ParallelChain Mainnet Whitepaper

Adam Version

Alice Lim[*] and the ParallelChain Lab Team

January 18, 2023

1 Introduction

This Whitepaper describes the ParallelChain Mainnet Protocol. The Protocol specifies how the different components of the ParallelChain Mainnet blockchain operates. In doing so, it serves three purposes:

- It lets users know exactly what to expect when they interact with the ParallelChain Mainnet: no surprises or broken promises.
- It helps implementors of ParallelChain software components ensure that their work interoperates smoothly with other existing components.
- It records changes to The Protocol over time, helping network updates proceed without unforeseen hitches.

Besides specifying The Protocol, this living document also provides justification for The Protocol's more esoteric, less obvious aspects. Of course, 'obvious' is subjective, and what is obvious to one person may not be so obvious to another. With this in mind, this document strives to strike a healthy balance between describing protocol behavior as succinctly as possible, which implementors should prefer, and narratively walking through the authors' design thinking, which other readers, e.g., token holders, the press, and teams or individuals working on derivative and related work should prefer. The goal of this project is not only to build a good trustless, distributed compute and storage platform, but also to develop the programming community's collective competency in building secure, maintainable, and useful distributed systems.

1.1 Relation with HotStuff-rs

HotStuff-rs is a Rust Programming Language implementation of the HotStuff consensus protocol, also created by the authors of this paper. ParallelChain Mainnet uses HotStuff-rs to replicate its state machine across a large set of machines around the globe. Membership in this 'Validator Set' is determined by the staking rules specified in this document. A combination of clever tokenomics and HotStuff's logically provable Byzantine Fault Tolerance ensures that the protocol is executed honestly by the Validator Set, and that committed Blocks remain committed.

This document makes frequent reference to concepts defined in HotStuff-rs (paper coming soon). To keep this document as self-contained as possible, the referenced HotStuff-rs concepts are listed in Appendix A.

2 Message Types

This section specifies types that Protocol implementations share across the network in messages. All types are serialized using near/borsh.

[*]Distinguished Engineer at ParallelChain Lab.

2.1 Cryptographic Primitives

- Public Address (`[u8; 32]`) Ed25519 Public Key.
- Private Key (`[u8; 32]`): Ed25519 Private Key.
- Signature (`[u8; 64]`): Ed25519 Signature.
- Cryptographic Hash (`[u8; 32]`): SHA256 hashes.
- Binary Merkle Trees and Binary Merkle Proofs: as generated by antouhou/rs-merkle.
- Base-16 Modified Merkle Patricia Tries (MPT) and MPT Proofs: as generated by paritytech/trie.
- Bloom Filters (`Vec<u8>`): as generated by paritytech/parity-common/ethbloom.

2.2 Block Header

A structure with fields:

- Hash (`CryptoHash`): the SHA256 Hash over (Block Height ++ Justify ++ Data Hash).
- Height (`u64`): the number of Justify-links between this Block and the Genesis Block. 0 for the Genesis Block.
- Justify (`QuorumCertificate`): a QuorumCertificate that points to the Block's parent.
- Data Hash (`CryptoHash`): the SHA256 Hash over (Chain ID, Proposer, Timestamp, Base Fee Per Gas, Transactions Hash, Receipts Hash, State Hash, Logs Bloom).
- Chain ID (`u64`): a number unique to a particular ParallelChain Mainnet-based blockchain. This prevents, for example, Blocks from one chain from being published in another chain as evidence of malfeasance.
- Proposer (`PublicAddress`): the Public Address of the Validator that is the Leader of the View this Block was proposed in.
- Timestamp (`u64`): a Unix timestamp. Same as Ethereum, this must be less than 900 seconds greater than the parent Block's timestamp. Additionally, Validators should reject Blocks with Timestamps lower than its local timestamp.
- Base Fee Per Gas (`u64`): The (inclusive) minimum number of Grays that a Transaction included in this Block must pay for every Gas used. This value ($BaseFeePerGas_n$) in the Block with height n is determined by the formula:

$$BaseFeePerGas_n = BaseFee_{n-1} * (1 + \frac{GasConsumed_{n-1} - TargetGasUsed}{TargetGasUsed} * 1/8)$$

- Transactions Hash (`CryptoHash`): the Binary Merkle Tree root hash over the Block's Transactions.
- Receipts Hash (`CryptoHash`): the Binary Merkle Tree root hash over the Block's Receipts.
- State Hash (`CryptoHash`): the SHA256 root hash of the blockchain's World State Merkle Patricia Trie (MPT) after executing all of this Block's Transactions.
- Logs Bloom (`Vec<u8>`): the 256-byte Block-level Bloom Filter union of all the Bloom Filters of each Log topic from the Block's Receipts.

2.3 Block

A structure with fields:

- Block Header (`BlockHeader`).
- Transactions (`Vec<Transaction>`): a dynamically-sized list of Transactions.
- Receipts (`Vec<Receipt>`): a dynamically-sized list of Receipts. If a Block contains a Transaction, it must also contain its Receipt. Receipts appear in the order of their Transactions.

