Practical Smart Contract Sharding with Static Program Analysis

Ilya Sergey
A recipe for a robust distributed service
A recipe for a robust distributed service

1. Replicate your data
1. Replicate your data
2. Make each replica execute an identical operation log
A recipe for a robust distributed service

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
A recipe for a robust distributed service

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant
A recipe for a robust distributed service

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant
5. Make it so anyone can participate
A recipe for a robust distributed service

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant
5. Make it so anyone can participate
6. Add arbitrary computations with costs
What are the problems?

• **Correctness:**
  arbitrary replicated computations (aka *smart contracts*) can be *buggy* and allow for exploits

  **Examples:**
  - DAO reentrancy attack
  - Parity Multi-Sig Wallet hack

  100s of papers published since 2016 on verification and vulnerability detection

• **Scalability:**
  consensus protocols are *slow* due to decentralization; no advantage from inherent distributed parallelism

  **Examples:**
  - Ethereum only handles 11 TPS
  - A single popular decentralised application (e.g., CryptoKitties) can cause congestion of the system
Scaling bottlenecks: state and execution

- User transactions
- Network nodes
- Each node executes every transaction
- Adding more nodes does not increase throughput OR allow more state to be stored
- Each node stores the entire state
Scaling bottlenecks: state and execution

• Ethereum’s head **state size** was ~130 GB as of Nov 2021\(^\text{[1]}\)
  • For best performance, this needs to be kept in RAM
  • In practice, it is disk-based (NVME SSD) with caching in memory:
    • AFAIK, most of the time spent processing an Ethereum transaction is spent on disk I/O

• Increasing node hardware requirements decreases decentralisation
  • Solana validators already need >> 256 GB of RAM and 1 Gbps network

• In monolithic architectures, transaction **execution throughput** is limited by the capacity of the *least performant node* in the network

---

\(^\text{[1]}\) [https://twitter.com/peter_szilagyi/status/1460202014919569410](https://twitter.com/peter_szilagyi/status/1460202014919569410)
Scaling Blockchains: State of the Art

• **Layer 1** solutions
  Revise the *rules of the consensus* protocol to increase the throughput
  • Changing Proof-of-Work to *Proof-of-Stake*
  • Parallelism via *sharding*

• **Layer 2** solutions
  Adding *auxiliary protocols* to offload transaction processing
  • *Nested blockchains*: the main chain stores results of side-chain transactions
  • *Off-chain executions* via zkSNARKs and optimistic roll-ups
Sharding

Each shard executes a subset of transactions

shard 1 nodes

shard 2 nodes

shard 3 nodes
shard 1 nodes

shard 2 nodes

shard 3 nodes
shard 1 nodes

shard 2 nodes

shard 3 nodes
Look at the contract’s code to learn how to shard it!
Smart Contract Sharding

with Static Program Analysis

Ownership
Commutativity

problem we are working on
technique we are applying
Goals of Static Program Analysis

• Verification and Validation
  • Without running a program, soundly prove the absence of bugs...
    (e.g., Astrée Static Analyzer)
  • ...or soundly show their presence
    (e.g., Coverity, FindBugs, Infer/RacerD/Pulse(X))

• Uncovering opportunities for program optimization
  • Constant propagation, function inlining, static method dispatch
    (e.g., any optimizing compiler)
  • Automated parallelization
Smart Contract Sharding with Static Program Analysis

2

≈

Formal reasoning for \textit{scalability}, \textit{not} just verification
## MyToken

<table>
<thead>
<tr>
<th>Balances:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice: 10</td>
</tr>
<tr>
<td>Bob: 25</td>
</tr>
<tr>
<td>Charlie: 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuyTokens(amount, buyer)</td>
</tr>
<tr>
<td>Transfer(amount, from, to)</td>
</tr>
</tbody>
</table>
BuyTokens(amount, buyer)

ownership analysis

shard 1

shard 2

shard 3
Transfer(5, Alice, Bob)

Alice: 11
Bob: 25
Charlie: 12

Transfer(3, Charlie, Bob)

Alice: 6
Bob: +5
Charlie: 12

Bob: +3
Charlie: 9

Bob: +8

= 25

Bob: +3
Bob: +8

Bob: 33

commutativity analysis

shard 1

shard 2

shard 3
CoSplit
Static Program Analysis
for
Smart Contract Sharding
Two key features:

• Clearly separates computation from communication
  • message-passing rather than method calls for contract interaction

• Strict distinction between pure and effectful computations
  • Scilla has a small imperative fragment with conditionals but without loops
  • Only pure (non-effectful) recursion is allowed

static analysis can be quite precise
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
    to_bal <- balances[to];
    new_to_bal = match to_bal with
    | Some bal => builtin add bal amount
    | None => amount
    end;
  balances[to] := new_to_bal
Static analysis for transition effects

