PARSEC: Executing Smart Contracts in Parallel

James Lovejoy
Federal Reserve Bank of Boston

Anders Brownworth
Federal Reserve Bank of Boston

Madars Virza
MIT Media Lab

Neha Narula
MIT Media Lab

Abstract

This paper presents PARSEC, a high-performance distributed platform for executing smart contracts at scale. Our design is generic and supports a wide class of smart contract runtimes. We implement and test two such runtimes: an expressive Lua-based runtime that we design, as well as one integrating the Ethereum Virtual Machine (EVM). Our EVM integration is a drop-in RPC replacement compatible with standard tooling.

PARSEC achieves linear scalability for non-conflicting workloads, and our evaluation shows up to 118K ERC-20 transactions per second on 128 hosts. We achieve this by observing that smart contract semantics do not require materializing a linear transaction history: a more permissive form of ordering, serializability, suffices and supports a parallel implementation.

This work is part of Project Hamilton, and extends our previous work on high-performance, centralized transaction processors for digital currency.

1 Introduction

Smart contracts are computer programs that enable multiple parties to agree on terms of execution, and then outsource that execution to a third party like a blockchain or external provider. There are millions of smart contracts [6] in use by tens of millions of users on Ethereum, a popular platform that uses the Ethereum Virtual Machine, or EVM, for contract execution. Beyond cryptocurrencies, several companies are experimenting with permissioned blockchains that support smart contracts to streamline business processes [11, 14].

The typical method of executing smart contracts is via a *replicated state machine* model—many servers, or nodes, execute every step of every smart contract to apply to their local state. This has several downsides: First, it does not scale well; adding more servers does not improve the throughput of execution. Second, this limits the computation model: contracts must be deterministic, and cannot directly access external services or randomness. Finally, this means that contracts and contract execution is public to all users in the network, barring additional techniques that improve privacy but come with a cost.

This work introduces PARSEC (Parallel ARchitecture for Scalably Executing smart Contracts), a platform for executing smart contracts at scale. PARSEC achieves linear scalability for non-conflicting workloads. To achieve high performance, we make the following observations:

- First, many proposed use cases for smart contracts, like central bank digital currency or securities settlement, do not require decentralization or validation for contract execution; they either already rely on a trusted third party to maintain the integrity of the system, or, because the set of involved parties are known and regulated institutions, they can use auditing or legal and contractual agreements to ensure non-malicious behavior. Even if this fails, for example in the event of software bugs, they can collaborate to rollback or correct mistakes.
- Second, correct smart contract execution does not require execution of smart contracts in a linear order; state must simply reflect an ordered execution, and contracts should not interfere with each other.

We exploit these observations in PARSEC to create a new type of smart contract platform that partitions the state across multiple servers, and runs many smart contract execution environments (or *agents*) in parallel. PARSEC has a distributed runtime, described in §4, that can execute different types of smart contract virtual machines. Doing so correctly requires a distributed commit protocol to handle concurrent accesses to conflicting data.

The key idea behind PARSEC is a simple interface between the agent and the distributed state. Every client transaction invoking a contract is executed by an agent running a virtual machine. Though VMs might be written as though they have exclusive access to local state, we turn this access (correctly) into shared access to distributed state via a simple primitive we call `trylock`. The contract's state-changing operations will execute as a distributed database transaction that reads and writes the shared state.

Perhaps surprisingly, we can even run complex execution environments, like the Ethereum Virtual Machine, without any modifications. Many Ethereum smart contracts, like Uniswap [4], do not require any changes at all to work in PARSEC, though some might benefit from rearchitecting to take advantage of parallelism when accessing different pieces of data (described further in §7). This means PARSEC can take advantage of the existing contracts and developer tooling in the Ethereum ecosystem.

We evaluate PARSEC and show that it can achieve 118K transactions per second. It appears its performance scales linearly in the number of shards, so we expect this number to go up with additional servers.

1.1 Continuation of Project Hamilton

This work was completed in 2022 as part of Project Hamilton, which was a multi-year technical research collaboration between the Federal Reserve Bank of Boston and MIT's Digital Currency Initiative.

Previously, we released two architectures for a high-performance, centralized transaction processor for digital currency [16, 17]. Both used the UTXO model, in particular a *UTXO Hash Set* (UHS) for storing unspent funds in the transaction processor. In this work, we wanted to investigate architectures for programmability. While we can support a limited set of programmable functionality in the original UHS-based systems, capabilities are limited. Expanding the programming model of the UHS would be challenging because it inherits the limitations of the UTXO model, and even further limits what kind of shared state is accessible. PARSEC is designed to enable a very generic programming model and experimentation with many different smart contract execution environments. This enables quicker testing of a wide-variety of financial use-cases without requiring changes to the transaction format or core transaction processor.

1.2 Contributions and outline

In summary, the contributions of this work are as follows:

- PARSEC, a centralized platform for executing a wide variety of smart contract virtual machines in parallel
- PARSEC's design, which uses a simple primitive called `trylock` that enables running existing VMs and contracts that are designed for local execution in parallel against distributed state, without requiring much modification,
- An implementation and evaluation which shows that PARSEC can achieve 118K transactions per second with average latency under 1.6 seconds on an ERC-20 transfer workload.[1]

[1] https://github.com/mit-dci/opencbdc-tx/tree/trunk/src/parsec

In §2 and §3 we describe the system model and interface for PARSEC. §4 describes two environments, one for a virtual machine which runs contracts written in Lua, and a second which runs the EVM. In §5 we describe PARSEC's backend and distributed commit protocol. We evaluate PARSEC in §6, and discuss future improvements in §7 and related work in §8. §9 concludes.

2 System Model

PARSEC is a distributed platform for running contracts. Figure 1 shows the high-level architecture of our system. A *client* engages with the system via a *wallet*, which is software that might run on a mobile device or computer. Wallets issue *transactions* which are processed by the system.

The system is composed of two layers. First, a distributed runtime called the *agent* which processes transactions from users by executing contracts, which run inside virtual machines. A contract's input is a client-specified byte string; contract interprets this as a transaction, i.e. a set of parameters that represent an execution instruction.

Second, a database which stores persistent shared state for the virtual machines and contracts, providing concurrency control and atomic database transactions. That database allows the agent to execute transactions and contracts which might read and write the same data in parallel, while guaranteeing isolation and atomicity.