The serialized size of a Block must not exceed *BlockSizeLimit* bytes.

2.4 Transaction

A structure with fields:

- Command (`TransactionCommand`): a Transaction Command Enum (2.4.1), specifying what the Transaction 'should do'.
- Origin (`PublicAddress`): the address of the External Account on the sending end of this Transaction. For Next Epoch Transactions, this must be all zeros.
- Hash (`CryptoHash`): the SHA256 hash over Signature.
- Signature (`Signature`): the Ed25519 Signature over (Command, Origin, Nonce, Gas Limit, Max Base Fee per Gas, Priority Per Gas) using the Private Key corresponding to Origin.
- Nonce (`u64`): the number of Transactions on chain with the same Origin that are on chain before this Transaction, unless the Transaction is a Next Epoch Transaction, in which case its value must be 0.
- Gas Limit (`u64`): the amount of Gas that the network should expend executing the Transaction.
- Max Base Fee per Gas (`u64`): The number of Grays the Origin Account is willing to pay the network for the work of processing this Transaction.
- Priority Fee per Gas (`u64`): the number of Grays the Origin Account is will pay the Proposer for including this Transaction in a Block.

2.4.1 Transaction Command

An enum with the following variants (operations) and fields (input):

Operation	**Input**
	Account Transactions
Transfer	• Target (`PublicAddress`): the address of the Account receiving the transfer. • Amount (`u64`): the number of Grays to transfer from Transaction.Origin to Target.

Deploy	• Contract (`Vec<u8>`): a vector of bytes satisfying the Contract Binary Interface (5). • CBI Version (`u32`): the version of the CBI that Contract targets. Must be 0 (currently the only version of the CBI). • Init Arguments (`Option<Vec<Vec<u8>>>`): if Some, the Contract's **`init`** guest function will be called, and this value (whole value, not only the value inside the `Option`) will be made available through the **`arguments`** host function. • Amount (`Option<u64>`): the number of Grays to transfer from Transaction.Origin to the Contract Account created by this Transaction before its **`init`** function is (potentially) called.
Call	• Target (`PublicAddress`): the address of the Contract Account that will be called. • Method (`Vec<u8>`): the value to be made available through the **`method`** host function inside the Contract call. • Arguments (`Option<Vec<Vec<u8>>>`): the value to be made available through the **`arguments`** host function inside the Contract call. • Amount (`Option<u64>`): the number of Grays to transfer from Transaction.Origin to Target before the Contract Call.

Network Transactions	
NextEpoch	None.
RegisterPool	• Commission Rate (`u8`): the percentage ($[0, 100]$) of the Epoch rewards of a non-own Stake in the Pool that will be received by its Operator, which will be Transaction.Origin.
ModifyPool	• New Commission Rate (`u8`): the *new* percentage ($[0, 100]$) of the Epoch rewards of a non-own Stake in the Pool that will be received by its Operator, Transaction.Origin.
UnregisterPool	None.
RegisterStake	• Operator (`PublicAddress`): the address of the Operator of the staking Pool that the created Stake will target. • Balance (`u64`): the number of Grays that will be locked up in the created Stake.
ReleaseStake	• Operator (`PublicAddress`): the address of the Operator of the Pool that the stake to be released targets.

2.5 Receipt

A structure with fields:

- Gas Used (`u32`): the number of units of gas expended in executing the Transaction. The sum of a Block's Transactions' Gas Used must not exceed $TargetGasUsed \times 2$.
- Logs (`Vec<Log>`): the log emitted during a Contract Call, in the order of emission.
- Return Value (`Vec<u8>`): the return value of a Contract Call.
- Exit Code (`ReceiptExitCode`).

2.5.1 Receipt Exit Code

An enum with the following variants (code):

Code	**Description**
$OperationSuccessful$	The Transaction successfully accomplished everything that it could have been expected to do.
General Errors	
$OperationFailed$	The Transaction failed to accomplish the primary operation that Transactions of its kind are expected to accomplish.
$GasExhausted$	The Gas Limit was exceeded by a dynamically-costed activity in a dynamic-cost Transaction.

2.5.2 Log

A structure with fields:

- Topic (`Vec<u8>`).
- Value (`Vec<u8>`).

3 World State

The World State is a singleton mapping between Account addresses and Account states. It is stored in a Base-16 Modified Merkle Patricia Trie (MPT). The 32-bytes root hash of this MPT after the execution of all Transactions in a Block is included in the Block Header's State Hash field.

3.1 Account

An Account's state is composed of various fields, each stored at key address ++ a prefix specified in the first column of the table below:

Prefix	**Field**	**Type**	**Description**
`0`	Nonce	`u64`	For an External Account, the number of Transactions originating from this Account so far on chain. Empty for a Contract Account.
`1`	Balance	`u64`	The number of Grays owned by the Account.
`2`	Contract	`Vec<u8>`	for a Contract Account, the Contract's WASM Bytecode. Empty for an External Account.
`3`	CBI Version	`u32`	for a Contract Account, the version of the Contract ABI that the Contract's Code expects. Empty for an External Account.
`4`	Storage Hash	`CryptoHash`	for a Contract Account, the root hash of its Storage Trie. Empty for an External Account.

