Towards Efficient Execution of Smart Contracts

Nuno P. Lopes, IST – U Lisbon

Joint work with Aptos Labs: George Mitenkov, Rati Gelashvili, Alexander Spiegelman, Zekun Li, Igor Kabiljo, Satyanarayana Vusirikala, Zhuolun Xiang, Aleksandar Zlateski

Efficient Smart Contract Execution

- Metering of Smart Contracts
- Parallel Execution of Smart Contracts

Smart contracts

Programs stored on the blockchain

1. Written in a high-level language

2. Compiled to bytecode

3. Executed in a virtual machine

```
module aptos_framework::coin {
  use std::error;

  struct Wallet<phantom T> has key {
    balance: u64,
  }

  public fun balance<T>(a: address): u64 acquires Wallet {
    assert!(exists<Wallet<T>>(a), error::not_found(404));
    borrow_global<Wallet<T>>(a).balance
  }

  public entry fun transfer<T>(
    from: &signer,
    to: address,
    amount: u64,
  ) acquires Wallet {
    deposit(to, withdraw<T>(from, amount));
  }
}
```

Gas metering

Gas: a fundamental **unit of computation** which represents the **cost of resources** used when a smart contract is executed.

1. Executing instructions costs gas

2. User defines a gas limit

3. Gas costs summed up

4. Execution halted if not enough gas

```
public balance<Ty>(Arg: address): u64
B0:                                                   //  GAS
     0: CopyLoc[0](Arg: address)                      //   1
     1: ExistsGeneric[0](Wallet<Ty>)                  //   5
     2: BrFalse(4)                                    //   2
B1:
     3: Branch(7)                                     //   2
B2:
     4: LdConst[6](U64: 404)                          //  10
     5: Call error::not_found(u64): u64               //  20
     6: Abort                                         //   3
B3:
     7: MoveLoc[0](Arg: address)                      //   1
     8: ImmBorrowGlobalGeneric[0](Wallet<Ty>)         //   5
     9: ImmBorrowFieldGeneric[0](Wallet.balance: u64) //   1
    10: ReadRef                                       //   3
    11: Ret                                           //   1
```

Cost Model

- Over-approximation of the costs incurred when executing a contract
  - Load the contract, VM start-up time, …
  - Execution: CPU, RAM, network
  - Storage
  - Protocol costs
- Deterministic and equal on all platforms
- Enforces limits on execution: transaction/block size, latency
- DoS protection

Different Cost Models

	Ethereum	Solana	NEAR	Aptos
Token price	\$1,631	\$17.5	\$1.1	\$5.5
Gas unit (signature) cost	$\$23.0 \cdot 10^{-6}$	$\$87.5 \cdot 10^{-6}$	$\$0.11 \cdot 10^{-15}$	$\$7.7 \cdot 10^{-6}$
Cost of addition	$\$69 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	$\$24.6 \cdot 10^{-9}$

Multiplication or division	Ethereum	Solana	NEAR	Aptos
# additions	1.67	1	1	1
USD	$\$115 \cdot 10^{-6}$	\$0	$\$0.09 \cdot 10^{-12}$	$\$24.6 \cdot 10^{-9}$

Division is 30x slower than addition; it’s easy/cheap to harm some networks!

Metering

- An implementation of the cost model
- Cost models must be efficiently implementable

pseudo-instruction

```
B2:
  3: ChargeGas(U64: 10)
  4: LdConst[6](U64: 404)
  5: ChargeGas(U64: 20)
  6: Call error::not_found(u64): u64
  7: ChargeGas(U64: 1)
  8: Abort
```

