Efficient Concurrent Execution of Smart Contracts in Blockchains using Object-based Transactional Memory

Sweta Kumari

Research Scientist
Huawei Technologies India Pvt. Ltd.

Outline

Outline

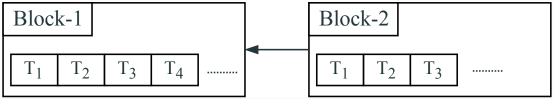
Introduction: Blockchain

- Blockchain is a distributed, decentralized database or ledger of records.

[1] https://bitcoin.org/en/
[2] https://www.ethereum.org/
[3] https://www.hyperledger.org/

Introduction: Blockchain

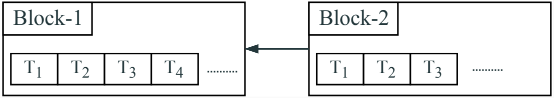
- Blockchain is a distributed, decentralized database or ledger of records.



[1] https://bitcoin.org/en/
[2] https://www.ethereum.org/
[3] https://www.hyperledger.org/

Introduction: Blockchain

- Blockchain is a distributed, decentralized database or ledger of records.

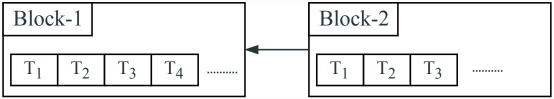


- Miners add blocks to the blockchain, and validators validate each block added to the blockchain.

[1] https://bitcoin.org/en/
[2] https://www.ethereum.org/
[3] https://www.hyperledger.org/

Introduction: Blockchain

- Blockchain is a distributed, decentralized database or ledger of records.



- Miners add blocks to the blockchain, and validators validate each block added to the blockchain.
- Example: Bitcoin[1], **Ethereum**[2], Hyperledger[3], etc.

Execution of Ethereum

[1] https://bitcoin.org/en/
[2] https://www.ethereum.org/
[3] https://www.hyperledger.org/

Introduction: Ethereum High Level Design

- Ethereum nodes form a peer-to-peer system.
- Clients (external to the system) wishing to execute smart contracts, contact a peer of the system.

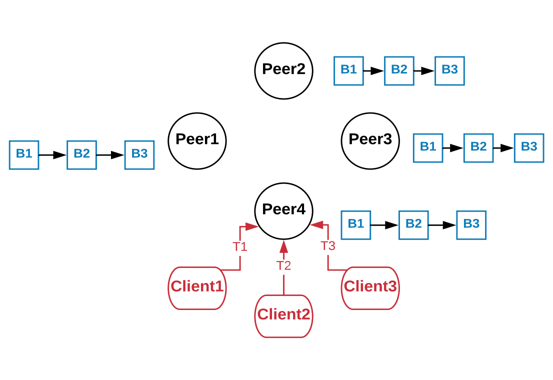


Figure 1: Clients send Transaction T1, T2 and T3 to Miner (Peer4)