Each Contract Account is associated with an MPT called a Storage Trie. A Contract Account's Storage Trie is accessible from inside Contract Code and through Standard HTTP API endpoints as a set of key-value pairs. Keys and values can be any arbitrary bit-sequence. Though publicly readable, a Contract's Storage can only be mutated from inside Call Transactions, and then only from the specific Contract's Code.

3.1.1 Network Account Storage

Some of the state that ParallelChain Mainnet maintains is of network-wide significance, instead of only being relevant to a single Account. This state is maintained in the Storage Trie of an identified Account called the Network Account, which resides at address 000... This Account is not associated with Ed25519 material. The network-significant data that the Network Account stores is composed of various fields, each stored in its Storage Trie under a 1-byte prefix specified below for each field before its type.

Prefix	**Field [(shorthand)]**	**Type**
`0`	Previous Validator Pools (`pvp`)	`Vec<PoolSnapshot>`
`1`	Validator Pools (`vp`)	`Vec<PoolSnapshot>`
`2`	Next Validator Pools (`nvp`)	`BinaryHeap<Pool, MaxValidatorSetSize>`
`3`	Next Validator Pools Length (`nvp_len`)	`u16`
`4`	Pools	`Operator → Pool`
`5`	Stakes	`(Operator, Owner) -> Stake`
`6`	Current Epoch	`u64`

Pool Snapshot

- Operator (`OperatorAddress`).
- Total Balance (`u64`).
- Commission Rate (`u8`).
- Stakes (`[StakeSnapshot; MaxStakesPerPool]`).

Stake Snapshot

- Operator (`OperatorAddress`).
- Owner (`PublicAddress`).
- Balance (`u64`).
- Locked (`bool`).
- Included (`bool`).

Pool (dict type)

- Operator (`0; OperatorAddress`).
- Balance (`1; u64`): (Total Balance).
- Commission Rate (`2; u8`).
- Stakes (`3; BinaryHeap<Stake, MaxStakesPerPool>`).
- Stakes Length (`4; u16`).

Stake (dict type)

- Operator (`0; OperatorAddress`).
- Owner (`1; PublicAddress`).
- Balance (`2; u64`).
- Locked (`3; bool`).
- Included (`4; bool`).

4 Transactions

At the most abstract level, ParallelChain Mainnet is a replicated state machine with a state transition function of kind $(WorldState, Block) \rightarrow WorldState$, where *Block* here can be seen as a list of Transactions. A mechanism called Gas Billing remunerates Staking Pools and the network as a whole for the work that they do in maintaining this replicated state machine.

The most substantive parts of this section specify this replicated state machine by separately describing the execution of all of the different possible Commands. But, though Transactions with different Command Operations do broadly different work, there are a handful essential processing steps (most importantly, Gas Billing) that all Transactions must go through. These are specified in 4.1.

Excluding Next Epoch Transactions, which are discussed in 4.4, and in the normal case where the Gas Limit is sufficient, the execution of a Transaction proceeds through a fixed sequence of steps, or 'Phases': **Tentative Charge** → **Work** → **Charge**. The Work phases of Transactions of each possible Command are discussed in 4.2 and 4.3.

Exceeding Gas Limit in the Work Phase causes all World State sets done in the Work Phase to be reverted, *exit_code* to be set to *GasExhausted*, and execution to jump immediately to the Charge Phase.

In the following sequence flows, we use a well-defined syntax of steps. Executing each step causes specified side effects, and costs an unambiguous amount of gas. This is imperative if the network's state machine is to be replicated across multiple machines each potentially running different implementations:

Syntax	**Meaning**
[*let*] *var* ← *expr*	A variable assignment. *var* is either: • A global variable, or • If accompanied with *let*, a variable local to the procedure, or • A key in the World State, denoted in any of the various ways described in for next piece of syntax.
dict[key][.field]	A World State access. If on the left-hand-side of a ← a get, if on the right-hand-side, a set. Here, *dict* can be *ws*, *ns*, of one of the 'dictionary types' listed in 3.1.1, in which case the trailing *.field* specifies the field being accessed. In general, this statement costs some $Cost_{StateSet}$ or $Cost_{StateGet}$, and with the provisions for warm gets and Contracts as specified in 6.1.1. The exception is in the Work Steps of Network Transactions, where all gets are charged warm.

