] }

### State

{ 'code': bytes, 'storage': bytes, 'proof': StateProof }

### Value

{ 'amount': uint256, 'token': Address }

## Wrapped Ethereum Interface (WEI)

To provide the security needed to execute arbitrary code such as smart contracts, the following functions will wrap the existing EEI functions.

### call_contract(target, data)

The call_contract

method is used to execute arbitrary code submitted to the EES while maintaining the caller, callee, value structure. It operates similarly as call

would in Solidity.

For simplicity, the value

is sent to the target

before switching contexts. An additional WEI function receive_value

would probably be a better approach as it would allow contracts to reject unwanted tokens and other custom behaviors.

def call_contract(target, data, value): ctx = eei.get_context() ctx['eth']['msg']['value'] = value ctx['eth']['msg']['sender'] = ctx['eth']['msg']['target'] ctx['eth']['msg']['target'] = target

```
# Transfer the `msg.value` to the `target`
# note: this interface could support sending multiple tokens
# via `msg.value` object by extending the `value` object
assert transfer_value(value, target)

assert verify_state(
    ctx['runtime']['pre_state_root'],
    ctx['runtime']['state'][target]
)

return eei.execute_code(ctx['runtime']['state'][target].code, data, ctx)
```

### call_execution_script(es_id, data)

The call_execution_script

method allows contract writers to interact with other execution environments on Ethereum. It maintains the caller, callee, value relationship, although the execution script destination doesn’t necessarily need to support Eth related operations. This opens the door for execution environment “ecosystems” that trust each other and can operate on the same shard without receipts.

def call_execution_script(es_id, target, data, value): ctx = eei.get_context() ctx['eth']['msg']['value'] = value ctx['eth']['msg']['sender'] = ctx['eth']['msg']['target'] ctx['eth']['msg']['target'] = target return eei.execute_script(es_id, data, ctx)

### set_storage

A contract should only have access to the storage value that is mapped to by its address. The set_storage

method leans on the ctx

described above to determine the current frame of execution and allows access to the Ethereum storage accordingly.

def set_storage(key, value): ctx = eei.get_context() target = ctx['eth']['msg']['target']

```
assert verify_state(
    ctx['runtime']['pre_state_root'],
    ctx['runtime']['state'][target]
)

storage = ctx['runtime']['state'][target]
storage[key] = value

ctx['runtime']['state'][target] = storage
```

# Interface

## process_block

All execution will begin with process_block

. This will iterate through all transactions in the block, and will call the intended contract with the provided data. Contract initialization strategies are ommitted here in favor of clarity around the other novel concepts outlined above.

def process_block(transactions, pre_state): post_state = {} receipts = []

```
for tx in transactions:
    state = tx.state

    # Verify that the tx.target's witness is valid against the
    # existing shard state root
    if not verify_state(
        pre_state,
        state[tx.target]
    ):
        continue

    account = state[tx.target].code

    # Setup an empty value for the entry call, let the wallet contract
    # parse the call to determine the correct value + context
    value = {
        amount: 0,
        token: 2**256 - 1
    }

    # Initial context
    ctx = {
        'eth': {
            'msg': {
                'value': None,
                'sender': None,
                'target': tx.target
            }
        },
        # Since txs will provide their own state + witnesses, it is important
        # to add this to the context of the initial call
        'runtime': {
            'pre_state_root': pre_state,
            'state': tx.state,
            'receipts': []
        },
        # Stores the wrapped functions in the special ctx key to be linked
        # into child runtimes
        'ext': get_wei_code()
    }

    # Pass off the tx data to the contract
    eei.execute_code(account.code, tx.data, ctx)

    ctx = eei.get_context()
    post_state.update(**ctx['runtime']['state'])
    receipts.append(ctx['runtime']['receipts'])

# Return the resulting state and receipts for the block
return (
    hash_tree_root(post_state),
    hash_tree_root(receipts)
)
```

# Comparing Against Continuations

In Vitalik's latest phase 2 proposal he gives the following lifecycle of a contract call:

A contract call roughly works by starting with stack = operations[::-1], then while len(stack > 0), pop the top operation off the stack and run executeCode(target.code, (target.state, operation.calldata)); this would be expected to return (new_state, continuation); the state transition function would set the state to equal the new state, and then add the continuation to the stack if any.

Following that construction, it would be possible to build a context manager with no additional EEI functions using the existing tools available to beacon chain contracts. It might look something like the following pseudocode (msg.value

and receipts omitted for simplicity).

def proccess_block(block, pre_state): post_state = {} receipts = []

```
for tx in block:
    stack = [tx]

    ctx = {
        'eth': {
            'msg': {
                'sender': None,
                'target': None
            }
        }
    }

    while len(stack) > 0:
        op = stack.pop()

        if not verify_state(
            pre_state,
            tx.state[op.target]
        ):
            break

        ctx['eth']['msg']['sender'] = ctx['eth']['msg']['target']
        ctx['eth']['msg']['target'] = op.target

        (new_state, continuation) = eei.execute_code(
            tx.state[op.target].code,
            (
                post_state[op.target] or tx.state[op.target].storage,
                op.calldata,
                ctx
            )
        )

        post_state[op.target] = new_state

        if continuation:
            op = {
                'target': continuation.target,
                'calldata': continuation.calldata
            }

            stack.append(op)

return (post_state, receipts)
```

With this sort of execution environment in mind, the continuation paradigm would be a fundamental piece of all smart contract development. Although this could be abstracted away from the user through development tooling, the resulting binaries will still need to be a fractured set of functions which increases progressively as the number of contract calls rise.

This example below is provided for clarity. Cross shard calls are not addressed in this example, but would follow the same pre/post processing pattern in both paradigms.

# continuation approach

def pre_process_employee_eligibility(employee_name, employee_data_contract_address): employee = get_storage[employee_name]

```
# push onto the stack the functions to be performed
return [
    generate_contract_call(
        employee_data_contract_address,
        (GET_SALARY_OP, employee.id),
```

```
    ),
    self.post_process_employees
]
```

# once the contract call is complete, call this function with the return data

def post_process_employees_eligibility(return_value): data = deserialize(return_value, SalaryReturnValue)

```
if data.salary < 50000:
    set_storage(data.employee_id, APPROVED)
else:
    set_storage(data.employee_id, REJECTED)
```

# dynamic linking approach

def process_employee_eligibility(employee_name, employee_data_contract_address): employee = get_storage[employee_name]

```
salary = wei.contract_call(
    employee_data_contract_address,
    (GET_SALARY_OP, employee.id),
)

if salary < 50000:
    wei.set_storage(employee.id, APPROVED)
else:
    wei.set_storage(employee.id, REJECTED)
```

There are some pros and cons to the continuation approach versus a context construction built into the client.

Pros:

- less opinionated
- functions may need to be split anyways for cross shard calls
- no additions to the current design

Cons:

- even in this construction, there is no trusted way of signaling to another beacon chain contract which execution environment a local call is coming from
- since each contract call requires a return, functions with contract calls will need to be split up by the HLL -> wasm compiler (potentially into many pieces)
- wasm runtime instantiations would increase by a factor of ~2, i.e. unless the contract call is the last line of a function, the VM will need to continue processing the function that initiated the call

We believe that at the core Ethereum VM, it’s best to not make assumptions about developers’ intentions. Therefore, having a general purpose context manager outside the runtime will provide the same benefits as a continuation model plus better performance and clearer low level code.

Any thoughts on the general ideas we’ve proposed?