CoSplit

Practical Smart Contract Sharding with Static Program Analysis

George Pîrlea
Amrit Kumar
Ilya Sergey

NUS National University of Singapore
zilliqa
Yale NUS College

problem

Smart Contract Sharding

with Static Program Analysis

technique

Scaling blockchains

that support smart contracts

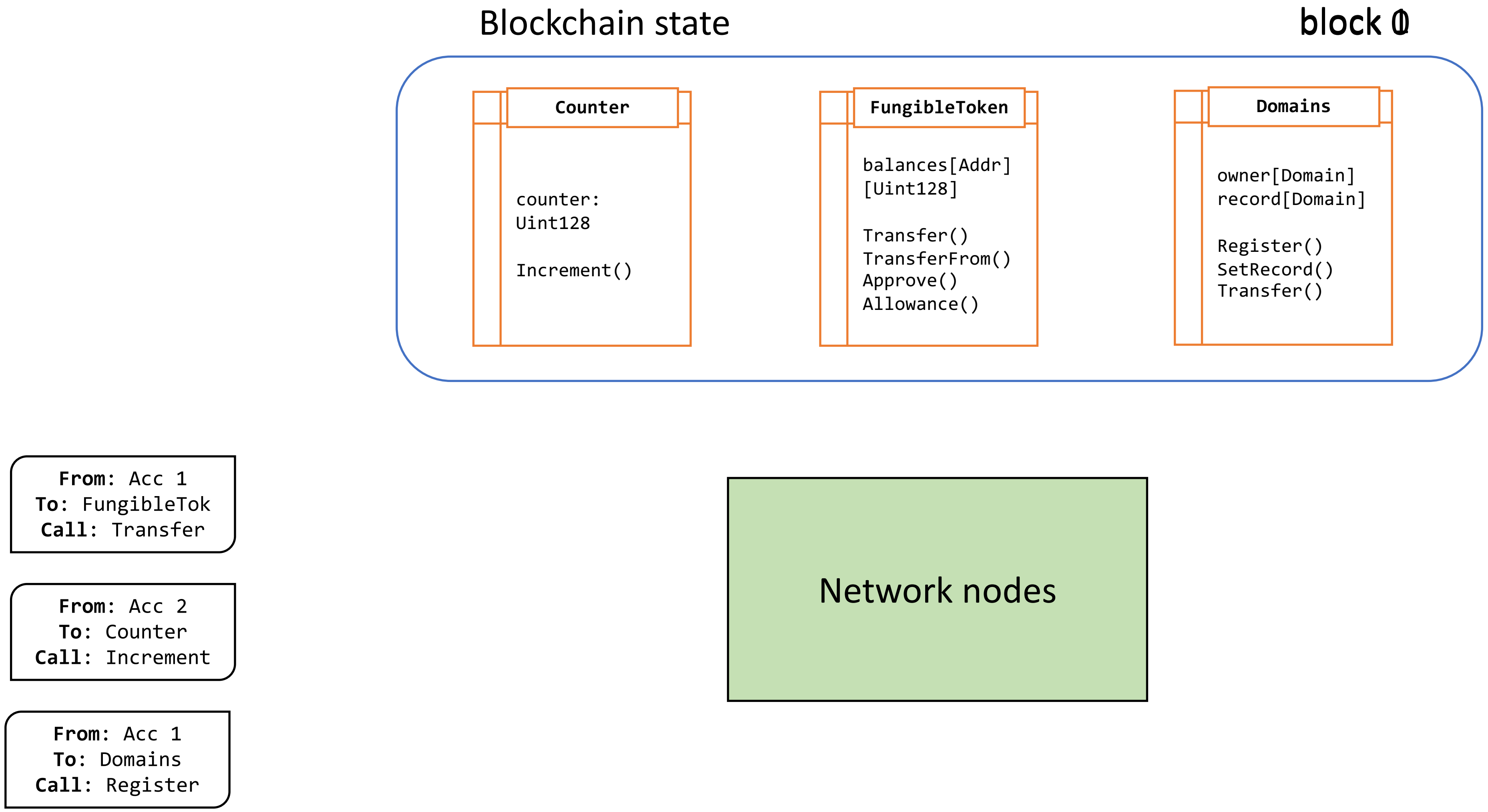
with Static Program Analysis

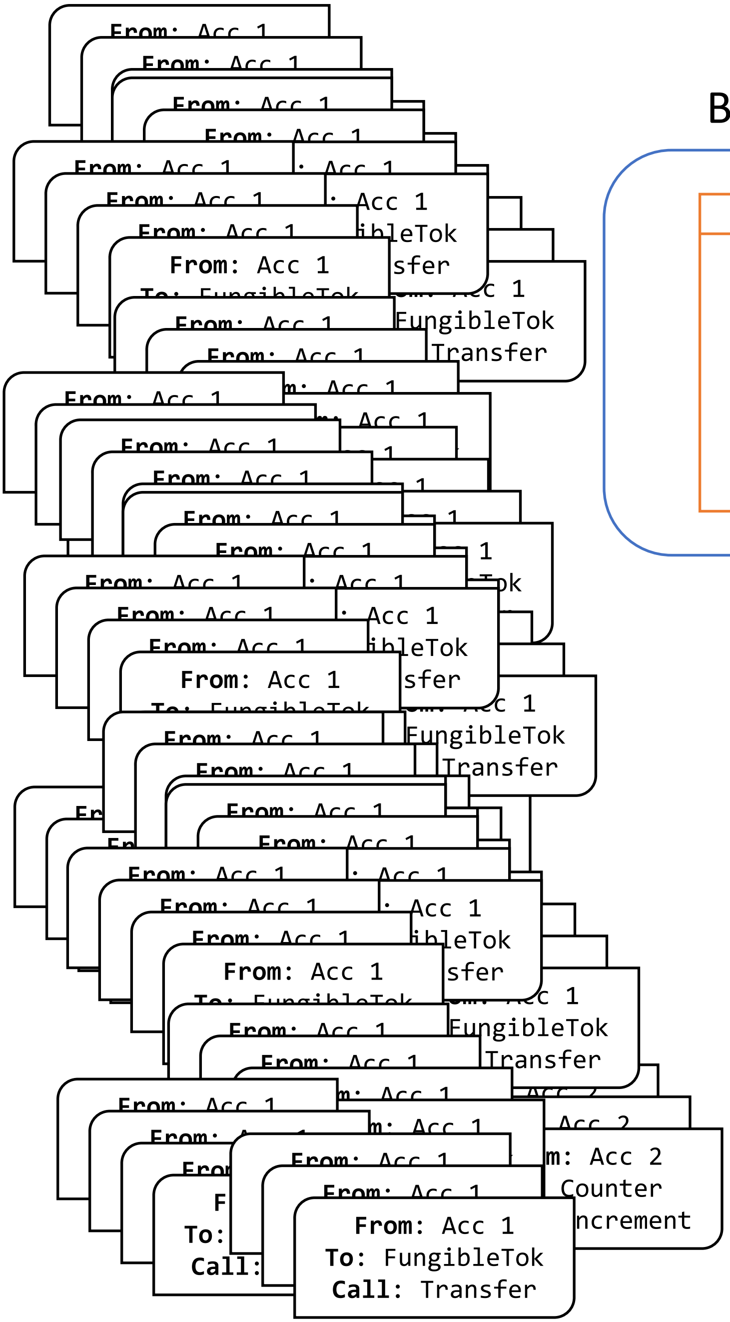
Scaling blockchains

that support smart contracts

with Static Program Analysis

Blockchains don’t scale


Blockchain state
block 0
Counter
counter: Uint128
Increment()
FungibleToken
balances[Addr][Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Domains
owner[Domain]
record[Domain]
Register()
SetRecord()
Transfer()
From: Acc 1
To: FungibleTok
Call: Transfer
From: Acc 2
To: Counter
Call: Increment
From: Acc 1
To: Domains
Call: Register
Network nodes



Blockchain state

block 2

Counter

counter: Uint128

Increment()

FungibleToken

balances[Addr][Uint128]

Transfer()
TransferFrom()
Approve()
Allowance()

Domains

owner[Domain]
record[Domain]

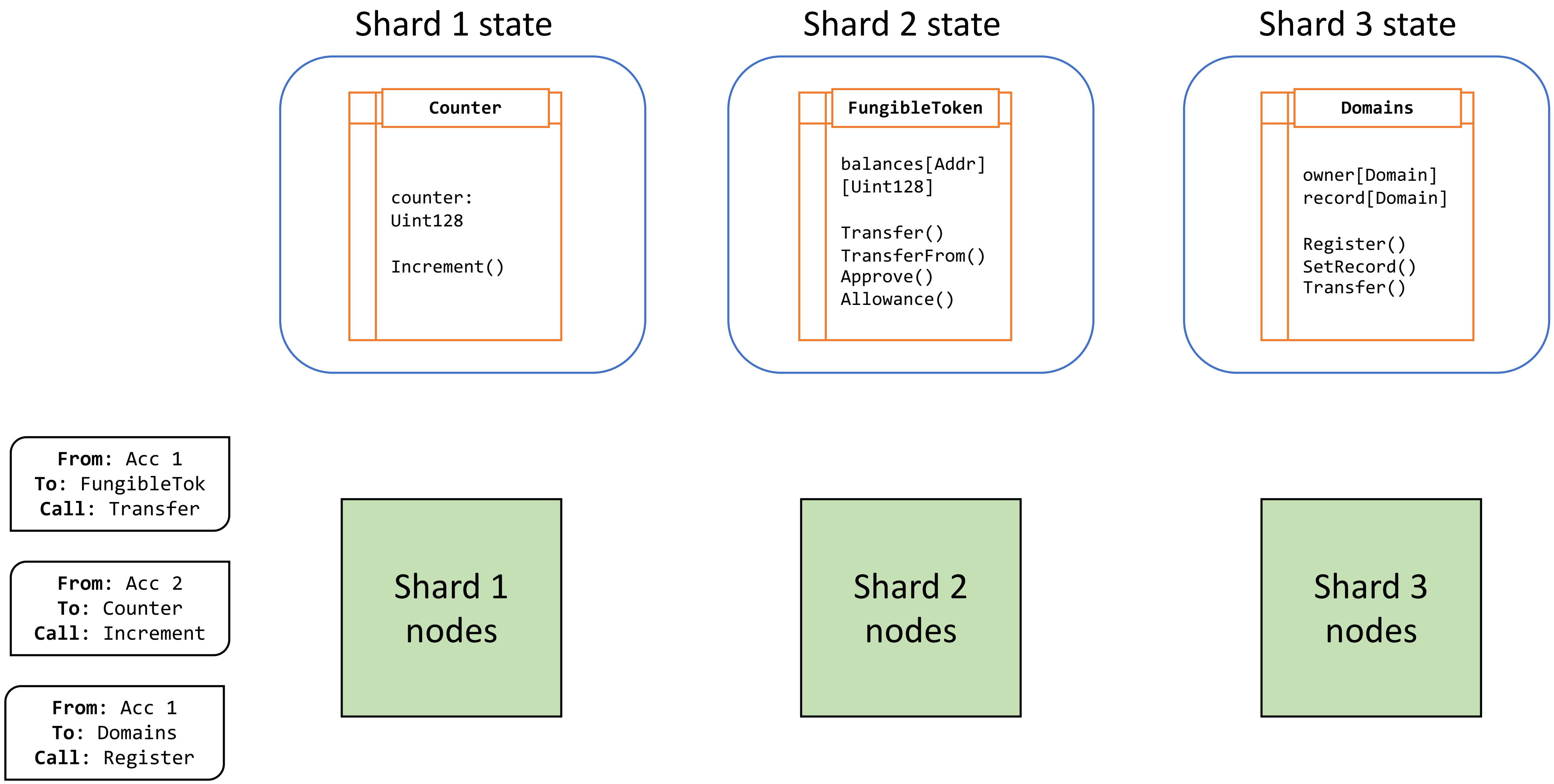
Register()
SetRecord()
Transfer()