Introduction: Ethereum High Level Design

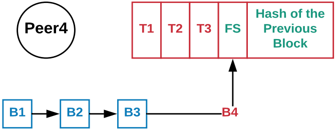


Figure 2: Miner forms a block B4 and computes final state (FS) sequentially

Introduction: Ethereum High Level Design

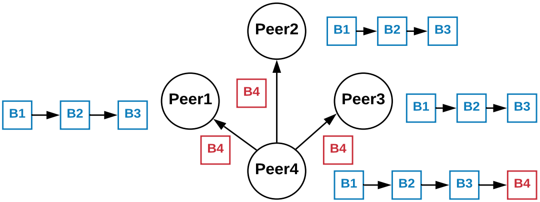


Figure 3: Miner broadcasts the block B4

Introduction: Ethereum High Level Design

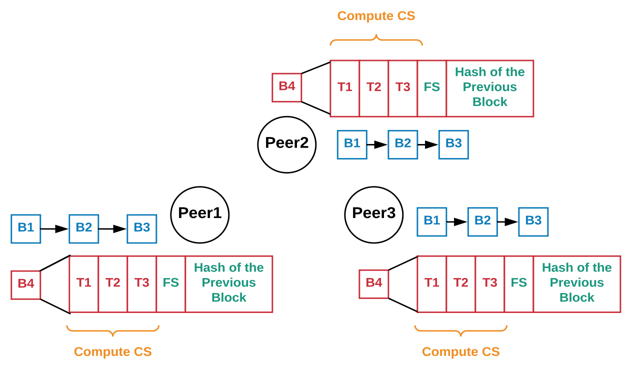


Figure 4: Validators (Peer 1, 2, and 3) compute current state (CS) sequentially

Introduction: Ethereum High Level Design

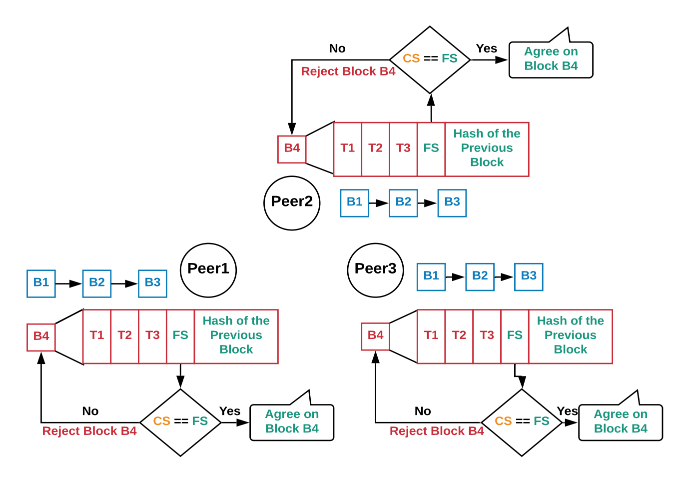


Figure 5: Validators verify the FS and reach the consensus protocol

Introduction: Ethereum High Level Design

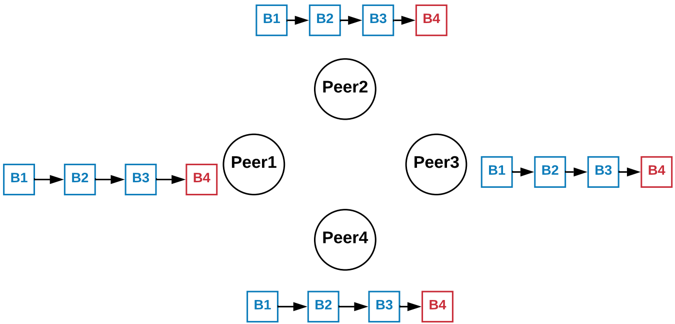


Figure 6: Block B4 successfully added to the blockchain

Introduction: Smart contracts

- Modern blockchain interposes an additional software layer between clients and the blockchain known as *smart contracts*.

[4] https://solidity.readthedocs.io/

Introduction: Smart contracts

- Modern blockchain interposes an additional software layer between clients and the blockchain known as *smart contracts*.
- A smart contract is a piece of code deployed in the blockchain node.

[4] https://solidity.readthedocs.io/

Introduction: Smart contracts

- Modern blockchain interposes an additional software layer between clients and the blockchain known as *smart contracts*.
- A smart contract is a piece of code deployed in the blockchain node.
- Client requests are directed to the smart contracts. Examples: **Coin**, Ballot, Simple Auction, etc.[4]

[4] https://solidity.readthedocs.io/

Introduction: Smart contracts

- Modern blockchain interposes an additional software layer between clients and the blockchain known as *smart contracts*.
- A smart contract is a piece of code deployed in the blockchain node.
- Client requests are directed to the smart contracts. Examples: **Coin**, Ballot, Simple Auction, etc.[4]

Listing 1: Transfer function

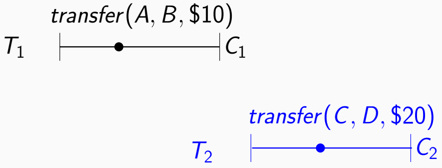
```
transfer(s_id, r_id, amt) {
  if(amt > bal[s_id])
    throw;
  bal[s_id] -= amt;
  bal[r_id] += amt;
}
```

[4] https://solidity.readthedocs.io/

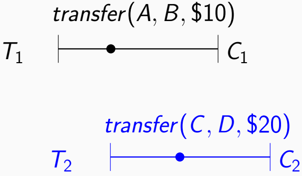
Outline

Bottleneck in Existing Blockchain: Ethereum

- Serial execution of the transactions by miners and validators fails to harness the power of multi-core processors', thus degrading throughput.



(a) Serial Execution of transactions



(b) Concurrent Execution

Figure 7: Motivation towards concurrent execution over serial

Bottleneck in Existing Blockchain: Ethereum

- Serial execution of the transactions by miners and validators fails to harness the power of multi-core processors', thus degrading throughput.