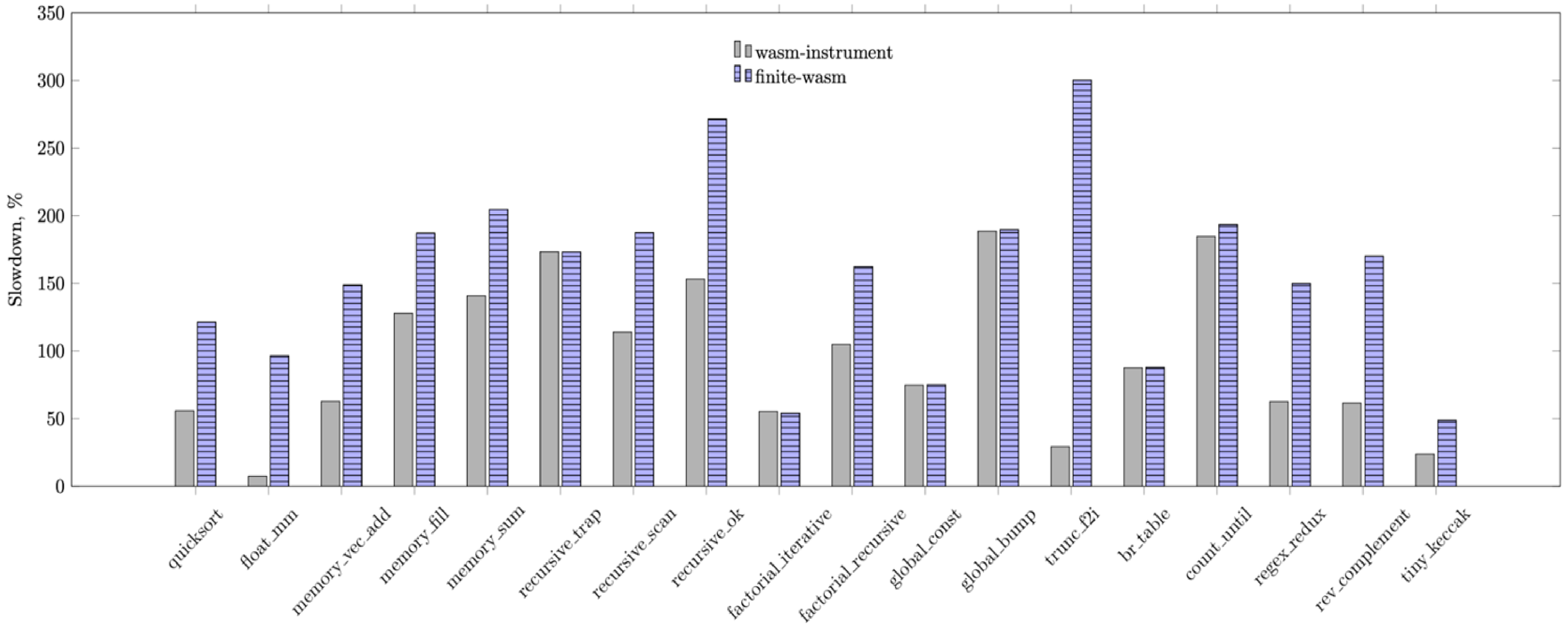
bytecode, e.g., WebAssembly

```
(block
  (call $meter(i64.const 10))
  (local.set $t6 404)
  (call $meter(i64.const 20))
  (call $not_found((local.get $t6))
  (call $meter(i64.const 1))
  (unreachable)
```

native code, e.g., x86

```
  1:   ; meter instruction gas cost
  2:   sub     rax, 10
  3:   jb     .out_of_gas
  4:   ; continue execution
       …
100: .out_of_gas
101:   ; handle out of gas
```

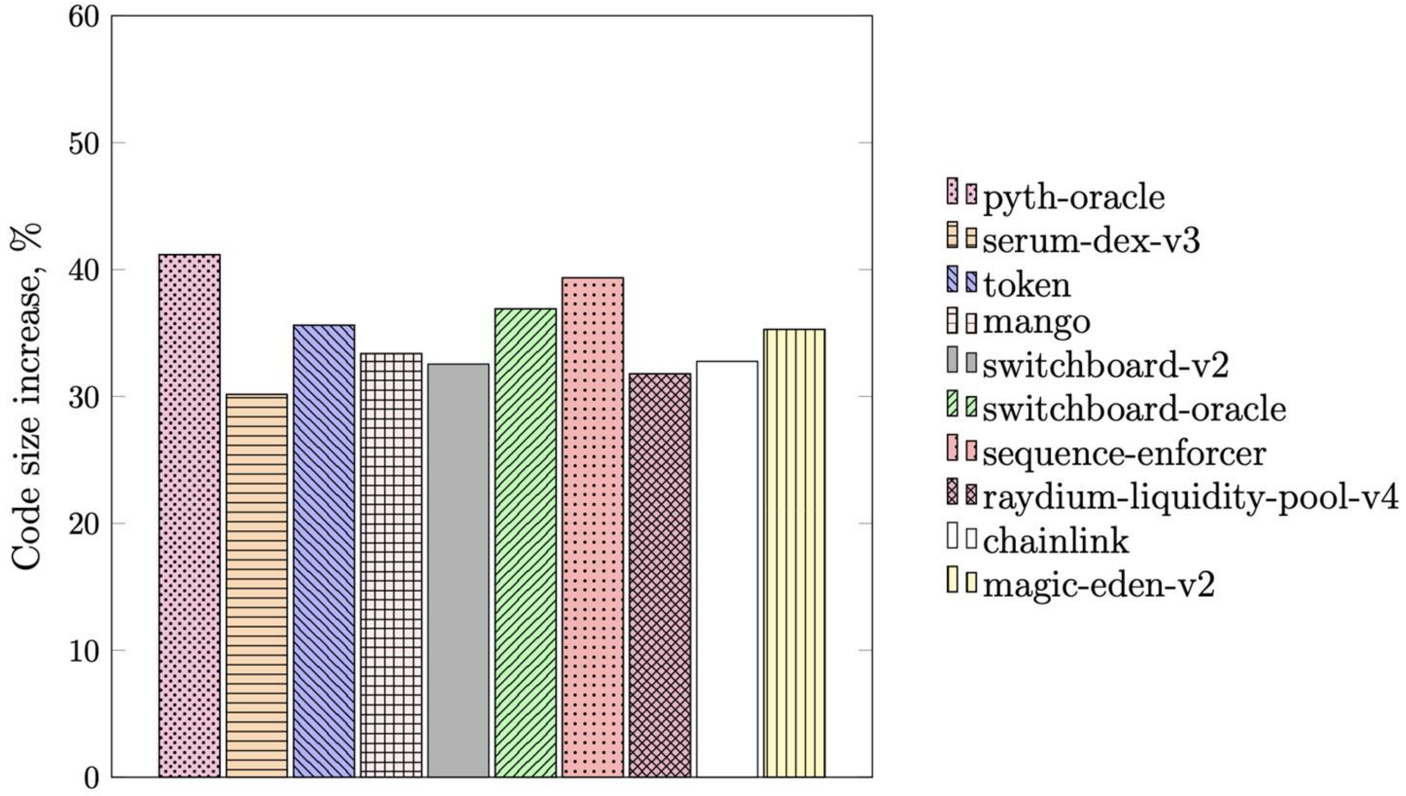
Metering in fast interpreters (WebAssembly)