We assume that a centralized, trusted entity operates the system on behalf of users. Neither the agent nor the database are required to tolerate Byzantine faults, and the agent might fail at any point during transaction execution. The database is responsible for implementing safe recovery in the event that an agent fails. For scalability, there might be multiple distinct agents processing transactions in parallel.

We assume that the database is a generic key/value store. It is not required to provide permissioning for contracts, type checking, a query language, or other schema-like features. The runtime is responsible for protecting the data store from tampering outside the semantics defined in contracts. Therefore, wallets always interact with the system via an agent to enforce contract semantics and the consistency of any returned values. One could use an existing ACID-compliant database such as PostgreSQL or CockroachDB for this purpose, but PARSEC does not require most of the features provided by those systems. Instead we implemented our own scalable, distributed, transactional key-value database with a simple interface for the agent, described in §3.

3 Data structures and abstractions

When an agent needs to access or modify system state, it interacts with the *broker*. In PARSEC, the broker exposes

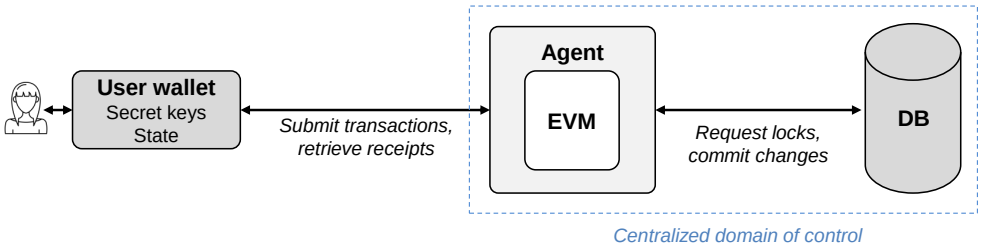

Figure 1: High-level system architecture

a clean, high-level interface that is compatible with implementing many kinds of containers in the agent; we use the same broker-agent interface, with no modifications, to support both the EVM and our Lua-based VM, the LVM.

In turn, the broker is tasked with translating an agent's high-level requests into database transactions to be executed by the database (see Figure 5). The broker is responsible for providing an atomic interface with the state and failure recovery. We later explain the underlying distributed commitment protocol in §5.2.

In detail, the broker exposes the following data structure and abstract operations to the agent:

Key-value store. The single data structure exposed by the broker is a key-value store mapping byte arrays to byte arrays. In practice, keys in this data structure are serialized account public keys, and the mapped values are associated smart contract data, e.g., amount of currency held by an account, smart contract bytecode or internal state.

The `trylock` abstraction. The key-value store is initially empty and accessed via an atomic locking interface we call `trylock`. The `trylock` interface gives the agent a way to realize transactional access to the key-value store as follows. Take the key-value store at any time T; To access it, the agent will first make a `begin()` call which initializes a database transaction between the agent and broker (see Step 1 in Figure 2) . Inside the agent, `begin()` might or might not be exposed to the smart contract code; in our agents, this call is implicit and occurs before the first access to the state.

Afterwards, the agent will make a number of `trylock` calls (see Step 2 in Figure 2). Each of these specifies a key to be read and locked, and the agent responds to each call with a value currently held by the state at time T. Importantly, keys (account and contract addresses and contract state) accessed via `trylock` can (and, in practice, will) depend on results of previous `trylock` calls. The agent can use `trylock` to upgrade a previously acquired read lock to a write.

Finally, the agent responds with a `commit` call optionally specifying new values for keys that this transaction holds write locks for (see Step 3 in Figure 2). The broker is responsible for atomically either updating all these keys with new values or, in case of conflicts, aborting and leaving the state unchanged.

4 Distributed Runtimes

In this section we describe the two distributed runtimes we implemented in the agent. We implemented a generic environment that executes Lua programs and an example contract which provides account-balance payments. We also implemented Ethereum transaction semantics, providing a drop-in replacement for Geth/Infura, compatible with existing Ethereum tools.

4.1 LVM: a minimal smart contract VM

We now describe LVM, a new, minimal smart contract environment. LVM smart contracts use the account-balance data model and are written in Lua, a lightweight embedded scripting language and virtual machine.

In this section, we first describe the data structures LVM contracts access and use, the LVM transaction format, and the semantics of their execution. We then explain the notion of a "root contract," essential to building systems in our environment, and analyze properties of LVM smart contract model. We finish with a detailed description of a simple LVM contract.

4.1.1 Data structures

The shared state of our smart contract environment is a global key-value map M in which both keys and values are arbitrary byte arrays. LVM uses this data structure for both storing smart contract bytecode and smart contract state, as follows:

- **Smart contract bytecode.** Each LVM smart contract is identified by its 32-byte address, chosen at the time of contract deployment. Contracts are stored in the key-value map in the usual way: a contract whose 32-byte address is C, has its bytecode stored at $M[C]$.

- **Smart contract state.** LVM smart contracts additionally use the key-value map for storing their internal state. By convention, a contract with address C stores its internal state variable v by using a global

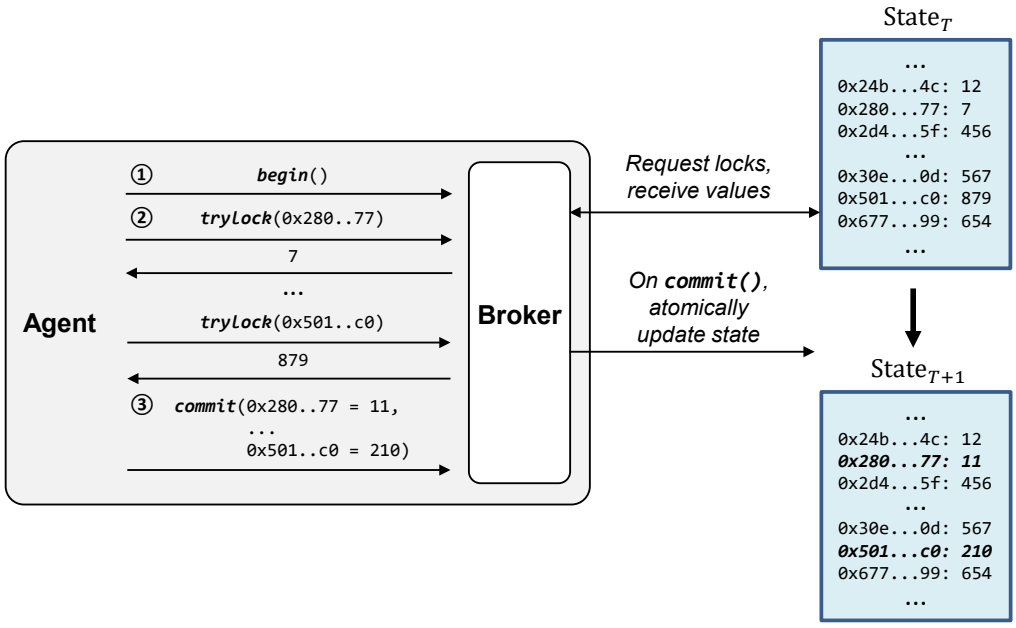

Figure 2: Interface between agent and broker

map's key that is a concatenation of C and the variable's name, i.e., at $M[C \| \text{“v”}]$.