Network nodes

nodes have limited bandwidth

and limited processing power

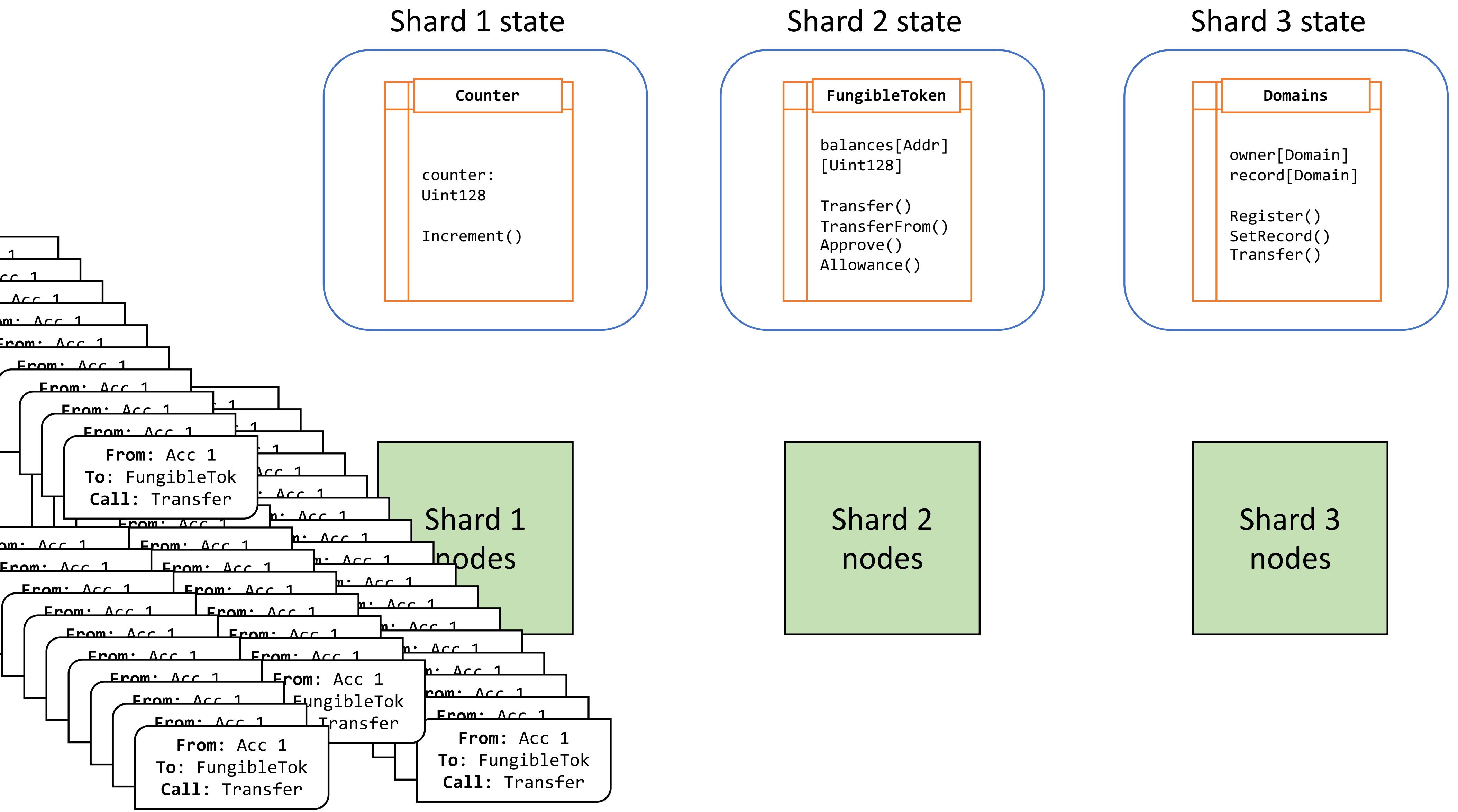
Network nodes

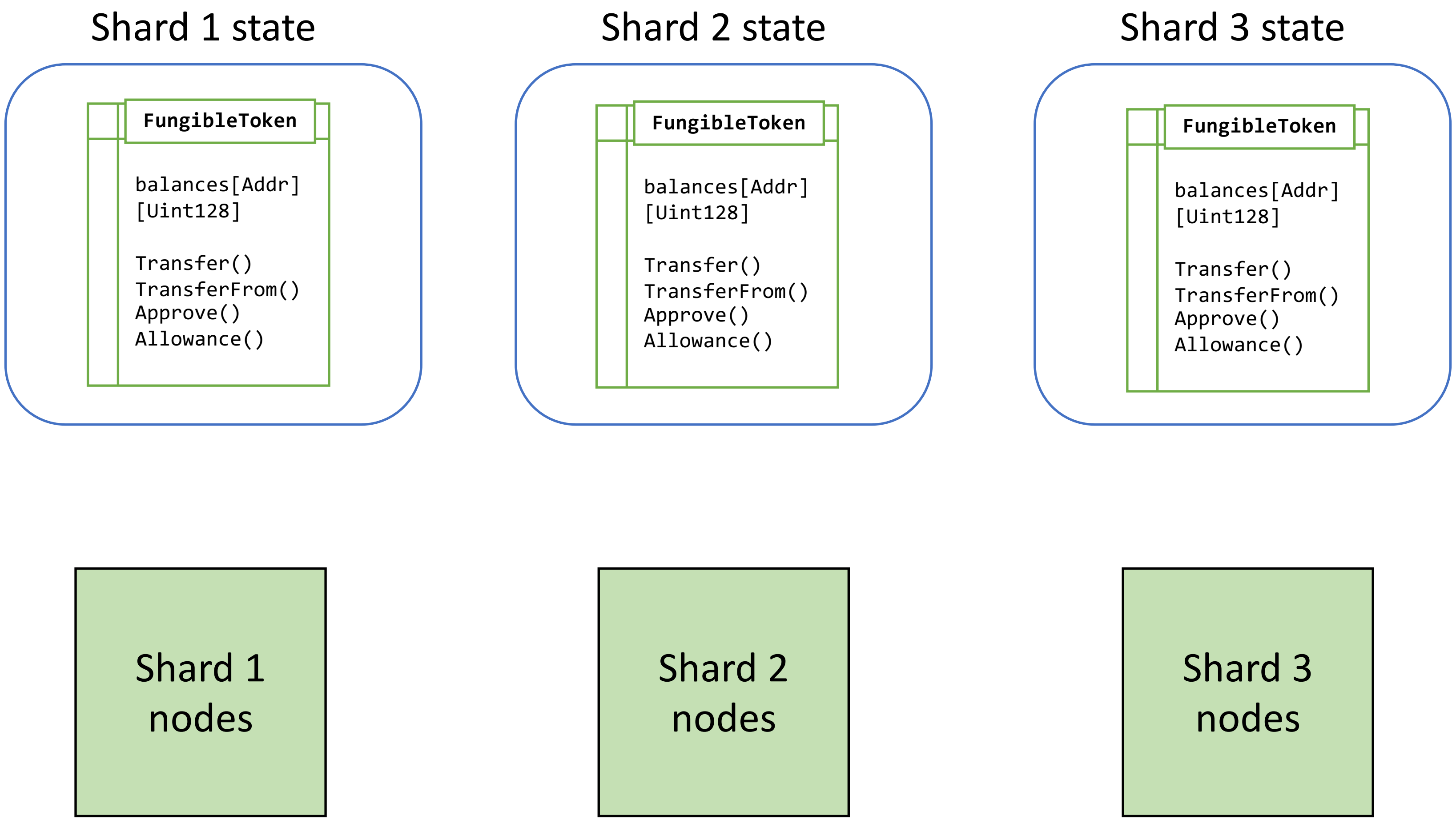


Shard 1 state
Counter
counter: Uint128
Increment()
Shard 2 state
FungibleToken
balances[Addr][Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Shard 3 state
Domains
owner[Domain]
record[Domain]
Register()
SetRecord()
Transfer()
From: Acc 1
To: FungibleTok
Call: Transfer
From: Acc 2
To: Counter
Call: Increment
From: Acc 1
To: Domains
Call: Register
Shard 1 nodes
Shard 2 nodes
Shard 3 nodes

Sharding has limitations

Especially for smart contracts!


Shard 1 state
Counter
counter: Uint128
Increment()
Shard 2 state
FungibleToken
balances[Addr][Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Shard 3 state
Domains
owner[Domain]
record[Domain]
Register()
SetRecord()
Transfer()
From: Acc 1
To: FungibleTok
Call: Transfer
Shard 1 nodes
Shard 2 nodes
Shard 3 nodes


Shard 1 state
FungibleToken
balances[Addr]
[Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Shard 2 state
FungibleToken
balances[Addr]
[Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Shard 3 state
FungibleToken
balances[Addr]
[Uint128]
Transfer()
TransferFrom()
Approve()
Allowance()
Shard 1 nodes