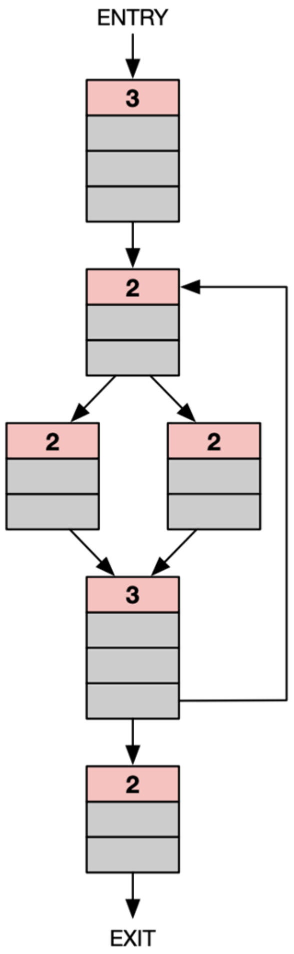
Metering in JIT compilers (Solana)

Contract is compiled from eBPF into x86, metering on every instruction is not feasible!

Solana uses a smart algorithm (~per each basic block)



Example Metering



State of the art

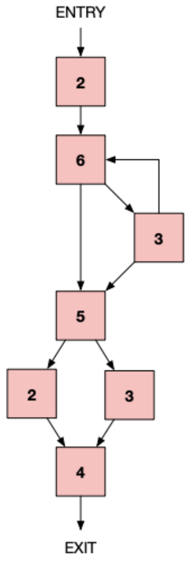
Minimal metering instrumentation problem

Find a minimal set of metering instrumentation points in the program so that:

1) Execution is metered online,
2) The sum of metered amounts is equal to the execution cost,
3) At most k gas executed for free (k-safe).

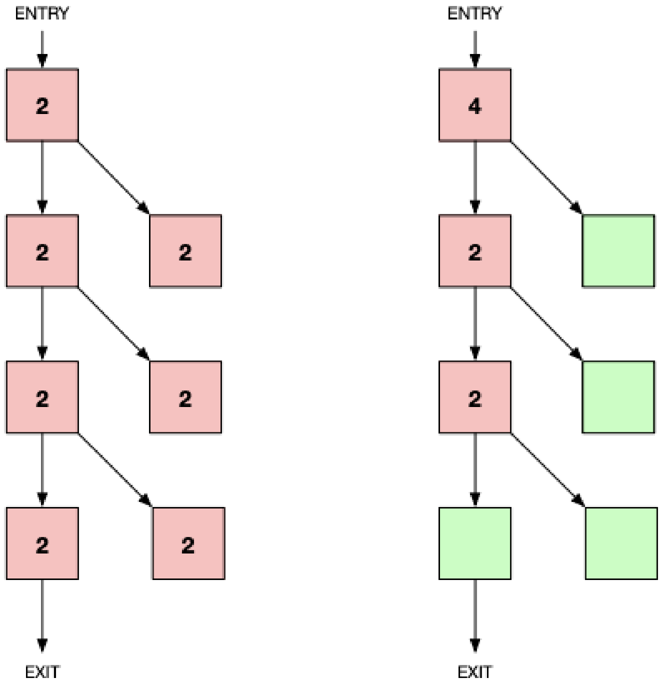
If k=0, nothing goes uncharged! (we say the metering is safe)