It is important to note that the aforementioned namespacing convention is not enforced at the execution environment layer but rather is a responsibility of the smart contracts deployed in the system. Our system enforces this separation and other security through a flexible and generic mechanism, which is itself based on smart contracts. Namely, the system is initialized with a single smart contract we call the "root contract," and this root contract is given the necessary tools to enforce desired system security properties. We explain this in §4.1.3.

4.1.2 Transaction semantics

A transaction $\mathsf{tx} := (C, \mathsf{p})$ comprises two parts: a 32-byte address C of the contract to be executed, and transaction payload p. Each LVM contract is a single-argument Lua function and has the flexibility and responsibility to interpret p according to its self-defined ABI.

To execute a LVM transaction, the smart contract environment will first look up the contract bytecode B in the key-value map, $B := M[C]$. If this look-up is successful (i.e., the smart contract exists), LVM executes the Lua bytecode on the described payload, i.e., computes $B(\mathsf{p})$. To enable the contract to interact with the system state, LVM agent exposes our `trylock` abstraction (§3) to the contract and uses contract's return value to update the system state, as we now explain.

Each transaction executes within its own coroutine and the contract must yield to the agent to acquire locks on keys for shared data. Whenever a contract calls the `coroutine.yield` function provided by the Lua environment, the contract yields its execution to the agent. The agent interprets the single `coroutine.yield` argument as a key, and calls `trylock` to obtain a write lock on the requested key and resumes execution of the contract once the lock is acquired. The return value of `coroutine.yield`(x) inside the contract is the value $M[x]$ that agent obtained in its `trylock` call.

Finally, the contract's return value is an update map U. This Lua table maps keys k, each of which the contract has previously requested a lock for, to their new values v. The LVM agent uses U in its `commit` call to the broker, requesting an atomic update that sets all $M[k] \leftarrow v$. If the broker confirms `commit`, this results in a successfully executed transaction and a failure otherwise.

4.1.3 Root contract

In LVM authentication, access control, resource limits, and other security properties are a responsibility of smart contracts deployed in the system. This is in contrast to other smart contract systems where these properties are enforced by the execution environment[2] We design our smart contract security perimeter by leveraging Lua's ability to execute a function's bytecode while simultaneously overriding the function's environment and introspecting its execution, as we now describe.

In our design, the key-value store is initialized with

[2]For example, EVM transaction semantics require transfers to bear digital signatures, execution semantics forbid a contract from directly accessing another contract's storage, and gas requirements provide a protection against resource exhaustion.

a single trusted root contract R, and all other contracts are deployed and executed through the root contract. In fact, the root contract will enforce that only the 32-byte long key (i.e. the smart contract address) in M is that of R itself. This way, the only transactions accepted by the execution environment are those calling R, and all user smart contract transactions need to be proxied through R.

The root contract namespaces user smart contracts during deployment and execution. When deploying a contract with address C, R stores its bytecode at $M[\texttt{contract\_}C]$. To execute a user contract C with payload p, one submits a transaction $\mathsf{tx} := (R, (\texttt{call}, C, \mathsf{p}))$, i.e., calling R with payload $(\texttt{call}, C, \mathsf{p})$. The root contract then provides an isolated execution of C as follows:

1. R calls `coroutine.yield("contract_" + C)` to look up C's bytecode.
2. R uses Lua built-ins `load`, `setmetatable`, and `setfenv` to obtain a Lua function f matching C's bytecode for which the lexical environment is overridden to wrap `coroutine.yield` (see below).
3. R calls $f(\mathsf{p})$, executing the sandboxed smart contract on the payload p. When f returns its update map U, R returns a sanitized update map U' to the execution environment.

The above isolated execution template is generic and can support a variety of security mechanisms. For example, to completely isolate contracts from each other, R can wrap `coroutine.yield(`k`)` to prefix keys with the contract's address, so that C's access to $M[k]$ is always redirected to $M[\texttt{storage\_}C\_k]$; the corresponding update map sanitization would simply prefix all keys with "`storage_`C`_`".

In addition to overriding the user contracts' interface to `trylock`, the root contract can also enforce execution time limits through Lua's `debug.sethook` interface.[3] That way, the root contract would get a callback every N instructions and could implement a simple version of "gas". Finally, the root contract can also provide additional convenience functions to user contracts, such as authenticated access to shared state, or digital signature authorization of transactions.

We note that, while expressive and generic, root contract-based deployments are not the only ones possible in our system. For example, the system could be initialized with a small number of trusted contracts (e.g., the self-contained account-balance transfer contract in Section 4.1.5) and no "wrapping" root contract. Such usage would benefit from our efficient runtime and the ability to express payment flows in a high-level language, as well as simplify security audits, as it avoids reasoning about complex interactions between different smart contracts.

[3] A complete resource control and sandboxing solution would necessarily implement parts of it in native code. For example, the orchestration infrastructure could isolate an agent in a Linux control group (cgroup), and apply CPU, memory, I/O, and other limits.

4.1.4 Properties of LVM environment

We now briefly remark on a number of features that our execution environment possesses.

Native parallelism. To support the efficient parallel execution of transactions within an agent, LVM leverages Lua coroutines. Since obtaining data from shards can take multiple milliseconds, this allows the agent to schedule other transactions without requiring multithreading, and makes the agent capable of processing a greater number of transactions in parallel in a single thread.

Upgradeability. In our design, many security features are meant to be implemented through an appropriate root contract, as execution environment enforces minimal transaction semantics. As security properties are expressed as Lua smart contract checks, these security features can be upgraded (as smart contract upgrades) while the system is running, without making any changes to the underlying execution environment (i.e., no agent code needs to change).