$dict.contains(expr)$	Checks whether a key is set to a value (i.e., is not $\emptyset$) in the World State. Evaluates into a `bool`. Some steps check for the existence of a key by checking if its value is $\neq \emptyset$ instead of using $.contains$. This is to save gas costs in cases where the value is also needed at a later step. Costs $Cost_{StateContains}$ as specified in 6.1.1.
abort	Causes all World State sets in the Work Phase to be reverted, *exit_code* to be set to *GasExhausted*, and execution to jump immediately to the Charge Phase.
return *Receipt*{...}	Ends execution and causes the specified Receipt to be included in the Block.
$Instantiate_{ver}(contract)$	Instantiate the *contract*, as specified in 5. Costs $Instantiate_{Ver,Cold}$ in the Deploy Work Step, and $Instantiate_{Ver,Warm}$ in the Call Work Step, as specified in 6.1.2.
$Call_{ver}(guest\_function)$	Call a guest function in a Contract, updating *logs* and *return_value* and increasing *gas_used* as Opcodes are executed and as host functions are called as specified in 5. Evaluates to the Return Value of the Call. Causes execution to jump to On Gas Exhausted when Gas Limit is exceeded as usual, and causes an **abort** when any other trap is raised.

The following global variables are used in the following sequence flows. These are initialized to a starting value at the beginning of a Transaction's Execution, and may be read or assigned to by steps:

- $base\_fee\_per\_gas$ (`u32`) and *proposer* (`PublicAddress`) the corresponding fields in the Header of the Block that the Transaction is part of.
- *gas_used* (`u64`): initialized to $TransactionBaseCost(txn)$, and increased along with the execution of each step by its gas cost. It does not matter whether this variable is increased before, or after the execution of its corresponding step.
- *exit_code* (`ExitCode`): initialized to $OperationSuccessful$.
- *logs* (`Vec<Log>`): initialized to a an empty vector.
- *return_value* (`Vec<u8>`): initialized to an empty vector.
- *ws*: the World State.
- *ns*: Network Account Storage.

In addition, the Work Phase of every Transaction Operation has access to the fields of its Command's Input. These are re-listed in the beginning of each Work Phase sequence flow after "**Data:**".

4.1 Common Phases

4.1.1 Tentative Charge

Algorithm 1: Tentative Charge Work Phase

```
  Data: $Transaction\{origin, nonce, gas\_limit, priority\_fee\_per\_gas\}$
1 if $ws[origin].nonce \neq nonce$ then
2     abort
3 if $gas\_limit > ws[origin].balance$ then
4     abort
5 $ws[origin].balance \leftarrow$
    $ws[origin].balance - gas\_limit \times (base\_fee\_per\_gas + priority\_fee\_per\_gas)$;
```

4.1.2 Charge

Algorithm 2: Charge Step Phase

```
Data: $Transaction\{origin, gas\_limit, priority\_fee\_per\_gas\}$
1 $ws[origin].balance \leftarrow$
    $ws[origin].balance + (gas\_limit - gas\_used) \times (base\_fee\_per\_gas + priority\_fee\_per\_gas)$;
2 $ws[Treasury].balance \leftarrow$
    $ws[Treasury].balance + TreasuryCutOfBaseFee \times gas\_used \times base\_fee\_per\_gas$;
3 $ws[proposer].balance \leftarrow ws[proposer].balance + gas\_used \times priority\_fee\_per\_gas$;
4 $ws[origin].nonce \leftarrow ws[origin].nonce + 1$;
5 return $Receipt\{gas\_used, logs, return\_value, exit\_code\}$;
```

4.2 Account Transactions

4.2.1 Transfer

Algorithm 3: Transfer Work Phase

```
Data: $Transaction\{origin\}, Input\{target, amount\}$
1 if $ws[origin].balance < amount$ then
2 |   abort
3 $ws[origin].balance \leftarrow WS[origin].balance - amount$;
4 $ws[target].balance \leftarrow WS[target].balance + amount$;
```

4.2.2 Deploy

Algorithm 4: Deploy Work Phase

```
Data: $Transaction\{origin, none\}, Input\{contract, cbi\_version, init\_arguments, amount\}$
1  let $contract\_addr \leftarrow sha256((origin, nonce))$;
2  if $let\ Some(amount) \leftarrow amount$ then
3  |   if $ws[origin].balance < amount$ then
4  |   |   abort
5  |   $ws[origin].balance \leftarrow WS[origin].balance - amount$;
6  |   $ws[contract\_addr].balance \leftarrow WS[target].balance + amount$;
7  if $let\ Ok(instance) \leftarrow Instantiate_{cbi\_version}(contract)$ then
8  |   $ws[contract\_addr].contract \leftarrow contract$;
9  |   $ws[contract\_addr].cbi\_version \leftarrow cbi\_version$;
10 |   if $let\ Some(args) \leftarrow init\_arguments$ then
11 |   |   $Call_{cbi\_version}(instance.init)$;
12 else
13 |   abort;
```

4.2.3 Call

Algorithm 5: Call Work Phase

```
   Data: $Transaction\{origin\}, Input\{target, method, arguments, amount\}$
1  if $let\ Some(amount) \leftarrow amount$ then
2      if $ws[origin].balance < amount$ then
3          abort
4      $ws[origin].balance \leftarrow WS[origin].balance - amount$;
5      $ws[target].balance \leftarrow WS[target].balance + amount$;
6  let $contract \leftarrow ws[target].contract$;
7  if $Contract \neq None$ then
8      let $cbi\_version \leftarrow WS[target].cbi\_version$;
9      let $Ok(instance) \leftarrow Instantiate_{cbi\_version}(contract)$;
10     $Call_{cbi\_version}(instance.action)$;
11 else
12     abort
```