Algorithm

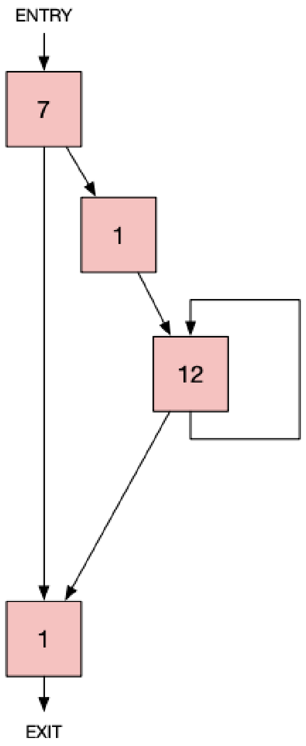


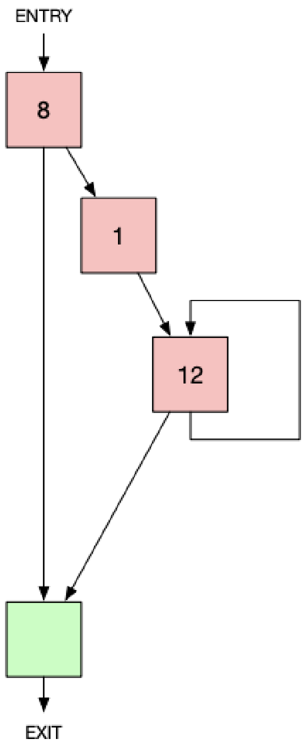
Original CFG

K-Safe Instrumentation examples

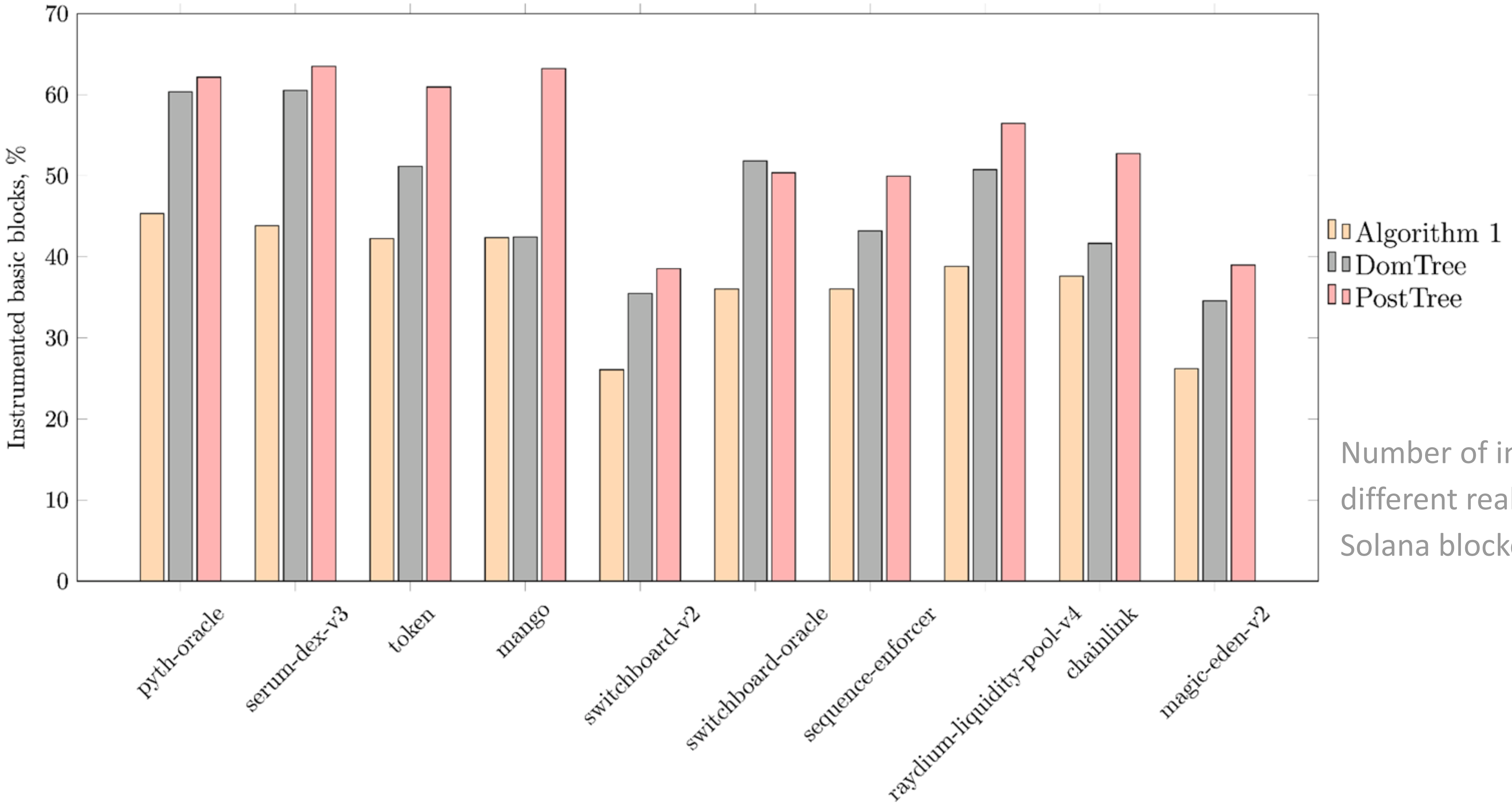


K-Safe Instrumentation examples





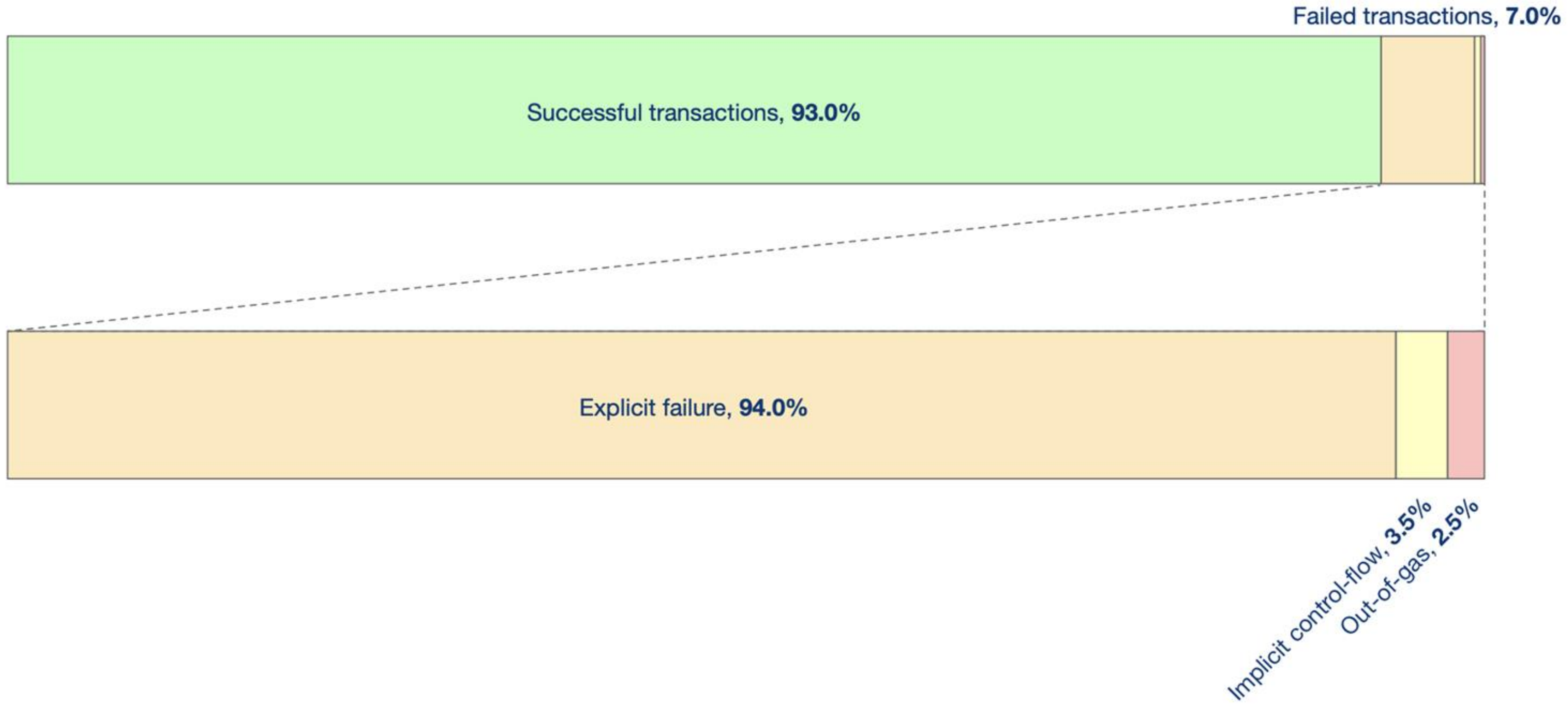
Real contracts



Number of instrumented blocks for different real contracts deployed on Solana blockchain.

Problems with per-block instruction gas cost metering

Do errors matter?



Recovery Mechanism

- Enable movement of charges across implicit control flow
    - E.g., division by zero traps in Move