Extensibility. While our contracts are written in Lua, they are not limited to using Lua-only code. An agent can be linked with highly-optimized compiled versions of common primitives (e.g., cryptographic operations like hashing or digital signature verification) and expose them to Lua contracts as "precompiles." Looking ahead, our example contract relies on such a precompile: `check_sig` wraps a call inside the efficient `secp256k1` library [8] which is written in C.

Interoperability. Finally, our design is not limited to contracts written in Lua: it would work equally well with smart contracts written in any other language through the agent appropriately exposing our `trylock` interface. An agent could support multiple smart contract languages in parallel and dynamically dispatching each smart contract to its interpreter. In fact, this design could also support contracts written in native code, and sandboxed via a system like NaCl [23].

4.1.5 An example contract: payments

In this section, we describe an example Lua account-balance contract which implements Ethereum-style account-balance transactions. The contract performs the necessary operations to atomically update account balances while preventing unauthorized payments and double-spends.

Recall that transactions enter our system via agent's TCP endpoint that wallets connect to and submit transactions for processing. The agent then calls the contract with the user-provided payment parameters, and returns

```lua
function (param)
    from, to, value, sequence, sig = string.unpack("c32 c32 I8 I8
       ↪    c64", param)

    function get_account_key(name)
        account_prefix = "account_"
        account_key = account_prefix .. name
        return account_key
    end

    function get_account(name)
        account_key = get_account_key(name)
        account_data = coroutine.yield(account_key)
        if string.len(account_data) > 0 then
            account_balance, account_sequence
                = string.unpack("I8 I8", account_data)
            return account_balance, account_sequence
        end
        return 0, 0
    end

    function pack_account(updates, name, balance, seq)
        updates[get_account_key(name)] = string.pack("I8 I8",
           ↪    balance, seq)
    end

    function update_accounts(from_acc, from_bal, from_seq, to_acc,
       ↪    to_bal, to_seq)
        ret = {}
        pack_account(ret, from_acc, from_bal, from_seq)
        if to_acc ~= nil then
            pack_account(ret, to_acc, to_bal, to_seq)
        end
        return ret
    end

    function sig_payload(to_acc, value, seq)
        return string.pack("c32 I8 I8", to_acc, value, seq)
    end

    from_balance, from_seq = get_account(from)
    payload = sig_payload(to, value, sequence)
    check_sig(from, sig, payload)
    if sequence ~= from_seq then
        error("incorrect sequence number")
    end

    if value > from_balance then
        error("insufficient balance")
    end

    if value > 0 then
        to_balance, to_seq = get_account(to)
        to_balance = to_balance + value
        from_balance = from_balance - value
    else
        error("value must be positive")
    end

    from_seq = sequence + 1
    return update_accounts(from, from_balance, from_seq, to,
       ↪    to_balance, to_seq)
end
```

Figure 3: Lua code implementing Ethereum-style account-balance transactions.

either a success status or error message to the user via the socket once execution has completed.

We now provide a step-by-step explanation of how the contract works, emphasizing the locking mechanism, security measures, and support for parallel execution.

The contract begins by unpacking the parameters (line 2), including the sender's public key, recipient's public key, amount, sequence number, and signature. It uses the `string.unpack` function to extract the necessary values from the byte string.

To support the efficient parallel execution of transactions within an agent, the Lua environment leverages coroutines. Each transaction executes within its own coroutine and the contract must yield to the agent to acquire locks on keys for shared data. Since obtaining data from shards can take multiple milliseconds, this allows the agent to schedule other transactions without requiring multithreading, which makes the agent capable of processing a greater number of transactions in parallel.

The contract retrieves the sender and receiver's account data using the `coroutine.yield` function (on lines 38 and 50 respectively), provided by the Lua environment. This function yields execution to the agent which calls `trylock` to obtain a write lock on the requested key and resumes execution of the contract once the lock is acquired. It unpacks the raw bytes returned by the function into the balance and sequence number for the account (lines 14 and 15). If the destination account does not exist, the deserialization process detects this and returns a new account (lines 13 and 18).

To ensure secure payments, the contract employs several measures. First, the `check_sig` function, provided by the Lua environment, verifies the cryptographic signature of the transaction payload against the sender's public key, ensuring its authenticity (line 40). Second, it checks that the sequence number in the transaction matches the sender account's current sequence number, and increments the sequence number if the transaction completes (lines 41 and 57). This prevents replay attacks where an attacker resubmits an old transaction. Finally, the contract checks if the sender has sufficient balance to perform the transaction, preventing balance attacks where a sender attempts to spend more than their available balance (line 45). If the contract raises an error, the transaction execution is aborted and the agent releases any locks acquired by the contract.

If the checks above succeed, the contract updates the sender and receiver account balances (lines 51-52), and packs the account data back into a byte string (lines 25-32). At the end of execution, the contract returns a map of updated account keys and values (line 58). The agent writes the updates back to the shards as part of an atomic commit, ensuring the integrity of the account data. Since the system provides deadlock resolution via preemption, the contract can be interrupted by any lock or after the contract returns, and the agent will have to restart it from scratch.

4.2 EVM

We also implemented an agent that supports EVM-compatible transactions in our parallel execution environment. Our implementation provides an Ethereum transaction interface, and is a drop-in RPC replacement compatible with many existing Ethereum tools, such as Hardhat and Truffle.

Our implementation realizes the Ethereum transaction semantics and consists of two parts. First, it uses a library implementing the Ethereum Virtual Machine; in our agent we used evmone [1]. Second, to integrate the virtual machine, we implement the EVM host interface using `trylock` and provide the required system calls for the EVM, described later in this subsection.

The agent exposes an HTTP JSON-RPC endpoint

through which users interact with the system that provides the subset of the Geth/Infura API required for processing transactions. The agent does not provide auxiliary functions provided by Geth, such as wallet, debugging, or contract development RPCs. The HTTP endpoint handles RPC requests from users and dispatches the relevant call to the host interface, which might invoke an instance of the EVM, depending on the operation.

As mentioned above, integrating EVM requires implementing the EVM host interface to provide several system calls specific to Ethereum's transaction semantics. Our host interface implementation provides these methods using our atomic commitment protocol. We devote the rest of this section to a detailed description of our host interface and its integration in our agent.

Setup. When the agent receives a raw Ethereum transactions submitted via `eth_sendRawTransaction` it executes the transaction as follows. The agent creates a new instance of the host interface for each transaction containing specific context information such as the timestamp, gas limit and origin account. Agent uses a thread pool to execute transactions in parallel. This is because the EVM implementation we used does not support cooperative multitasking, and so each of the host interface methods are blocking. During the execution the agent also caches keys that have been locked or modified and provides a map of updated keys and values to be committed once execution has completed.