4.3 Network Transactions

4.3.1 Register Pool

Algorithm 6: Register Pool Work Phase

```
   Data: $Transaction\{operator : origin\}, Input\{operator, commission\_rate\}$
1  if $pools.contains((operator, 00u8))$ then
2      abort;
3  $pools[operator].operator \leftarrow operator$;
4  $pools[operator].balance \leftarrow balance$;
5  $pools[operator].stakes\_length \leftarrow 0$;
6  $pools[operator].commission\_rate \leftarrow commission\_rate$;
7  $nvp.insert\_then\_extract((operator, 0))$
```

4.3.2 Modify Pool

Algorithm 7: Modify Pool Work Phase

```
   Data: $Transaction\{operator : origin\}, Input\{new\_commission\_rate\}$
1  if $!pools.contains((operator, 00u8))$ then
2      abort;
3  $pools[operator].commission\_rate \leftarrow new\_commission\_rate$;
```

4.3.3 Release Pool

Algorithm 8: Release Pool Work Phase

```
   Data: $Transaction\{operator : origin\}$
 1 let $stakes\_len \leftarrow pools[operator].stakes\_len$;
 2 if $stakes\_len == \emptyset$ then
 3     abort;
 4 for $let\ i \leftarrow 0$ to $stakes\_len$ do
 5     let $(owner, stake\_balance) \leftarrow pools[operator].stakes[i]$;
 6     if $!stakes[(operator, owner)].locked$ then
 7         $stakes[(operator, owner)] \leftarrow \emptyset$
           $ws[owner].balance \leftarrow ws[owner].balance + stake\_balance$;
 8     else
 9         $stakes[(operator, owner)].included \leftarrow \emptyset$;
10 end
11 $nvp.delete(operator)$;
12 $pools[operator] \leftarrow \emptyset$;
```

4.3.4 Register Stake

Algorithm 9: Register Stake Work Phase

```
   Data: $Transaction\{owner : origin\}, Input\{operator, balance\}$
 1 if $stakes.contains((operator, owner, 00u8))$ then
 2     abort;
 3 if $ws[owner].balance < balance$ then
 4     abort;
 5 $ws[origin].balance \leftarrow ws[origin].balance - balance$;
 6 if $let\ pool\_balance \leftarrow pools[operator].balance\ and\ pool\_balance \neq \emptyset$ then
 7     $stakes[(operator, owner)] \leftarrow \{\ operator, owner, balance, false, true\ \}$;
 8     if $let\ Ok(replaced\_stake\_balance) = pool.stakes.insert\_then\_extract((operator, owner))$
        then
 9         $new\_pool\_balance \leftarrow pool\_balance + (replaced\_stake\_balance - balance)$
           $nvp.insert\_then\_extract\_or\_change\_balance(operator, new\_pool\_balance)$;
10     else
11         abort;
12 else
13     abort;
```

4.3.5 Release Stake

```
Algorithm 10: Release Stake Work Phase
Data: $Transaction\{owner : origin\}, Input\{operator\}$
let $stake\_included \leftarrow stakes[(owner, operator)].included$;
if $stake\_included \neq \emptyset$ then
    let $stake\_locked \leftarrow stakes[(owner, operator)].locked$;
    if $stake\_included$ then
        let $pool \leftarrow pools[operator]$;
        let $Ok(removed\_stake\_balance) \leftarrow pool.stakes.delete(owner)$;
        if $!stake\_locked$ then
            $stakes[(operator, owner)] \leftarrow \emptyset$;
            $ws[owner].balance \leftarrow ws[owner].balance + removed\_stake\_balance$;
        else
            $stakes[(operator, owner)].included \leftarrow false$;
    else
        if $stake\_locked$ then
            abort
        else
            $stakes[(operator, owner)] \leftarrow \emptyset$;
else
    abort
```

4.3.6 Publish Evidence

Coming Soon.

4.4 Next Epoch

Coming Soon.

4.5 Binary Heap

Polymorphic Methods:

- `heap.insert_and_extract(entity: Entity::Key) → Result<Option<u64>, ()>`: u64 is the replaced entity's balance.
- `heap.delete(key: Entity::Key) → Result<u64, ()>`: u64 is the deleted entity's balance.
- `heap.change_balance(key: Entity::Key, new_key: Entity::Key`.

Other Methods:

- `nvp.insert_then_extract_or_change_balance(pool: Pool::Key, balance_change: u64)`.

5 Contracts

Contracts are code that are deployed into Contract Accounts. They can modify their Account's Storage Tries and control their balance.

In order to be deployed, Contract code needs to satisfy the version of the ParallelChain Contract Binary Interface (CBI) specified in the Deploy Input of the Transaction which deployed it.