- Traps and out-of-gas are rare (< 1%)
- Switch to a slower recovery mechanism

Metering Wrap-up

- Existing cost models don't give the right incentives
- Metering can introduce high overhead
- Being optimistic is a good tradeoff

Open Questions

- Can we give economic incentives to contract developers & compilers to make metering more efficient?
  - E.g., branches with same cost, no implicit control flow

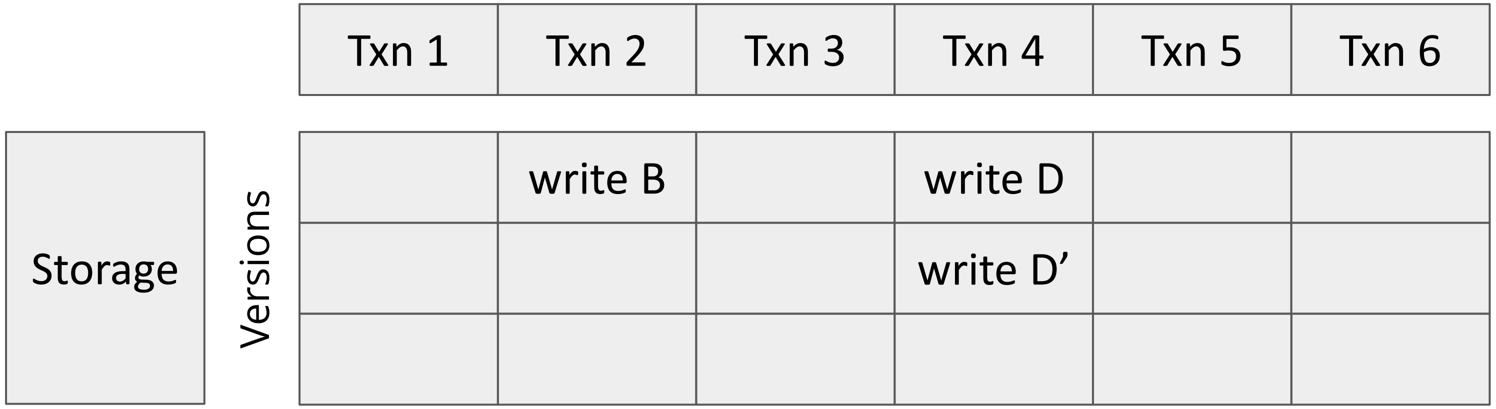
- Can metering be computed at compile time?
  - Validated at run time (a la proof carrying code)