Shard 2 nodes
Shard 3 nodes

Sharded contracts: intuition

- A *blockchain* is a state transition system, consisting of:
  - State
  - Rules that say which state updates are legal

- We have strategies for sharding blockchains (for certain kinds of rules*)

* - some rules shard better than others and some don't shard at all

- A **smart contract** is a state transition system, consisting of:
  - State
  - Code that says which state updates are legal

- …

Does the code of the contract define a shardable state machine?

Does the code have property X?

We can shard contracts the same way we shard blockchains.

Static analysis uncovers the opportunities.

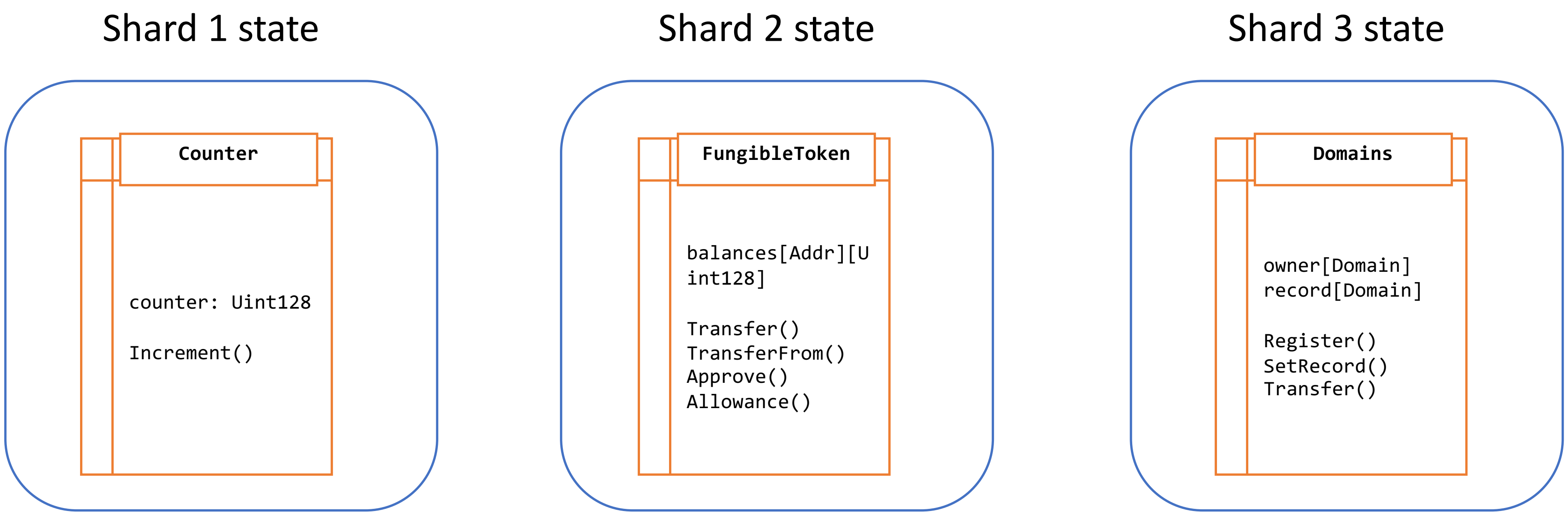
CoSplit

Practical Smart Contract Sharding with Static Program Analysis

Our contributions

- Identify enabling mechanisms for sharding Ethereum-style contracts
  - and show their adequacy for some realistic contracts
- CoSplit, a static analysis tool that infers sharding strategies for smart contracts written in Scilla, an ML-style smart contract language
- End-to-end integration of CoSplit with a production-grade sharded blockchain (Zilliqa)
- Evaluation of the inferred sharding strategies
  - almost linear throughput increase as number of shards goes up