Version 0.0 is the current version of the CBI, and specifies that Contracts:

- *Must* be valid WebAssembly modules as described in the WebAssembly 2.0 Specification.
- *Must not* import any function besides those specified in 5.1 (they may import less).
- *Must not* contain any Opcode listed in 5.4.

Additionally, to be callable using a Call Transaction (most Contracts will want this), they must export the `action` and `alloc` methods specified in 5.2.

5.1 Host Functions

5.1.1 Account State Accessors

`set(key_ptr: u32, key_len: u32, value_ptr: u32, value_len: u32)`

Sets a key to a value in the current Contract Account's Storage. Calling this function inside a View Call causes a panic.

`fn get(key_ptr: u32, key_len: u32, value_ptr_ptr: u32) -> i64`

Gets the value corresponding to a key in the current Contract Account's Storage.

`fn balance(balance_ptr_ptr: u32) -> u64`

5.1.2 Block Field Getters

`fn block_height(height_ptr_ptr: u32) -> u64`

Gets the Height of the Block which the Transaction at the start of the current Call Chain is included in. Calling this function inside a View Call causes a panic.

`fn block_timestamp(timestamp_ptr_ptr: u32) -> u64`

Gets the Timestamp of the Block which the Transaction at the start of the current Call Chain is included in. Calling this function inside a View Call causes a panic.

`fn prev_block_hash(hash_ptr_ptr: u32) -> u32`

Gets the Hash of the Parent of the Block which the Transaction at the start of the current Call Chain is included in. Calling this function inside a View Call causes a panic.

5.1.3 Call Context Getters

`fn calling_account(address_ptr_ptr: u32) -> u32`

Gets the Address of the Account that triggered the current Call. This could either be an External Account (if the Call is directly triggered by a Call Transaction), or a Contract Account (if the Call is an Internal Call). Calling this function in a View Call causes a panic.

`fn current_account(address_ptr_ptr: u32) -> u32`

Gets the Address of the current Account.

`fn method(method_ptr_ptr: u32) -> u32`

Gets the Method of the current Call. If the Call is directly triggered by a Call Transaction, this is the Method field of the Transaction's Command Input. If it is an Internal Call, this is the method passed into `call_action` or `call_view`. If it is an Init Call in a Deploy Transaction, this is an empty vector.

`fn arguments(arguments_ptr_ptr: u32) -> u32`

Gets the Arguments of the current Call. If the Call is directly triggered by a Call Transaction, this is the Arguments field of the Transaction's Command Input. If it is an Internal Call, this is the arguments passed into `call_action` or `call_view`. If it is an Init Call in a Deploy Transaction, this is an empty vector.

`fn amount(amount_ptr_ptr: u32) -> u64`

Gets the number of Grays transferred into the current Account by the current Call.

`fn in_internal_call() -> i32`

Returns whether the current Call is an Internal Call.

`fn transaction_hash(hash_ptr_ptr: u32) -> u32`

Get the Hash of the Transaction at the start of the current Call Chain.

`fn prev_block_hash(hash_ptr_ptr: u32) -> u32`

Get the Hash field of the previous Block. Calling this function in a View Call causes a panic.

5.1.4 Internal Call Triggers

`fn call_action(address_ptr: u32, method_ptr: u32, method_len: u32, arguments_ptr: u32, arguments_len: u32, amount: u64, return_val_ptr_ptr: u32) -> u32`

Triggers an Action Call targeting the specified Contract Account, passing in the provided method and arguments and Transferring the specified number of Grays. Then, returns the Call's return value, if any.

`fn call_view(address_ptr: u32, method_ptr: u32, method_len: u32, arguments_ptr: u32, arguments_len: u32, return_val_ptr_ptr: u32) -> u32`

Triggers a View Call targeting the specified Contract Account, passing in the provided methods and arguments. Then, returns the Call's return value, if any.

`fn transfer(address_ptr: u32, amount: u64)`

Transfers the specified number of Grays to a specified Address, or the current Account's entire Balance, if it is smaller than the specified number.

5.1.5 Logging

`log(log_ptr: u32, log_len: u32)`

Add a Log to the current Transaction's Receipt.

5.1.6 Cryptographic Operations

`fn sha256(input_ptr: *const u8, input_len: u32, digest_ptr_ptr: *const u32)`

Computes the SHA256 digest of arbitrary input.

`fn keccak256(input_ptr: u32, input_len: u32, digest_ptr_ptr: u32) -> u32`

Computes the Keccak256 digest of arbitrary input.

`fn ripemd(input_ptr: u32, input_len: u32, digest_ptr_ptr: u32) -> u32`

Computes the RIPEMD160 digest of arbitrary input.

`fn verify_ed25519_signature(msg_ptr: u32, msg_len: u32, signature_ptr: u32, address_ptr: u32) -> i32`

Returns whether an Ed25519 signature was produced by a specified by a specified address over some specified message.

5.2 Guest Functions

5.2.1 Call Entrypoints

`fn init()`

`fn action()`

`fn view()`

5.2.2 Memory Allocation

`fn alloc(len: u32) -> u32`

Allocates a contiguous segment in the Contract's Memory, returning the offset to it.