Efficient Smart Contract Execution

- Metering of Smart Contracts
- **Parallel Execution of Smart Contracts**

Aptos runs on Block-STM

- Executes transactions in parallel
- Optimistic concurrency control
- Writes recorded in a multi-versioned data-structure
- Rolling commits: prefix of transactions committed on the fly



Inherently sequential workloads

Sequentiality is mostly due to **counters**:

- total supply tracking
- user balance updates
- sequence numbers
- NFT collection size tracking and indexing

Simple counter in Move

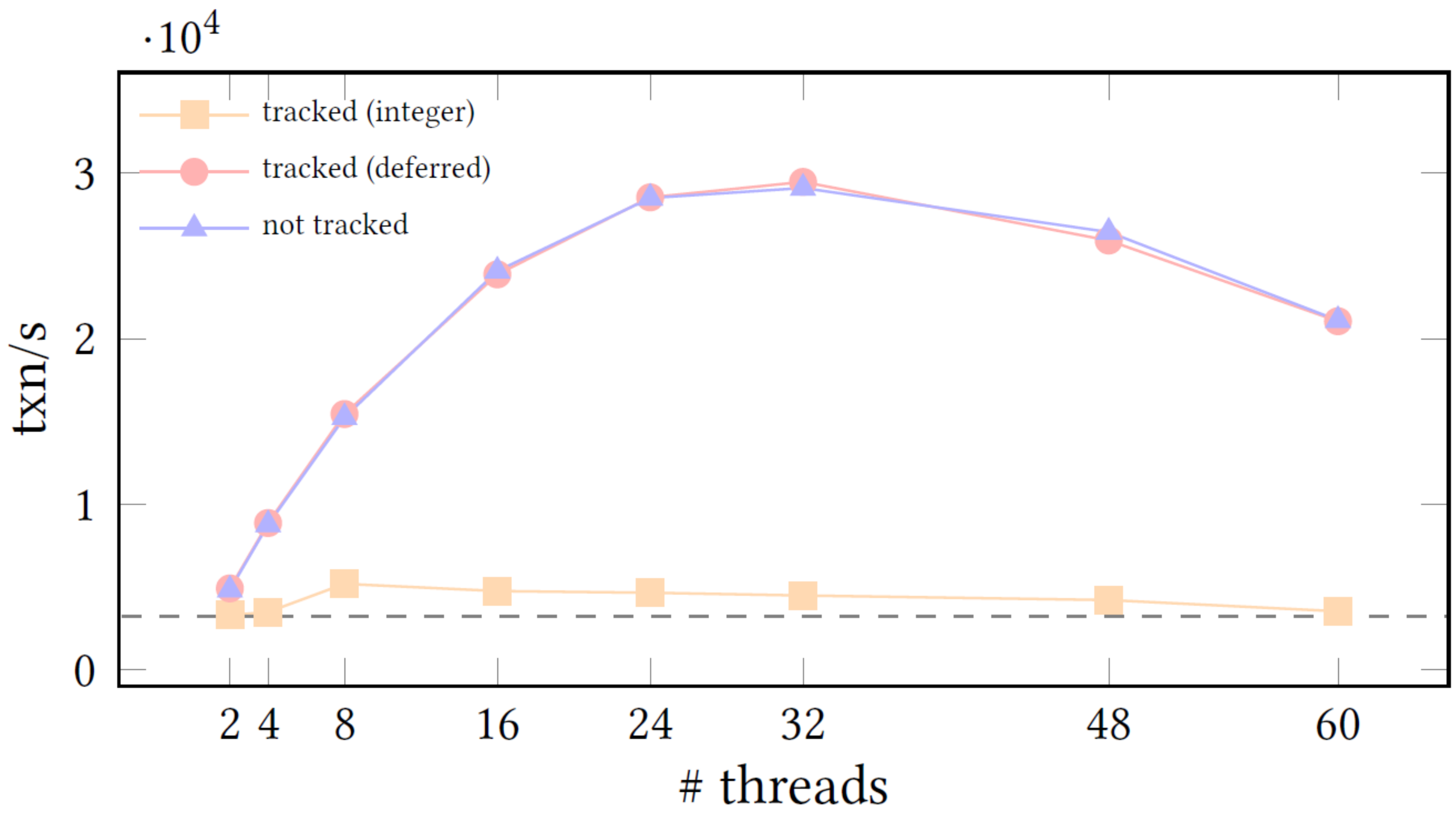
```
module 0x123::counter {
  struct Counter has key { value: u64 }
  entry fun increment(addr: address) acquires Counter {
    let counter = borrow_global_mut<Counter>(addr);
    let value = &mut counter.value;
    *value = *value + 1;
  }
}
```