Mechanism (1): disjoint state ownership

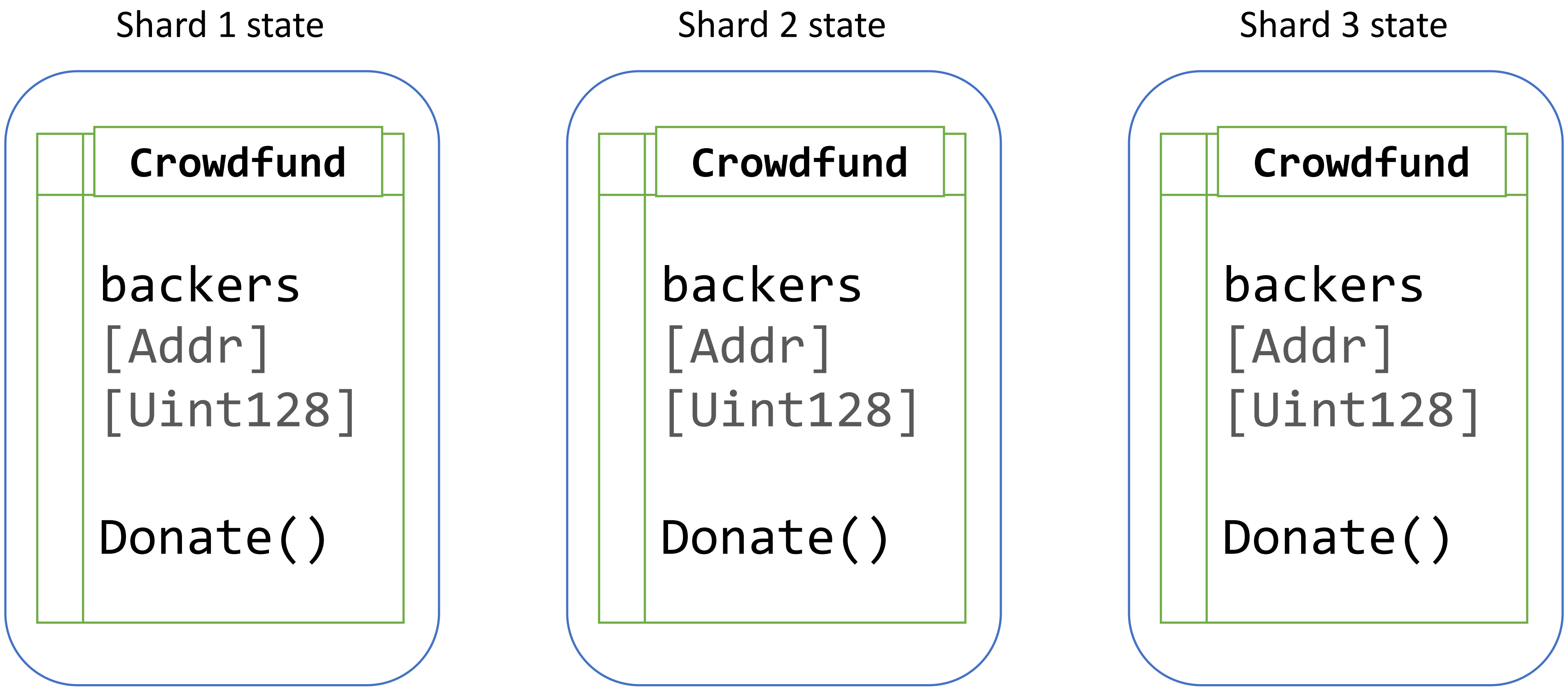


field backers : Map ByStr20 Uint128

```
transition Donate ()
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
      c <- exists backers[_sender];
      match c with
      | False =>
        accept;
        backers[_sender] := _amount;
```

cf. **Safer Smart Contract Programming with Scilla**, Sergey *et al.*, OOPSLA'19

Mechanism (1): disjoint state ownership



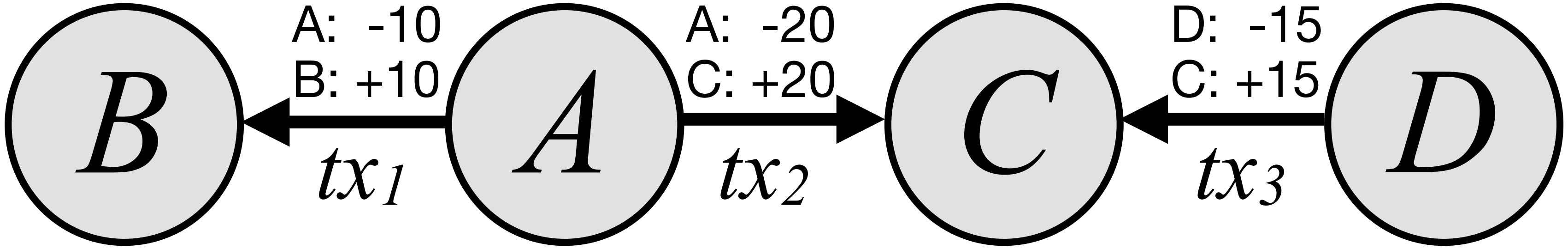
Mechanism (2): commutative effects

```
field counter : Uint128 = Uint128 0

transition Increment ()
  c <- counter;
  inc = Uint128 1;
  new_c = builtin add c inc;
  counter := new_c
end
```

Cumulative result can be obtained by ***joining*** the contributions from each shard.