5.3 Opcode Gas Metering

Opcode Family	Opcode Name	Gas Cost
Constants	I32Const	0
	I64Const	0
Type parameteric operators	Drop	2
	Select	3
Flow control	Nop, Unreachable, Else, Loop, If	0
	Br, BrTable, Call, CallIndirect, Return	2
	BrIf	3
Registers	GlobalGet, GlobalSet, LocalGet, LocalSet	3
Reference Types	RefIsNull, RefFunc, RefNull, ReturnCall, ReturnCallIndirect	2
Exception Handling	CatchAll, Throw, Rethrow, Delegate	2
Bulk Memory Operations	ElemDrop, DataDrop,	1
	TableInit	2
	MemoryCopy, MemoryFill, TableCopy, TableFill	3
Memory Operations	I32Load, I64Load, I32Store, I64Store, I32Store8, I32Store16, I32Load8S, I32Load8U, I32Load16S, I32Load16U, I64Load8S, I64Load8U, I64Load16S, I64Load16U, I64Load32S, I64Load32U, I64Store8, I64Store16, I64Store32	3
32 and 64-bit Integer Arithmetic Operations	I32Add, I32Sub, I64Add, I64Sub, I64LtS, I64LtU, I64GtS, I32Eq, I32Ne, I32LtS, I64GtU, I64LeS, I64LeU, I64GeS, I64GeU, I32Eqz, I32LtU, I32GtS, I32GtU, I32LeS, I32LeU, I32GeS, I32GeU, I64Eqz, I64Eq, I64Ne, I32And, I32Or, I32Xor, I64And, I64Or, I64Xor	1
	I32Shl, I32ShrU, I32ShrS, I32Rotl, I32Rotr, I64Shl, I64ShrU, I64ShrS, I64Rotl, I64Rotr	2
	I32Mul, I64Mul	3
	I32DivS, I32DivU, I32RemS, I32RemU, I64DivS, I64DivU, I64RemS, I64RemU	80
	I32Clz, I64Clz	105
Type Casting and Truncation Operations	I32WrapI64, I32Extend8S, I32Extend16S, I64ExtendI32S, I64ExtendI32U, I64Extend8S, I64Extend16S, I64Extend32S	3

5.4 Disallowed Opcodes

Disallowed Opcode Family	**Opcode**	
	Subgroup	**Examples**
Floating Point Operations	F32x	F32Eq, F32Lt, F32Const
	F64x	F64Eq, F64Lt, F64Const
	Casting/Truncation	F32x4ConvertI32x4U, I32x4TruncSatF64x2SZero
SIMD Instructions	I8x	I8x16GtU, I8x16LtS
	I16x	I16x8GeS, I16x8GtU
	I32x	I32x4MinU, I32x4MaxS
	I64x	I64x2Sub, I64x2Mul
	V128x	V128Const, V128Load8x8S
Atomic Memory Instructions	Memory	AtomicFence, I32AtomicLoad
	Read-Write-Modify	I64AtomicRmwAdd
	Compare and Exchange	I32AtomicRmw8XchgU

6 Constants

6.1 Gas Metering-related

6.1.1 Storage-related

- $BlockWritePerByteCost$: 30.
- $GetCodeDiscount$: 50
- $StorageHashComputePerNibbleCost$: 55.
- $StorageReadPerByteCost$: 100.
- $StorageRefundProportion$: 50
- $StorageTrieTraversePerNibbleCost$: 10.
- $StorageWritePerByteCost$: 1,250.

$\boldsymbol{Cost_{StateSet}}$

The first case in the cost formula corresponds to the creation of a new Storage tuple, the second case to the updating of an existing Storage tuple, the third to the destruction of an existing Storage tuple, and the fourth to a no-op:

$$Cost_{StateSet}(k,a,b) = Cost_{get}(k) + \begin{cases} [(k+b+W)*X] + [2*k*Y] & \text{if } a=0,\ b>0, \\ [b*X] - [Z*a*X] + [2*k*Y] & \text{if } a>0,\ b>0, \\ -[Z*(k+b+W)*X] + [2*k*Y] & \text{if } a>0,\ b=0, \\ 0 & \text{if } a=0,\ b=0, \end{cases}$$

where k is the length of the key being set, a is the length of the old value (which could be 0), b is the length of the new value, W is $LeafNodeBaseLength$, X is $StorageWritePerByteCost$, Y is $StorageHashComputePerNibbleCost + StorageTrieTraversePerNibbleCost$, and Z is $StorageRefundProportion$.

$\boldsymbol{Cost_{StateGet,Warmness[,Contract?]}}$

$Cost_{StateGet,Cold}(k,a) = a * StorageReadPerByteCost + [2*k*StorageTrieTraversePerNibbleCost]$

where k is the length of the key being got, and a is the length of the value that is gotten.