Language extension: deferred objects

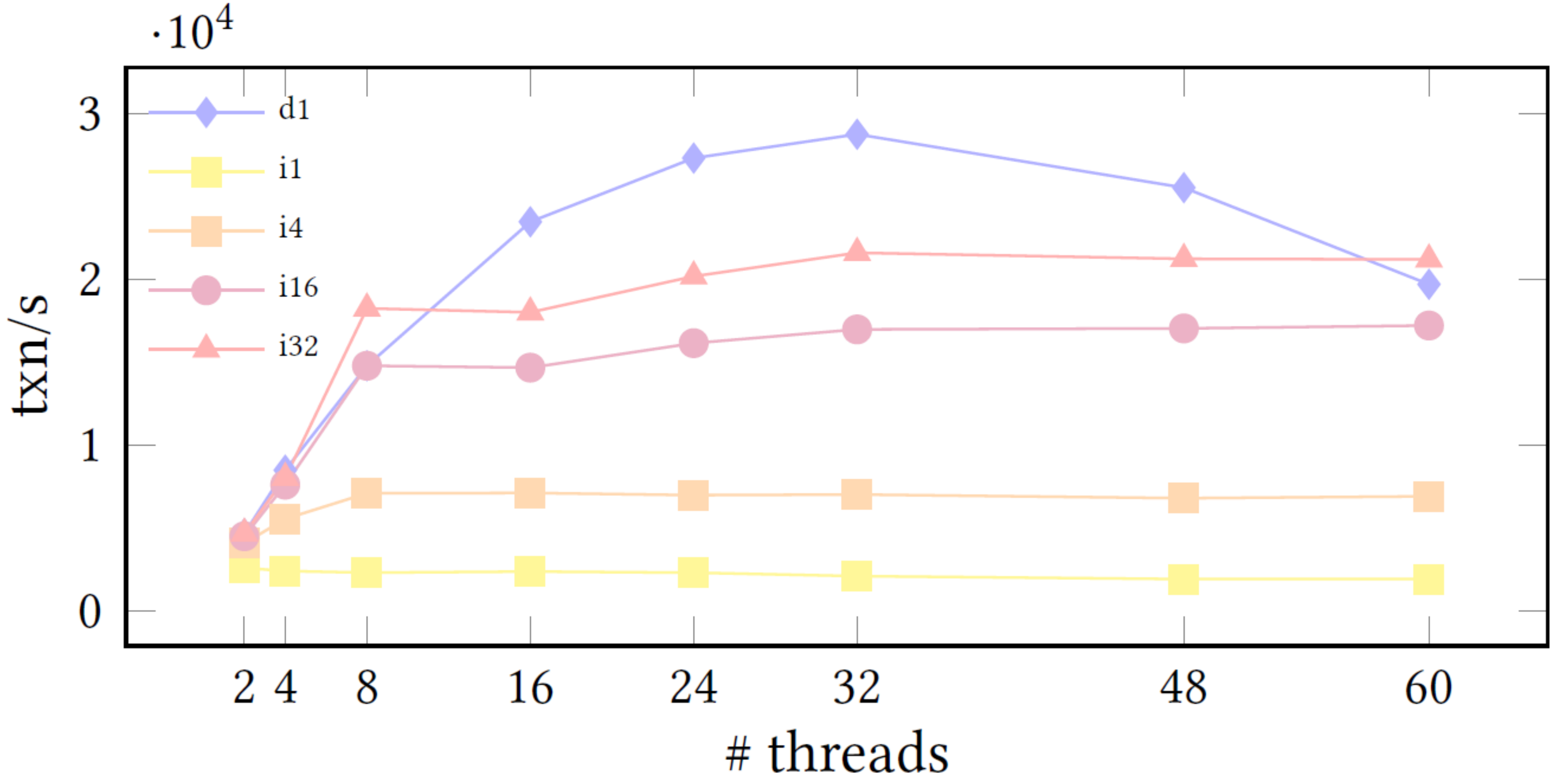
- Language support for updates with pre/post-conditions
  - Effectively removes read-write conflicts
- STM write log collects deltas
- Updates are delayed until commit time

```
struct Balance has key { cnt: Deferred<u64> }

struct NumFailures has key { cnt: Deferred<u64> }
struct NumSuccesses has key { cnt: Deferred<u64> }

entry fun charge_fee(payer: address, fee: u64) {
  let balance = &mut borrow_global_mut<Balance>(payer).cnt;
  if balance.is_at_least(fee) {
    balance.sub(fee);
    let successes = &mut borrow_global_mut<NumSuccesses>(payer).cnt;
    successes.add(1);
  } else {
    let failures = &mut borrow_global_mut<NumFailures>(payer).cnt;
    failures.add(1);
  }
}
```