Execution. After setting up the host interface, the agent executes the transaction as follows. First, the agent deserializes raw transaction and checks its signature. It then acquires a write lock on the origin account metadata and checks the current account nonce matches the one cited in the transaction. It locks a key for the transaction receipt, before constructing an `evm_message` and calling `call` (described below) on the host interface instance created for the transaction. The agent waits for `call` to return, retrieves the map of any keys that have been modified, generates an Ethereum transaction receipt and commits the changes and receipt before responding to the user. The user can retrieve the transaction receipt via the standard `eth_getTransactionReceipt` RPC.

Host interface. Our implementations of the necessary host interface APIs (see Figure 4) work as follows:

- `call` implements the core Ethereum transaction semantics including charging gas, transferring funds between accounts, deploying new contracts, and invoking the EVM if necessary. `call` uses `get_code_size` to determine if the destination account is a contract and creates an instance of the EVM to call with the bytecode retrieved via `copy_code`. If the transaction is deploying a new contract, `call` will invoke the EVM which returns the contract bytecode, and create a new account with the bytecode.[4]

- `account_exists`, `get_storage`, `get_balance`, `get_code_size`, `get_code_hash` and `copy_code` are implemented using `trylock` either by acquiring a new read lock on the key where the relevant information is stored, or retrieving from a local cache if the data has previously been referenced. Our implementation uses key prefixes to separate account metadata, contract bytecode and account storage elements in the shared state.

- `access_account` and `access_storage` track whether accounts or storage elements have previous been accessed by a contract, which affects the gas cost for some operations in the EVM.

- `set_storage` and `selfdestruct` acquire write locks on the relevant data and cache the changes locally, to be committed later by the agent.

- Since there are no blocks in our system, `get_block_hash` always returns zero.

```
account_exists(address) -> bool
get_storage(address, key) -> bytes32
set_storage(address, key, value) -> storage_status
get_balance(address) -> uint256
get_code_size(address) -> uint
get_code_hash(address) -> bytes32
copy_code(address, code_offset, buffer_data*, buffer_size) -> uint
selfdestruct(address, beneficiary_address)
call(evm_message) -> evm_result
get_tx_context() -> evm_tx_context
get_block_hash(height) -> bytes32
emit_log(address, data*, data_size, topics[], topics_count)
access_account(address) -> access_status
access_storage(address, key) -> access_status
```

Figure 4: The EVM host interface

4.3 Discussion

4.3.1 Non-deterministic runtimes

Since the agent and broker do not have to be replicated, the distributed runtime does not have to be deterministic. Runtimes can take non-deterministic actions like generating random numbers or referencing data from external sources like the Internet. Furthermore, runtime execution is not public, meaning that applications can store secrets, like private keys, as long as they trust the system operator. Secret storage and random number generation could be handled by HSMs to help guard against side-channel attacks. This allows for a richer set of program semantics for applications executing in the runtime. For example, a contract could sign transactions for other systems. This might eliminate the need for third-party bridges to implement interoperability with other assets.

[4]Contract deployment transactions in Ethereum do not contain the new account bytecode directly, but rather a generator program which is executed in the EVM which returns the new bytecode to be deployed.

4.3.2 Inter-runtime communication

Since our backend is a generic database, multiple distributed runtimes with distinct semantics can operate simultaneously, with serialization enforced by the database. For example, one could deploy the EVM environment at the same time as an implementation of the Bitcoin data model and transaction format. Each agent implementation could use key prefixes to separate the state for each runtime environment. Both environments could provide a method to transfer funds between Bitcoin-like UTXOs and Ethereum-like account balances, within the transaction semantics of each environment. This is useful because different data models are better suited for certain applications, and the system operator is not required to choose a single runtime.

4.3.3 Runtime sandboxing

The database does not provide permissions for different elements of the database and relies on the runtime environments to isolate user-provided applications. In the Ethereum environment, sandboxing is provided by the semantics of the EVM; contracts can only access their own storage, and interaction between contracts is only possible via their respective ABIs. Futhermore, the Ethereum environment provides resource limiting via gas, and the EVM does not allow contracts to interact with the host system. In contrast, generic VMs like Lua, Javascript or Python do not provide any sandboxing by default. The system operator would have to either closely audit VMs, or implement their own resource limits, isolation, and sandboxing for each VM.

5 Distributed transactional key/value store

Many contracts can be written so that transactions access independent keys; therefore we'd expect to see a performance benefit from partitioning the datastore across many servers. Doing so safely requires a distributed commit protocol so that we can execute many transactions and contracts that might read or write the same data in parallel.

There are many existing systems that provide distributed transactions on a sharded key/value store. We do not evaluate them in this work. We implemented our own and describe how it works here. Any such system should support distributed transactions and client failure recovery. It should not assume keys are known ahead of time, so it would need to provide deadlock detection. In addition, it should provide sharding to scale non-conflicting workloads. One could use a strongly consistent database like PostgreSQL or CockroachDB, instead of our custom backend, to provide the functionality for `trylock`. An important piece of future work is evaluating the performance of our approach versus existing systems.

5.1 Overview

Agents interact with the state through a broker, as described in §3. This section discusses how we implement `trylock`. The state is stored in a key/value database, partitioned across a set of *locking shards*. Locking shards implement a set of functionality to safely and durably process distributed transactions. A broker uses a directory service to locate the appropriate shard for a given key; we do not discuss the implementation of this service, but assume it is correct and available [9, 13]. A broker communicates with a replicated *ticket machine* to obtain monotonically increasing sequence numbers to assign to distributed transactions. A broker and shards use tickets to uniquely identify distributed transactions when providing concurrency control. Tickets are used for recovery and prioritization in deadlock resolution between conflicting transactions. Figure 5 shows an overview of the architecture with an agent that is running the EVM.

5.2 Executing database transactions

The database transaction commit protocol uses two-phase locking (2PL) to access keys. A broker coordinates the protocol. Figure 6 shows the steps of the commit protocol. At some point during transaction execution, the agent calls `begin()` in its broker. The broker requests a ticket from the ticket machine and assigns the ticket number as the transaction ID for the database transaction.