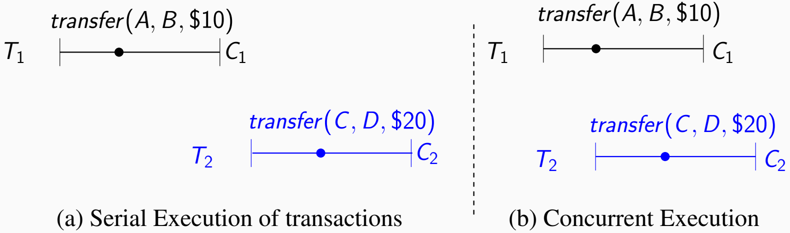


Figure 7: Motivation towards concurrent execution over serial

- By leveraging multiple threads to execute transactions, we can achieve better efficiency and higher throughput.

Outline

Concurrent Execution Challenges (1/2)

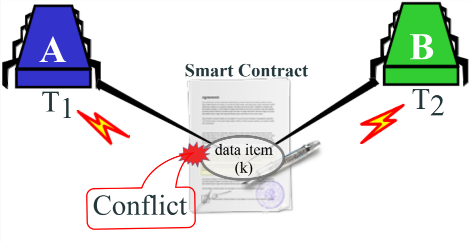


Figure 8: Conflicting access to shared data item.

Concurrent Execution Challenges (1/2)

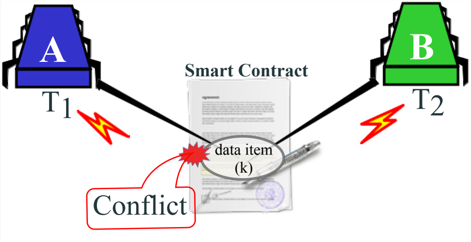


Figure 8: Conflicting access to shared data item.

- Identifying the conflicts at run-time is not straightforward.

Concurrent Execution Challenges (1/2)

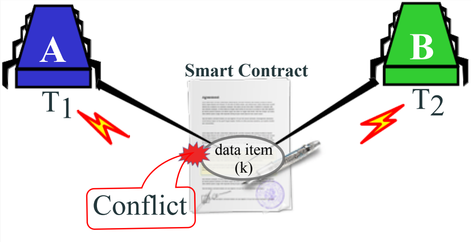


Figure 8: Conflicting access to shared data item.

- Identifying the conflicts at run-time is not straightforward.
- Improper use of locks may lead to deadlock.

Concurrent Execution Challenges (1/2)

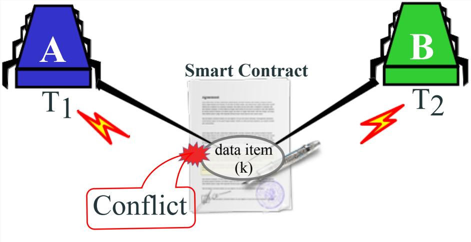


Figure 8: Conflicting access to shared data item.

- Identifying the conflicts at run-time is not straightforward.
- Improper use of locks may lead to deadlock.
- Discovering an equivalent serial schedule of concurrent execution of SCTs is difficult.

Concurrent Execution Challenges (1/2)

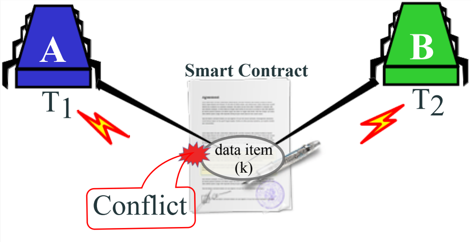


Figure 8: Conflicting access to shared data item.

- Identifying the conflicts at run-time is not straightforward.
- Improper use of locks may lead to deadlock.
- Discovering an equivalent serial schedule of concurrent execution of SCTs is difficult.

Solution: We use *Software Transactional Memory Systems (STMs)* to solve these challenges.