Mechanism (2): commutative effects



Commutative operations do not imply commutative effects!

```
field counter : Uint128 = Uint128 0
```

```
transition Increment ()
  c <- counter;
  inc = Uint128 1;
  new_c = builtin add c inc;
  counter := new_c
end
```

```
transition Double ()
  c <- counter;
  new_c = builtin add c c;
  counter := new_c
end
```

$$2 * (2x + 1) \neq 2 * 2x + 1$$

Static analysis for transition effects

- Produce an **effect summary** for every transition in the contract
  - 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

Static analysis for transition effects

- Produce an **effect summary** for every transition in the contract
  - 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

- Determine which effects are commutative using a **linearity-tracking flows-to analysis**
  - The analysis is expressed as a type system for "contribution types" and is compositional (but gives uninformative types in some cases)

Constant	x, y	constant contract field or transition parameter
Mutable	f	mutable field or map-field access via parameter
Contrib. src.	$cs ::=$	$x \mid \mathsf{f}$
Cardinality	$card ::=$	$\mathsf{None} \mid \mathsf{Linear} \mid \mathsf{NonLinear}$
Operation	$op ::=$	$+ \mid - \mid * \mid \ldots$
Abstr. expr.	$e ::=$	$\top \mid \overline{(cs, card, \overline{op})}$
Effect	$\varepsilon ::=$	$\mathsf{Read}(\mathsf{f}) \mid \mathsf{Write}(\mathsf{f}, e) \mid \mathsf{AcceptFunds} \mid$
		$\mathsf{Condition}(e) \mid \mathsf{Event}(e) \mid \mathsf{SendMsg}(e) \mid \top$

```
transition Transfer(to: ByStr20, 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
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

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

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

Read(balances[to])

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

Sharding Constraints

A language for restricting a set of shards that can execute a certain transition of a contract.

$$\text{Constraint} \qquad oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y\rangle) \mid$$
$$\mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$$

Constraint $oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y\rangle) \mid$
$\mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$

Join $\uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Read(balances[_sender])

Condition(balances[_sender])

Weak reads

Condition(balances[_sender], amount)

Owns(balances[_sender])

OwnOverwrite join for owned contributions

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

Read(balances[to])

IntMerge join for un-owned contributions

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

```
transition Transfer(to: ByStr20, 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
```

Owns(balances[_sender])

OwnOverwrite join for owned contributions

IntMerge join for un-owned contributions

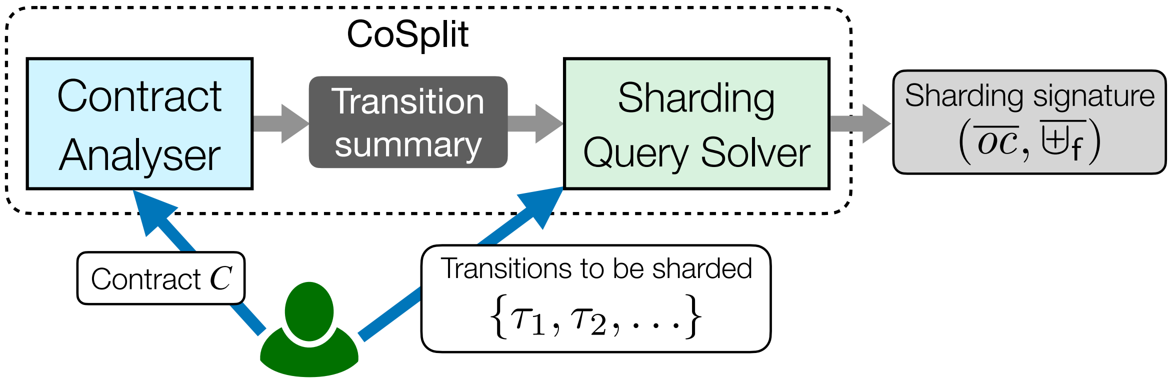
```
type expr_type =
  | ETop
  | EVal of known_contrib
  | ECompositeVal of expr_type * expr_type

  | EOp of contrib_op * expr_type

  | EComposeSequence of expr_type list
  | EComposeParallel of expr_type * expr_type list