As the agent executes a contract (described in §4), it will use `trylock` to read and write keys. For each read or write, the broker takes out read and write locks on the appropriate shards (found through the directory service) using the database transaction's ticket number. It retrieves keys from the shards for reads and buffers writes locally. Upon receiving a request for a read or write lock, a shard might (1) grant the lock (2) wait because another transaction has a conflicting lock or (3) return *pre-empt* (described below). Shards record locks that are granted along withe ID of the broker that requests them, and the ticket number. Locks can be upgraded from read to write.

After all reads and writes, the agent calls `commit()` and the broker begins the commit protocol, which is two-phase commit. First, it calls `prepare()` on all relevant shards, including all the new values for keys that were written. Assuming all shards respond affirmatively, the broker then calls `commit()` on each shard.

Once a `commit()` is received by any shard, the transaction will (eventually) complete successfully—the shard will release locks and apply writes. Afterwards, the broker calls `discard()` to clean up any state the shards are storing about the database transaction.

The broker might `rollback()` the database transaction if the executing agent or contract fails, or if it does not receive successful responses to `prepare()` from all shards. This releases all locks without applying any

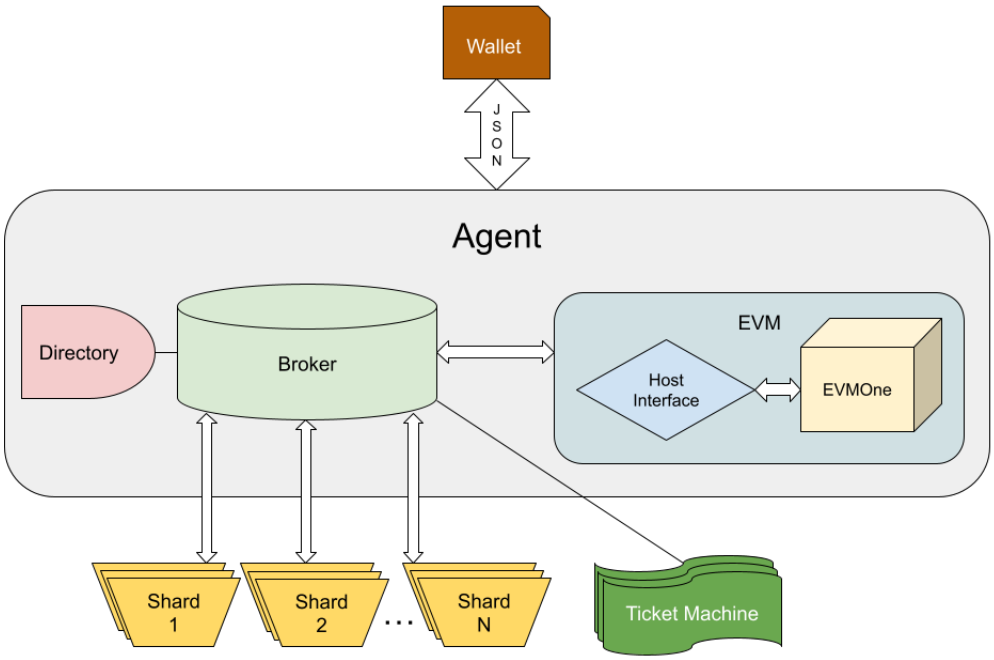

Figure 5: PARSEC Architecture Diagram

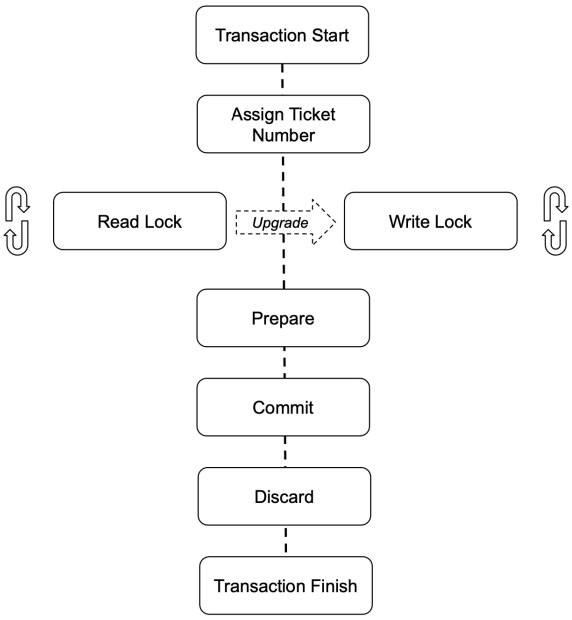

Figure 6: Transaction Lifecycle

writes. The broker still needs to call `discard()` after a `rollback()`.

Optimizations. Agents actually request batches of tickets from the ticket machine, so the broker can request a ticket locally instead of talking to the ticket machine for every database transaction. In addition, `begin()` is implicitly called the first time an agent calls `trylock`.

5.3 Detecting deadlocks

Note that we do not know what keys a transaction or contract might access before it is executed; we also cannot mandate that keys are taken out in a certain order. Since we use 2PL with two-phase commit (2PC), we must have a way of resolving deadlocks. PARSEC uses ticket numbers with the Wound Wait algorithm to resolve deadlocks [21]. Wait Die is more efficient, but less fair; it doesn't guarantee that all transactions will eventually complete if there is heavy lock contention. So if PARSEC used Wait Die, an adversary could starve honest transactions by continually issuing transactions which contend on heavily used keys.

Ticket numbers are used as timestamps in Wound Wait. To recap, if a database transaction with a lower ticket number attempts to acquire an already locked key, the shard will pre-empt the higher-ticket database transaction, which it will discover when its broker either attempts to

call `prepare()` or tries to acquire another lock on the same shard. If a broker discovers its database transaction has been pre-empted, it will `rollback()`; the agent will restart execution of the transaction, using the same ticket number. The shard will not pre-empt a database transaction for which it has already started processing `prepare()`.

5.4 Fault Tolerance

Each shard runs in a replicated state machine to tolerate individual shard server failures; in our implementation we use Raft. The ticket machine is also replicated using Raft. Brokers are responsible for either cleaning up or finishing database transactions that might have been interrupted due to failures: We assume brokers are managed by an orchestration layer which ensures (1) each broker has a unique ID, (2) at most one broker with a given ID is running, and (3) eventually a broker with a given ID will run. Shards store information about database transaction state, but rely on the appropriate broker to either finish committing partially committed transactions, or to rollback failed database transactions that have not yet begun to commit, and clean up state.