Concurrent Execution Challenges (2/2)

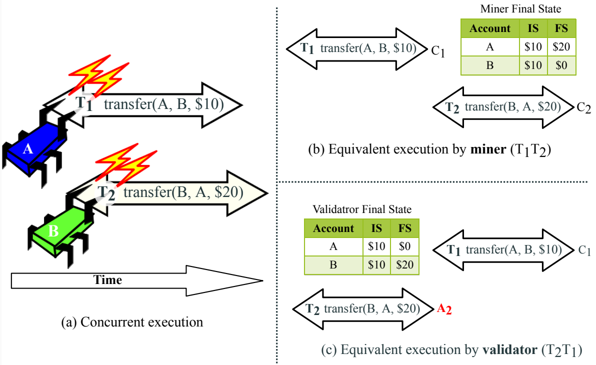
- Validator may incorrectly reject a valid block proposed by the miner. We call such error as **False Block Rejection (FBR)** error.

[5] Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts. PODC, 2017

[6] Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. PDP, 2019

Concurrent Execution Challenges (2/2)

- Validator may incorrectly reject a valid block proposed by the miner. We call such error as **False Block Rejection (FBR)** error.

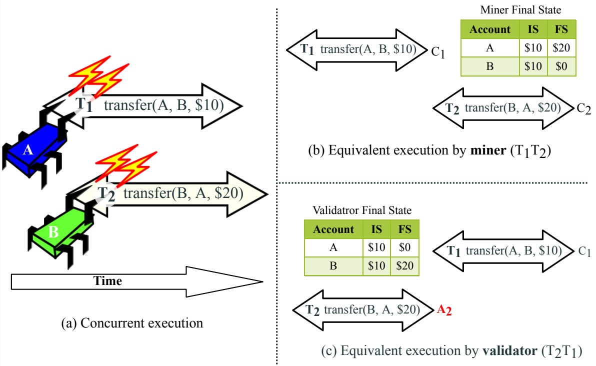


[5] Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts. PODC, 2017

[6] Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. PDP, 2019

Concurrent Execution Challenges (2/2)

- Validator may incorrectly reject a valid block proposed by the miner. We call such error as **False Block Rejection (FBR)** error.



Solution: Miner appends the *Block Graph (BG)*[5,6] in the proposed block to avoid the FBR error.

[5] Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts. PODC, 2017

[6] Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. PDP, 2019

Outline

1. Introduction
2. Bottleneck in Existing Blockchain Design
3. Challenges in Executing Smart Contract Transactions Concurrently
4. Proposed Methodology: Multi-threaded Miner and Validator
5. Experimental Evaluation
6. Real-world applications of Blockchain
7. Conclusion
8. Research Opportunities

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]

- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]

- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.
- Traditional STMs work on read-write primitives. We refer to these as *Read-Write STMs (RWSTMs)*.

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]

- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.
- Traditional STMs work on read-write primitives. We refer to these as *Read-Write STMs (RWSTMs)*.
- *OSTMs* operate on higher level objects rather than primitive reads and writes which act upon memory locations.

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]

- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.
- Traditional STMs work on read-write primitives. We refer to these as *Read-Write STMs (RWSTMs)*.
- *OSTMs* operate on higher level objects rather than primitive reads and writes which act upon memory locations.
- OSTMs provide greater concurrency than RWSTMs. ▸ example

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

Proposed Methodology

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.[7]
- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.
- Traditional STMs work on read-write primitives. We refer to these as *Read-Write STMs (RWSTMs)*.
- *OSTMs* operate on higher level objects rather than primitive reads and writes which act upon memory locations.
- OSTMs provide greater concurrency than RWSTMs. example
- Hash Table based OSTMs export the following methods:
  - STM_begin()
  - STM_insert()
  - STM_delete()
  - STM_lookup()
  - STM_tryC()
  - STM_Abort()

[7] Peri, S., Singh, A., Somani, A.: Efficient means of Achieving Composability using Transactional Memory. NETYS, 2018.

A Thread Safe Integration of STMs in Smart Contracts

Listing 1: Transfer function

```
transfer(s_id, r_id, amt) {
  if(amt > bal[s_id])
    throw;
  bal[s_id] -= amt;
  bal[r_id] += amt;
}
```

A Thread Safe Integration of STMs in Smart Contracts

Listing 1: Transfer function

```
1  transfer(s_id, r_id, amt) {
2    if(amt > bal[s_id])
3      throw;
4    bal[s_id] -= amt;
5    bal[r_id] += amt;
6  }
```

Listing 2: Transfer function using STM

```
7  transfer(s_id, r_id, amt) {
8    t_id = STM_begin();
9    s_bal = STM_lookup(s_id);
10   if(amt > s_bal) {
11     abort(t_id);
12     throw;
13   }
14   STM_delete(s_id, amt);
15   STM_insert(r_id, amt);
16   if(STM_tryC(t_id)!= SUCCESS)
17     goto Line 8;//Trans aborted
18 }
```

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- Edges of the BG depends on the conflicts given by the OSTMs.