• **Ownership:** produce an **effect summary** for every transition
  • Effects include: reads, writes, accepting funds, sending messages, conditioning on values derived from mutable fields
  • The effect summary *over-approximates* the behaviour of the transition
  • Loosely inspired by Concurrent Separation Logic

• **Commutativity:** **linearity-aware flows-to analysis**
  • Effects of monotone operations, which use a field just once, *commute*
  • Inspired by GHC’s cardinality analysis (POPL’14)
  • Expressed as a type system for “contribution types”
    • compositional, but sometimes gives uninformative types
Constant $x, y$ constant contract field or transition parameter
Mutable $f$ mutable field or map-field access via parameter

<table>
<thead>
<tr>
<th>Contrib. src. $cs ::= x \mid f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality $card ::= \text{None} \mid \text{Linear} \mid \text{NonLinear}$</td>
</tr>
<tr>
<td>Operation $op ::= + \mid - \mid * \mid \ldots$</td>
</tr>
<tr>
<td>Abstr. expr. $e ::= \top \mid (cs, card, op)$</td>
</tr>
<tr>
<td>Effect $\varepsilon ::= \text{Read}(f) \mid \text{Write}(f, e) \mid \text{AcceptFunds} \mid \text{Condition}(e) \mid \text{Event}(e) \mid \text{SendMsg}(e) \mid \top$</td>
</tr>
</tbody>
</table>
transition Transfer(to: Address, amount: Uint)
    from_bal <- balances[_sender];
    match from_bal with |
        | Some bal =>
            match amount ≤ bal with |
                | True =>
                    new_from_bal = builtin sub bal amount;
                    balances[_sender] := new_from_bal;
            to_bal <- balances[to];
            new_to_bal = match to_bal with |
                | Some bal => builtin add bal amount
                | None => amount
            end;
        end
        balances[to] := new_to_bal;
    end;
Constraint $o_c ::= \text{Owns}(f) \mid \text{UserAddr}(x) \mid \text{NoAliases}(\langle x, y \rangle) \mid \text{SenderShard} \mid \text{ContractShard} \mid \bot$
Constraint: \( oc ::= \text{Owns}(f) \mid \text{UserAddr}(x) \mid \text{NoAliases}(\langle x, y \rangle) \mid \text{SenderShard} \mid \text{ContractShard} \mid \bot \)

Join: \( \forall_f ::= \text{OwnOverwrite} \mid \text{IntMerge} \)

**Weak reads**

- Owns(balances[_sender])
- NoAliases(<_sender, to>)

**OwnOverwrite** join for owned contributions

**IntMerge** join for un-owned contributions

Read(balances[to])

Write(balances[to],
\[\{(\text{balances[to]}, \text{Linear}, \text{add}),
(\text{amount}, \text{Linear}, \text{add})\}\])

Write(balances[_sender],
\[\{(\text{balances[_sender]}, \text{Linear}, \text{sub}),
(\text{amount}, \text{Linear}, \text{sub})\}\])

Condition(balances[_sender])

Condition(balances[_sender], amount)
Integration
A sharded blockchain design
Integrating CoSplit with Zilliqa

1. Run the static analysis when the contract is first deployed
2. Store the resulting sharding signature
   = set of transition constraints + join instructions for each field in the contract
3. When processing a transaction, solve the constraints to determine which shard(s) the transaction can be processed by
   • if the constraints have no solution, must process sequentially/cross-shard
4. After parallel processing, merge (join) state contributions from shards before sequential transactions are processed
Evaluation
Throughput: Transactions per Second

- Baseline 3 shards
- CoSPLIT 3 shards
- CoSPLIT 4 shards
- CoSPLIT 5 shards

- FT fund
- FT transfer
- CF donate
- NFT mint
- NFT transfer
- ProofIPFS
- UD bestow
- UD config
Limitations and Discussion

• Currently no support for sharding multi-contract transactions
  • We would need to somehow combine the signatures from multiple contracts

• Some contracts require simple rewriting to be shardable
  • An opportunity for program repair
transition transfer(to: ByStr20, tokenId: Uint256)

getchain <- getChain();
match getchain with
| None => throw
| Some chain =>

isOwner = builtin eq _sender tokenOwner;
(* ... *)

getchain <-
operatorApprovals[tokenOwner][_sender];
(* ... *)
tokenOwners[tokenId] := to;
Limitations and Discussion

• Currently no support for sharding multi-contract transactions
  • We would need to somehow combine the signatures from multiple contracts

• Some contracts require simple rewriting to be shardable
  • An opportunity for program repair

• Some programming languages (e.g., Move or Solana’s Rust dialect) might be even better targets for sharding analysis
  • one can also ask the programmer for ownership/commutativity annotations
Conclusion: What this talk was about

a parallelising compiler for blockchains
Sharding is a solution to the blockchain scalability problem

Some smart contract logic can be sharded (i.e., executed in parallel)

We developed static analysis to soundly determine sharding conditions for smart contracts

The technique has been integrated into real-world blockchain and gave observable increase in the throughput

Thanks!