  | EFun of efun_desc
  | EApp of efun_desc * expr_type list
```

Analysis Pipeline



1. Derive summaries for all contract transitions (R/W, operations, lin-ty)

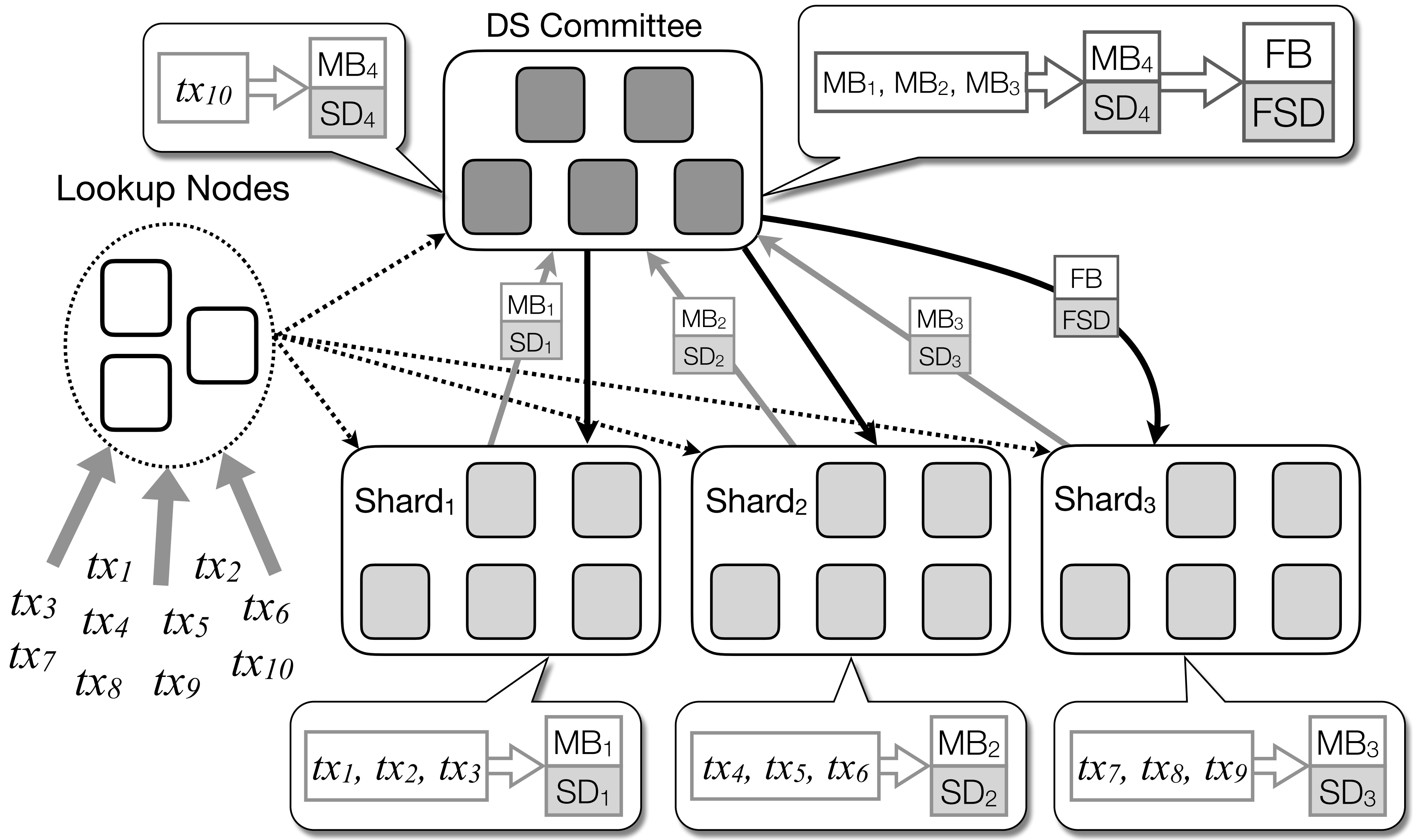
2. Take from a user a set of transitions she wants to shard + weak reads

3. Produce an optimal (the most permissive) set of sharding constraints. These constraints determine conditions a shard need to satisfy in order to run the transaction with this transition.

4. Sharding more transitions of a contract => stronger constraints

Integration

A sharded blockchain design



Integrating CoSplit with Zilliqa

1. Run the static analysis upon contract deployment

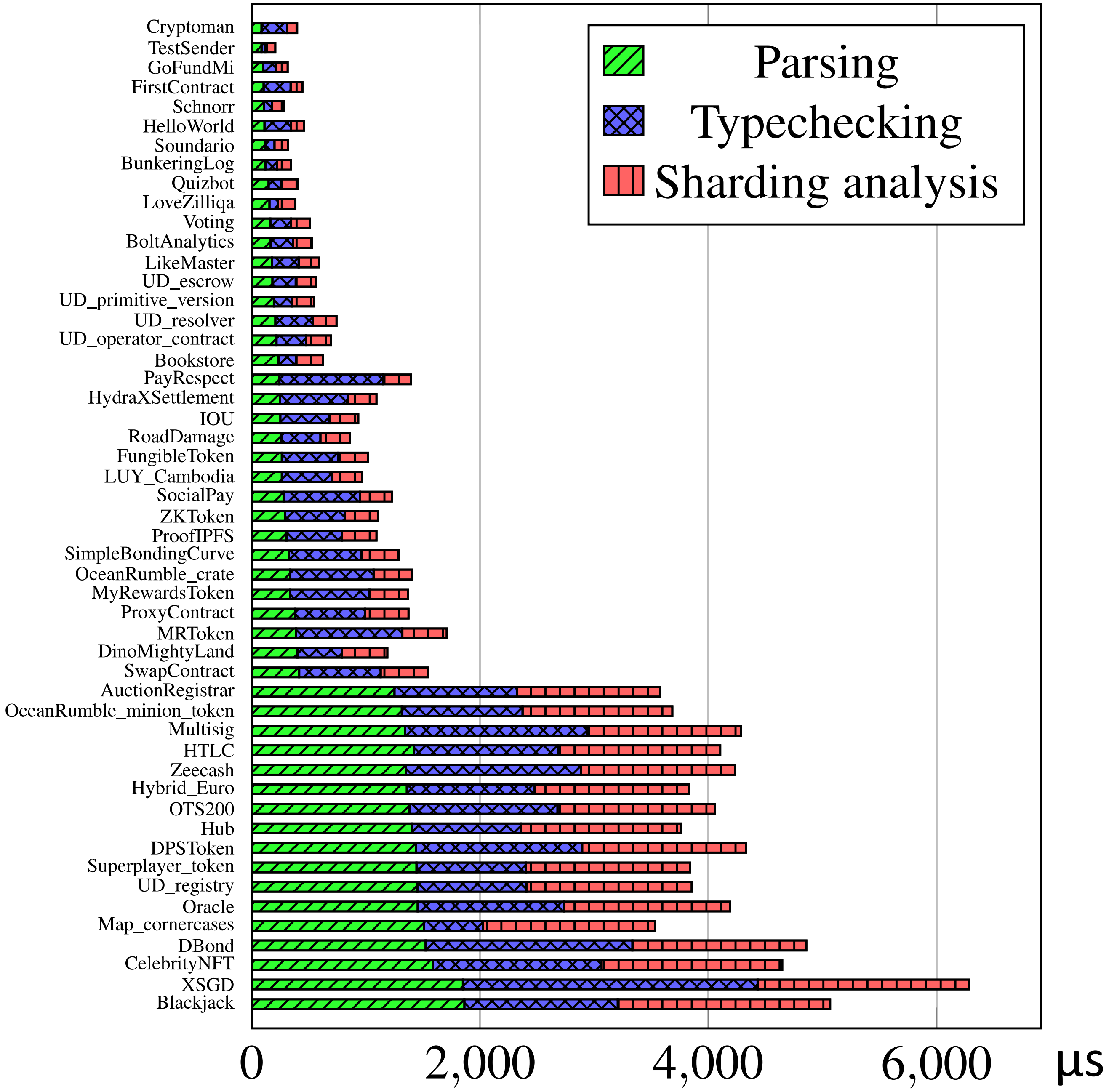
2. Store the resulting sharding signature

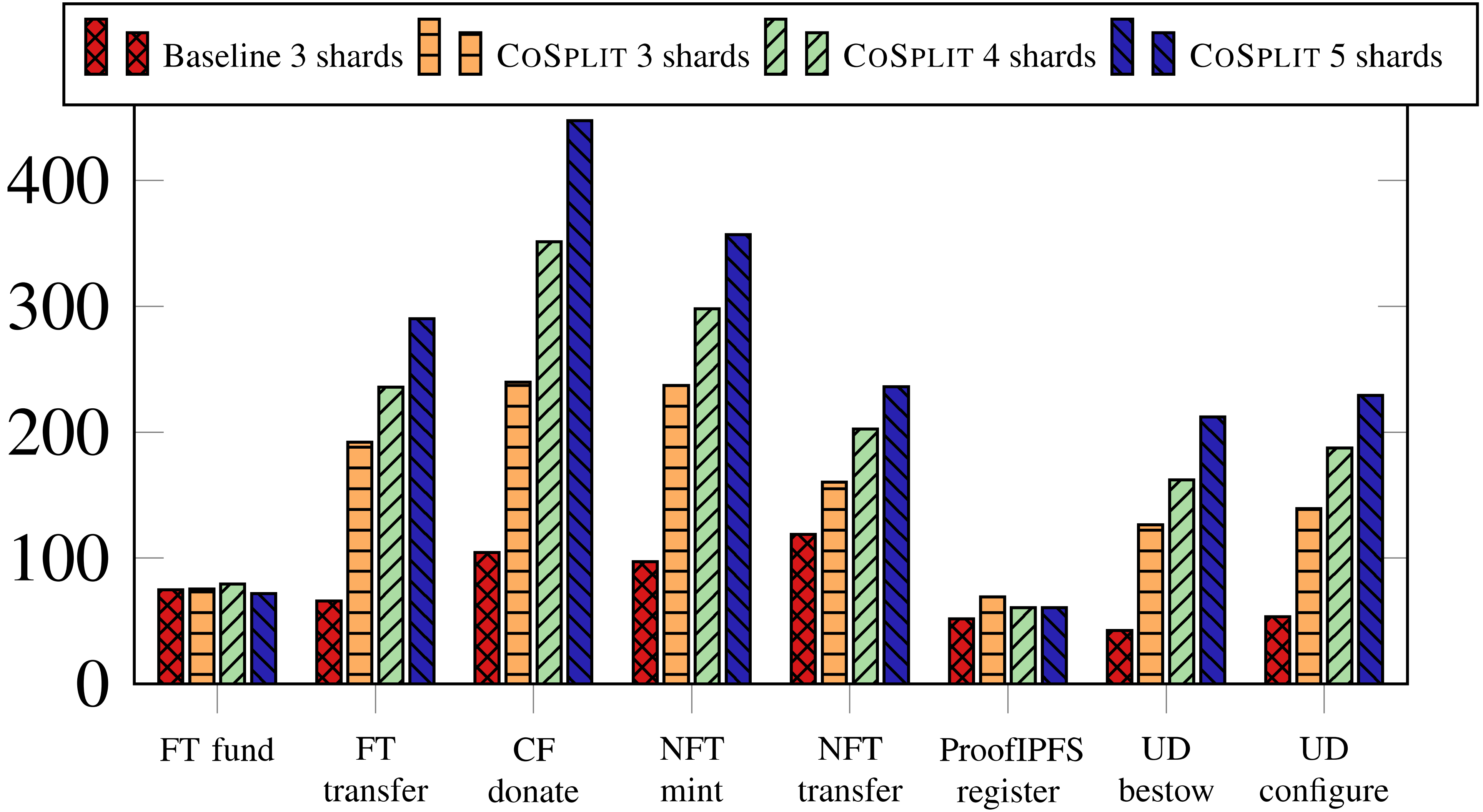
   = set of transition constraints + join instructions for each field in the contract

3. Dispatch transactions to shards using the constraints to determine which shard the transaction needs to go to

4. Merge (join) contributions from shards before sequential/cross-shard transactions are processed

Evaluation


Parsing
Typechecking
Sharding analysis
Cryptoman
TestSender
GoFundMi
FirstContract
Schnorr
HelloWorld
Soundario
BunkeringLog
Quizbot
LoveZilliqa
Voting
BoltAnalytics
LikeMaster
UD_escrow
UD_primitive_version
UD_resolver
UD_operator_contract
Bookstore
PayRespect
HydraXSettlement
IOU