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- Edges of the BG depends on the conflicts given by the OSTMs.

$$Conflicting\ Operations = \begin{cases} STM\_lookup_i() & - \quad STM\_tryC_j() \\ STM\_delete_i() & - \quad STM\_tryC_j() \\ STM\_tryC_i() & - \quad STM\_tryC_j() \\ STM\_tryC_i() & - \quad STM\_delete_j() \\ STM\_tryC_i() & - \quad STM\_lookup_j() \end{cases} \tag{1}$$

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- Edges of the BG depends on the conflicts given by the OSTMs.

$$Conflicting\ Operations = \begin{cases} STM\_lookup_i() & - & STM\_tryC_j() \\ STM\_delete_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_delete_j() \\ STM\_tryC_i() & - & STM\_lookup_j() \end{cases} \quad (1)$$

- **Multi-threaded miner** uses addVert() and addEdge() methods of BG.

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- Edges of the BG depends on the conflicts given by the OSTMs.

$$Conflicting\ Operations = \begin{cases} STM\_lookup_i() & - & STM\_tryC_j() \\ STM\_delete_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_delete_j() \\ STM\_tryC_i() & - & STM\_lookup_j() \end{cases} \quad (1)$$

- **Multi-threaded miner** uses addVert() and addEdge() methods of BG.
- Later, validators re-execute the same SCTs concurrently and deterministically relying on the BG.

Block Graph (1/2)

- Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- Edges of the BG depends on the conflicts given by the OSTMs.

$$Conflicting\ Operations = \begin{cases} STM\_lookup_i() & - & STM\_tryC_j() \\ STM\_delete_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_tryC_j() \\ STM\_tryC_i() & - & STM\_delete_j() \\ STM\_tryC_i() & - & STM\_lookup_j() \end{cases} \quad (1)$$

- **Multi-threaded miner** uses addVert() and addEdge() methods of BG.
- Later, validators re-execute the same SCTs concurrently and deterministically relying on the BG.
- Two SCTs that do not have a path can execute concurrently.

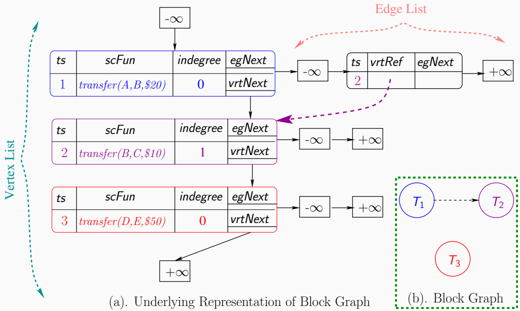
Block Graph (2/2)

- **SMV** uses searchGlobal() and decInCount() methods of BG. ▸ SMV

[8] Herlihy, M., Koskinen, E.: Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. PPoPP, 2008.

Block Graph (2/2)

- **SMV** uses searchGlobal() and decInCount() methods of BG. ▸ SMV



(a). Underlying Representation of Block Graph (b). Block Graph

Figure 9: Data structure of BG

8 Herlihy, M., Koskinen, E.: Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. PPoPP, 2008.

Block Graph (2/2)

- **SMV** uses searchGlobal() and decInCount() methods of BG. ▸ SMV

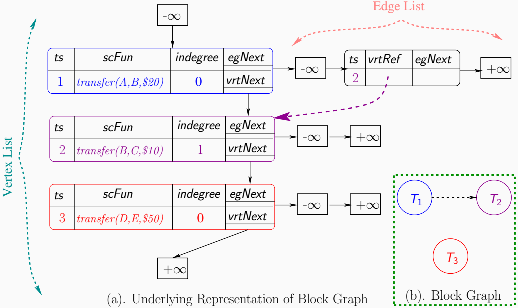


Figure 9: Data structure of BG

- OSTMs[8] have fewer conflicts than RWSTMs which in turn, allows validators to execute more SCTs concurrently.

[8] Herlihy, M., Koskinen, E.: Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. PPoPP, 2008.