When a broker is restarted, it asks all of the shards for information on any outstanding database transactions with its broker ID and their state. For any database transactions that are committed on all relevant shards, it calls `discard()`. For any where at least one shard is committed, it calls `commit()` on the remaining shards. For any other outstanding database transactions it calls `rollback()`.

Note that because brokers are not run in a replicated state machine, and restart any database transactions that have not yet begun to commit, agents can have non-deterministic functionality.

6 Evaluation

For benchmarking we deployed the PARSEC codebase in Amazon Web Services (AWS) on EC2 virtual servers using the c5a.large (2vCPU, 4GB RAM) instance type. We ran the system components in three geographical regions, Virginia (us-east-1), Ohio (us-east-2), and Oregon (us-west-2). Raft cluster leaders, agents and transaction load generators were located in the Virginia region. Shards were replicated by a factor of three, with the followers located in the Ohio and Oregon regions.

We evaluated how uncontended ERC-20 transaction throughput using the EVM scaled when increasing the number of shards. Our workload consisted of payments between a fixed set accounts managed by ERC-20 token contracts. We implemented the ERC-20 token for the benchmark using the OpenZeppelin examples, which provides payments between accounts with a pre-minted balance. Each load generator deployed an ERC-20 contract

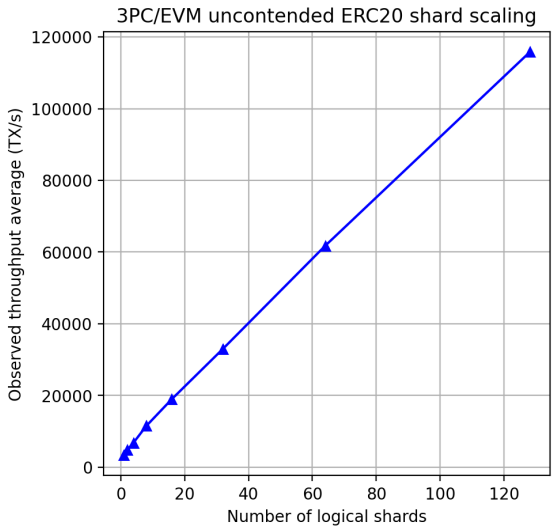

Figure 7: PARSEC Shard Scaling

and generated back and forth payments between 3072 randomly selected accounts in the contract. The load generators pipelined the transactions for parallelism, while ensuring that none of their in-flight transactions involved conflicting accounts. Each load generator was paired with an agent and sent signed and serialized Ethereum transactions calling the ERC-20 contract with the relevant parameters. The load generator waited for the agent to return the transaction receipt, which it used to confirm the transaction completed successfully, and measure the transaction latency.

We performed experiments for configurations of our system with shard cluster counts in powers of 2 from 1 through 128. For each shard count we ran multiple benchmarks, increasing the number of load generators and agents to find the peak average throughput. The load generators recorded the timestamp and settlement delay for each transaction. This data was aggregated and used to calculate the average transaction throughput and the latency distribution for each experiment.

Figure 7 shows how the peak average throughput scales with the number of shard clusters. The plot shows that the peak average transaction throughput scales linearly and that PARSEC is capable of 118K transactions per second with 128 shards. We believe our system would continue to scale with additional shards, as acquiring ticket numbers from the ticket machine should not be a bottleneck due to batching. Intuitively, our uncontended ERC-20 workload scales linearly because balances for different ERC-20 accounts are stored in different keys (even if they use the same contract), the the number of keys locked by each transaction is fixed, and the keys are uniformly distributed across the shards. Our EVM implementation uses read locks to allow multiple transactions using the same ERC-

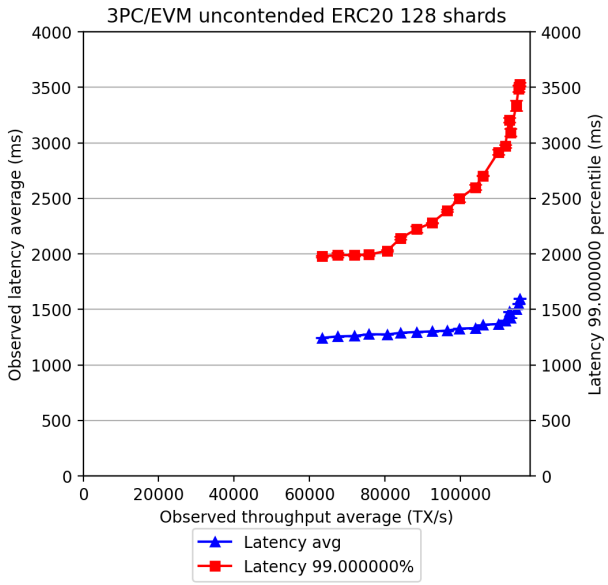

Figure 8: PARSEC ERC-20 Throughput

20 contract to lock the contract code and account metadata keys in parallel. Since the two accounts in a pending transaction are guaranteed by the load generators not to conflict, no transaction is ever pre-empted and forced to retry.

Figure 8 shows how the average transaction throughput compares to the average latency and tail latency for each of the experiments in the 128 shard cluster configuration. Our system shows low average and tail latency in a geo-replicated configuration. The average transaction at peak throughput completed in under 1.6s and tail latency was under 3.5s. The ERC-20 transactions in our workload using the OpenZeppelin implementation serially lock 8 distinct keys during execution. The contract acquires locks on the following keys in order:

1. Sender account metadata (write)
2. Transaction receipt (write)
3. Destination account code (read)
4. Sender ERC-20 account balance (read)
5. Destination account metadata (read)
6. Sender ERC-20 account balance (read→write)
7. Destintaion ERC-20 account balance (read)
8. Destination ERC-20 account balance (read→write)

If the keys locked by a contract call were known ahead of time, latency could be improved by pipelining and locking the keys in parallel rather than blocking for each lock operation. One could also edit the host interface implementation to eagerly take out write locks on EVM storage keys when they are read to avoid two lock upgrades in the above example. This is possible with the existing Ethereum transaction semantics using EIP-2930 [3], but requires extra implementation from the user's wallet, and is not currently implemented by our runtime.

7 Future Work

In this section, we outline potential areas for future research and improvement of our system.

BFT semantics. One avenue for future exploration is the investigation of Byzantine Fault Tolerant (BFT) semantics for our architecture. While our current system operates under Crash Fault Tolerant (CFT) semantics, which assume benign faults, incorporating BFT semantics would enhance the system's resilience against malicious nodes. Research is needed to explore how to modify our system to support BFT semantics while maintaining performance and scalability.