If the key has been get or set previously in the Transaction, execution, the cost is instead:

$$Cost_{StateGet,Warm}(k, a) = Cost_{StateGet,Cold}(k, a) \times \frac{1}{5}$$

If the get is for Contract code, the cost is instead:

$$Cost_{StateGet,Warm,Contract}(k, a) = Cost_{StateGet,Cold}(k, a) \times \frac{1}{4}$$

$\boldsymbol{Cost_{StateContains}}$

$$Cost_{StateContains,Cold}(k) = Cost_{StateGet,Cold}(k, 0)$$

$\boldsymbol{AccountStateKeyLength}$

33.

$\boldsymbol{LeafNodeBaseLength}$

150.

$\boldsymbol{ReceiptBaseSize}$

13.

$\boldsymbol{TransactionBaseSize}$

160.

$\boldsymbol{TransactionBaseCost(txn)}$

$$(13 + 160 + len(txn.command)) \times BlockWritePerByteCost + 4 \times Cost_{StateRead,Cold}(AccountStateKeyLength, 4) + 2 \times Cost_{StateRead,Warm}(AccountStateKeyLength, 4) + 4 \times Cost_{StateWrite}(AccountStateKeyLength, 4, 4)$$

6.1.2 Computation

- $Ed25519VerifyCost = 14$.
- $Keccak256PerByteCost = 16$.
- $LogicalOrPer64BitsCost$: 1.
- $Ripemd160PerByteCost = 16$.
- $Sha256PerByteCost = 16$.
- $WasmBytecodeCompilePerByteCost$: 100.
- $WasmMemoryWritePer64BitsCost$: 3.
- $WasmMemoryReadPer64BitsCost$: 3.

$\boldsymbol{Cost_{Instantiate,Ver,Warmness}}$

$$Cost_{Instantiate,0,Warm}(l) = l \times WasmBytecodeCompilePerByteCost$$

$$Cost_{Instantiate,0,Warm}(l) = Cost_{Instantiate,0,Cold}(l)\frac{1}{5}\times$$

6.2 Economics-related

- $BlocksPerEpoch$: 8,640 (1 epoch will take approximately one day).
- $MaxStakesPerPool$: 1024 (2^{10}).
- $MaxValidatorSetSize$: 64 (2^6).
- $SlashSize$: 20%.
- $StakeReleaseDelay$: 2.
- $TreasuryCutOfBaseFee$: 20%.

Issuance

Issuance during Epoch $n < 3650$ is specified by the below formula:

$$Issuance_n = \frac{0.08 \times 0.85^{\frac{n}{365}}}{}$$

and then drops to a constant $Issuance_n = \frac{0.015}{365}$ for $n >= 3650$

6.3 Workload Limits

- $BlockSizeLimit$: 2,097,152 (2 Megabytes).
- $TargetBlockTime$: 10 seconds.
- $TargetGasUsed$: 35,000,000.

7 Protocol Evolution

A feature that makes public blockchains unique as compute and storage platforms are their communities' commitment (or at least aspiration) to maintain their respective chains' histories indefinitely. The ParallelChain Mainnet shares these aspirations. As time goes on and as the problem domain evolves and the contributors' understanding of it improves, The Protocol will have to be updated. Some of these updates ('major version updates') change the Block format or the state transition function in ways that prevent software implementing the previous version of the Protocol from accepting and/or voting on new Blocks, while others ('minor version updates') do not. Major version updates and minor version updates correspond to 'hard forks' and 'soft forks' in the terminology popularized by the Bitcoin and Ethereum communities.

Fullnode software must be able to execute any Block that is valid according to the rules specified by an accepted version of The Protocol. After a new version of The Protocol gets accepted, the community decides on a Block Height above which the new version of The Protocol will apply. Later, the once-new version of The Protocol is in turn replaced by a newer version of The Protocol. This process maps every version of The Protocol with a range of Block Heights. The end of the current version of The Protocol's range is defined after the next version of The Protocol is approved.

Version Number (major.minor)	Version name	CBI Version	Starting Block Height
1.0	Adam	0	0

A Referenced HotStuff-rs Concepts

- The configuration variable `target_block_time`.
- The `App` trait and its two methods:
  - `propose_block(ProposeBlockRequest) -> ProposeBlockResponse`

  - `validate_block(ValidateBlockRequest) -> ValidateBlockResponse`

- and the relevant Request and Response structs:
  - `ProposeBlockRequest`
    - `storage`
    - `validator_set`
    - `proposer_addr`
    - `view_number`
    - `deadline`
  - `ProposeBlockResponse`
    - `data`
    - `data_hash`
    - `storage_updates`
    - `validator_set_updates`
  - `ValidateBlockRequest`
    - `storage`
    - `block`
    - `validator_set`
    - `proposer_addr`
    - `view_number`
    - `deadline`
  - `ValidateBlockResponse`
    - `error`
    - `storage_updates`
    - `validator_set_updates`
- The `hotstuff_rs::Block` type, which has fields:
  - `hash`
  - `height`
  - `justify`
  - `data_hash`
  - `data`
- The `Vote` variant of the `ConsensusMsg` enum type, which has fields:
  - `view_number`
  - `block_hash`
  - `phase`
  - `signature`
- The `QuorumCertificate` type.