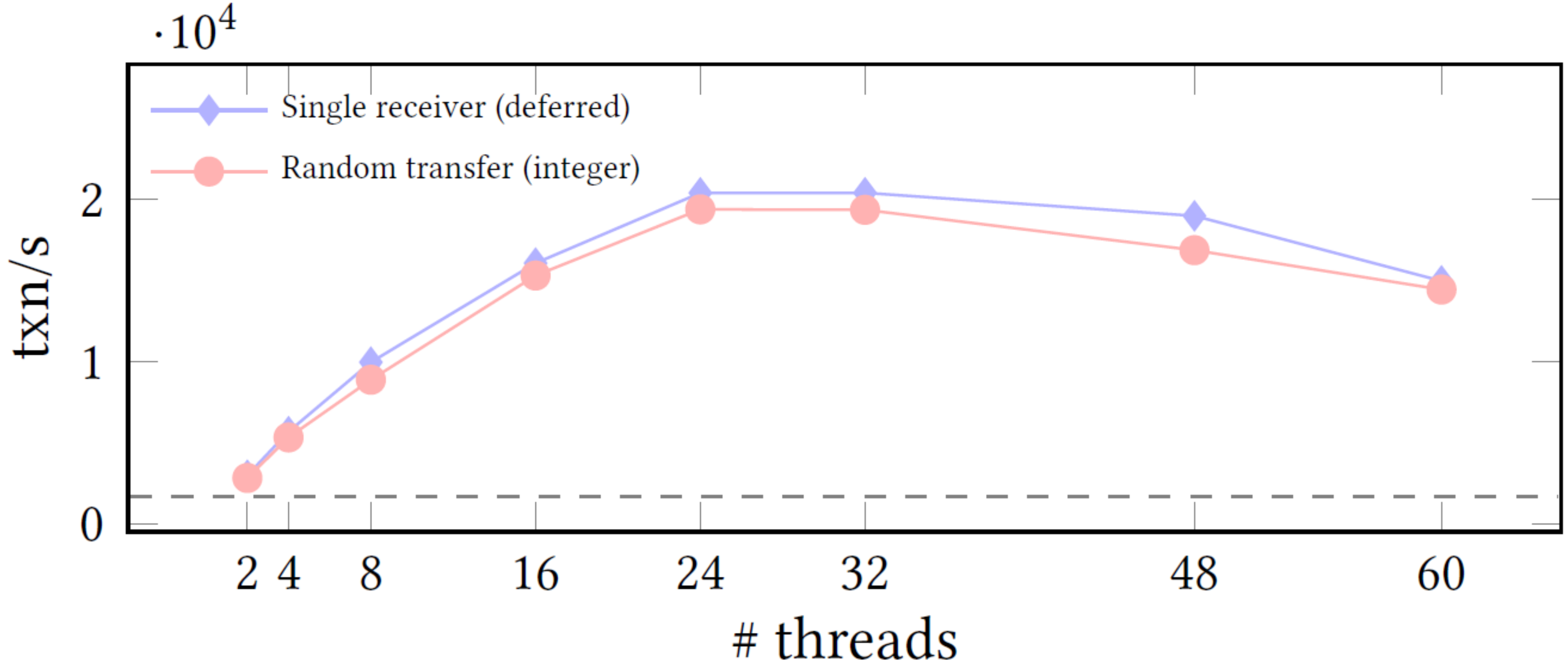
No-op workload



Sponsored workload



Transfer workload

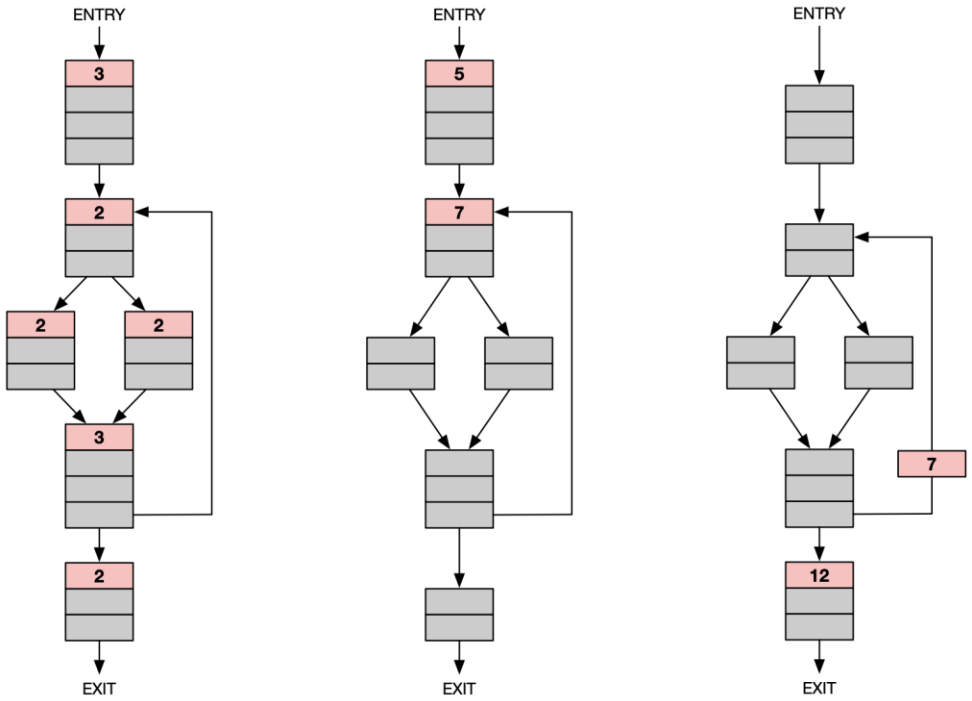


Deferred objects wrap-up

- Deployed at APTOS
- Economic incentives for parallel-friendly workloads?
- Can we automate usage during compilation?

Efficient Smart Contract Execution

- Metering of Smart Contracts
- Parallel Execution of Smart Contracts




```
struct Balance has key { cnt: Deferred<u64> }
struct NumFailures has key { cnt: Deferred<u64> }
struct NumSuccesses has key { cnt: Deferred<u64> }
entry fun charge_fee(payer: address, fee: u64) {
  let balance = &mut borrow_global_mut<Balance>(payer).cnt;
  if balance.is_at_least(fee) {
    balance.sub(fee);
    let successes = &mut borrow_global_mut<NumSuccesses>(payer).cnt;
    successes.add(1);
  } else {
    let failures = &mut borrow_global_mut<NumFailures>(payer).cnt;
    failures.add(1);
  }
}
```