Optimizing smart contracts for more parallel execution. Our system allows for parallel execution of transactions, providing increased scalability and throughput. However, many smart contracts in Ethereum assume serial execution in a blockchain environment, and so aren't designed to benefit from this parallelism: They rely heavily rely on shared state. Further work is needed to improve the scalability of these contracts and determine how these contracts can be redesigned to minimize shared state or utilize batching techniques. Exploring alternative implementations that maximize parallel execution would enhance the overall efficiency of our system by avoiding conflicting transactions where possible.

Enhanced deadlock resolution algorithm. As described in §5, our current implementation employs a simple Wound Wait deadlock resolution algorithm. However, given the high volume of conflicting transactions targeting popular accounts or contracts, the system may benefit from a more sophisticated deadlock resolution algorithm. Future work could focus on designing and implementing an enhanced deadlock resolution mechanism that can handle the scale and complexity of conflicting transactions, particularly in scenarios where smart contracts rely heavily on shared state.

Evaluating existing distributed databases. Another avenue for future research is the evaluation of existing geo-distributed databases as a potential replacement for our custom key-value-based backend. Databases such as Google Spanner, CockroachDB, and YugabyteDB offer transactional SQL interfaces and are specifically designed to provide strong consistency in geo-distributed scenarios.

By substituting our custom backend with a proven geo-distributed database, we can leverage the rich features and optimizations that these databases offer, including built-in transaction management, automatic sharding, and replication. This transition could potentially simplify the overall architecture and reduce the maintenance overhead associated with supporting a custom backend.

8 Related Work

Related work falls into two categories: techniques to scale smart contract execution in a decentralized setting, and distributed runtimes designed to execute applications at scale in a centralized setting.

Scalable smart contracts. The scalability of Ethereum and other decentralized smart contract platforms is limited in part because they rely on a decentralized model where every full node needs to execute and validate all smart contracts, deterministically.

There have been many proposals to address scaling the EVM: one technique is to move transactions and computations off of the main chain and onto a *layer 2*, whether via state channels or rollups [15, 18, 19]. These techniques vary in how much performance improvement they can get, and sometimes change security guarantees to require liveness for correct execution. It can also be more challenging to write smart contracts in this paradigm, especially ones that might cross multiple channels or rollups. There have been many designs to better parallelize the execution of smart contracts within a validating server [20]. These techniques might help improve PARSEC's agent performance, but PARSEC can get scalability with additional servers, which those systems cannot.

Many systems have investigated sharding, either data or contract execution, so that not every node in the system needs to execute every contract. In Hyperledger [7], nodes execute contracts in parallel optimistically, like agents in PARSEC. However, Hyperledger still eventually sequences results through a single consensus instance, which limits scalability and performance. Like PARSEC, systems like Chainspace [5] shard smart contract execution and use a distributed commit protocol to avoid a central sequencing step. They still get much lower performance than PARSEC. Note that these systems all provide a feature PARSEC does not, which is public verifiability.

Distributed runtimes. There is a lot of work in the distributed systems literature which enables applications to run in a distributed setting against shared data [10, 12, 22, 24]. PARSEC takes inspiration from these systems to design a platform for a new use case, executing smart contracts in different types of virtual machines.

9 Conclusion

Generic programming architectures for financial transactions usually operate in decentralized environments. This constraint limits both their scalability and the capabilities the virtual machine can support.

Our research presents a centralized scalable architecture capable of supporting generic virtual machines. We find that operating centrally and sharding the state has significant scalability advantages.

Our open source implementation can run both the Ethereum Virtual Machine as well as smart contracts written in Lua on a scalable datastore in an administratively centralized context. Because we don't have to wait for decentralized consensus, a single execution of a smart contract is enough for validation, creating a more efficient environment. While most workloads should scale linearly, workloads that contend on a low number of keys have limits.

PARSEC is open-source and slated for inclusion in OpenCBDC [2]. This release enables a system which supports scalable testing of a wide variety of financial use cases without modification of the core platform. Policymakers and the general public are now able to experiment with cutting edge "off the shelf" financial applications in a scalable architecture.

10 Acknowledgements

The authors express gratitude to Gert-Jaap Glasbergen, Kevin Karwaski, Alistair Hughes, Michael Maurer, Alex Jung, Sam Stuewe and Rainer Böhme for their help on this project. We are also grateful for the efforts and leadership of Robert Bench and Jim Cunha who made this work possible.

References

[1] evmone – fast Ethereum Virtual Machine implementation. https://github.com/ethereum/evmone.

[2] OpenCBDC: A transaction processor for a hypothetical, general-purpose, central bank digital currency. https://github.com/mit-dci/opencbdc-tx.

[3] Eip 2930, 2020. https://eips.ethereum.org/EIPS/eip-2930.

[4] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson. Uniswap v3 core. https://uniswap.org/whitepaper-v3.pdf.

[5] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[6] Alchemy.com. Ethereum statistics (2022). https://www.alchemy.com/overviews/ethereum-statistics.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[8] Bitcoin Core Developers. libsecp256k1. https://github.com/bitcoin-core/secp256k1.

[9] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[10] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[11] Consensys. Quorum. https://consensys.net/quorum/.

[12] J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, et al. Ambrosia: Providing performant virtual resiliency for distributed applications. *Proceedings of the VLDB Endowment*, 13(5):588–601, 2020.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[14] Hyperledger Foundation. Hyperledger Besu. https://www.hyperledger.org/use/besu.

[15] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.

[16] J. Lovejoy, C. Fields, M. Virza, T. Frederick, D. Urness, K. Karwaski, A. Brownworth, and N. Narula. A high performance payment processing system designed for central bank digital currencies. *Cryptology ePrint Archive*, 2022.

[17] J. Lovejoy, M. Virza, C. Fields, K. Karwaski, A. Brownworth, and N. Narula. Hamilton: A high-performance transaction processor for central bank digital currencies. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, pages 901–915, 2023.

[18] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 16–30, 2019.

[19] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, pages 508–526. Springer, 2019.

[20] Monad. EVM scalability: The case for radically higher throughput. https://monad-labs.notion.site/EVM-scalability-the-case-for-radically-higher-throughput-53b5188b9b034701ba0565a468691b6a.

[21] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[22] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[23] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, 2009.

[24] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 97–112, 2014.