RoadDamage
FungibleToken
LUY_Cambodia
SocialPay
ZKToken
ProofIPFS
SimpleBondingCurve
OceanRumble_crate
MyRewardsToken
ProxyContract
MRToken
DinoMightyLand
SwapContract
AuctionRegistrar
OceanRumble_minion_token
Multisig
HTLC
Zeecash
Hybrid_Euro
OTS200
Hub
DPSToken
Superplayer_token
UD_registry
Oracle
Map_cornercases
DBond
CelebrityNFT
XSGD
Blackjack
0
2,000
4,000
6,000
µs



Average TPS for different contract transitions as a function of number of shards, over 10 epochs.

Limitations

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

- At the moment, the contract is *always* sharded
  - non-parallelizable operations become cross-shard (synchronized)
  - inefficient if the contract has either low transaction volume (overhead dominates) or many transactions to non-parallelizable transitions (forced into expensive synchronization)
  - would want dynamic sharding – only shard the contract when transactions are of a profile that is known to shard well; otherwise keep it to a single shard

Future work

- Implement dynamic sharding to alleviate fixed-strategy limitation
- Produce sharding signatures that allow sharding of multi-contract calls with commutative effects
  - proxy contracts are a common example of this
- Automatic contract repair to make contracts shardable, e.g.:
  - split records into a separate map for each component
  - translate to compare-and-swap transitions

```
transition transfer(to: ByStr20, tokenId: Uint256)
  getTokenOwner <- tokenOwners[tokenId];
  match getTokenOwner with
  | None => throw
  | Some tokenOwner =>
    isOwner = builtin eq _sender tokenOwner;
    (* … *)
    getOperatorStatus <-
      operatorApprovals[tokenOwner][_sender];
    (* … *)
    tokenOwners[tokenId] := to;
```

```
transition transfer(tokenOwner: ByStr20,
                    to: ByStr20, tokenId: Uint256)
  getTokenOwner <- tokenOwners[tokenId];
  match getTokenOwner with
  | None => throw
  | Some actual =>
    isCorrectOwner = builtin eq tokenOwner actual;
    match isCorrectOwner with
    | False => throw
    | True =>
      isOwner = builtin eq _sender tokenOwner;
      (* … *)
      getOperatorStatus <-
        operatorApprovals[tokenOwner][_sender];
      (* … *)
      tokenOwners[tokenId] := to;
```

To Take Away

- Sharding is a solution to the blockchain scalability problem

- Some smart contract logic can be sharded (“pessimistically parallelized”) in the same way as simple blockchain transactions.

- We use static analysis to soundly determine conditions under which (parts of) smart contracts can be executed in parallel.

- The technique has been integrated into real-world blockchain and shown to give observable increase in the throughput.

Thanks!