Block Graph (2/2)

- **SMV** uses searchGlobal() and decInCount() methods of BG. ▸ SMV

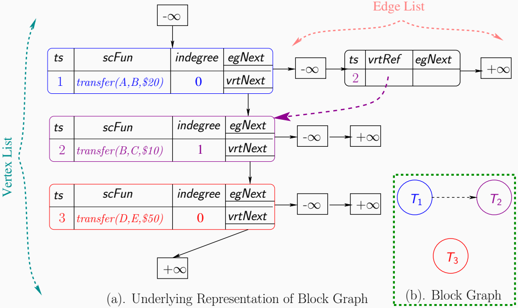


Figure 9: Data structure of BG

- OSTMs[8] have fewer conflicts than RWSTMs which in turn, allows validators to execute more SCTs concurrently.
- This also reduces the size of the BG leading to a smaller communication cost than RWSTMs.

[8] Herlihy, M., Koskinen, E.: Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. PPoPP, 2008.

Greater Concurrency: Multi-Version OSTM based Miner

- *Multi-Version OSTMs (MVOSTMs)*[9] maintain multiple versions for each shared data item and provide greater concurrency relative to *Single-Version OSTMs (SVOSTMs)*.

9 Juyal, C., Kulkarni, S., Kumari, S., Peri, S., Somani, A.: An innovative approach to achieve compositionality efficiently using multi-version object based transactional systems. SSS, 2018.

Greater Concurrency: Multi-Version OSTM based Miner

- *Multi-Version OSTMs (MVOSTMs)*[9] maintain multiple versions for each shared data item and provide greater concurrency relative to *Single-Version OSTMs (SVOSTMs)*.
- MVOSTM-based BG has fewer edges than an SVOSTM-based BG, and further reduces the size of the BG leading to a smaller communication cost.

▸ MVOSTM

9 Juyal, C., Kulkarni, S., Kumari, S., Peri, S., Somani, A.: An innovative approach to achieve compositionality efficiently using multi-version object based transactional systems. SSS, 2018.

Outline

Experimental Evaluation (1/2)

- In Ethereum blockchain, smart contracts are written in Solidity language, which runs on Ethereum Virtual Machine (EVM).

Experimental Evaluation (1/2)

- In Ethereum blockchain, smart contracts are written in Solidity language, which runs on Ethereum Virtual Machine (EVM).
- EVM does not supports multi-threading.

Experimental Evaluation (1/2)

- In Ethereum blockchain, smart contracts are written in Solidity language, which runs on Ethereum Virtual Machine (EVM).
- EVM does not supports multi-threading.
- We converted smart contracts from Solidity to **C++** language for multi-threaded execution.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
  1. **Coin:** A simple cryptocurrency contract.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.

    1. **Coin:** A simple cryptocurrency contract.
    2. **Ballot:** An electronic voting contract.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
  1. **Coin:** A simple cryptocurrency contract.
  2. **Ballot:** An electronic voting contract.
  3. **Simple Auction:** An online auction contract.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
    1. **Coin:** A simple cryptocurrency contract.
    2. **Ballot:** An electronic voting contract.
    3. **Simple Auction:** An online auction contract.
    4. **Mix:** Combination of above three contracts in equal proportion.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
  1. **Coin:** A simple cryptocurrency contract.
  2. **Ballot:** An electronic voting contract.
  3. **Simple Auction:** An online auction contract.
  4. **Mix:** Combination of above three contracts in equal proportion.
- We ran our experiments on Intel (R) Xeon (R) CPU E5-2690 that supports 56 hardware threads and 32GB RAM.

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
    1. **Coin:** A simple cryptocurrency contract.
    2. **Ballot:** An electronic voting contract.
    3. **Simple Auction:** An online auction contract.
    4. **Mix:** Combination of above three contracts in equal proportion.
- We ran our experiments on Intel (R) Xeon (R) CPU E5-2690 that supports 56 hardware threads and 32GB RAM.
- We consider two workloads:

Experimental Evaluation (2/2)

- We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.
    1. **Coin:** A simple cryptocurrency contract.
    2. **Ballot:** An electronic voting contract.
    3. **Simple Auction:** An online auction contract.
    4. **Mix:** Combination of above three contracts in equal proportion.
- We ran our experiments on Intel (R) Xeon (R) CPU E5-2690 that supports 56 hardware threads and 32GB RAM.
- We consider two workloads:

Workload	SCTs	Threads	Shared data items
Workload 1 (W1)	50 - 300	50	500
Workload 2 (W2)	100	10 - 60	500

Results: Multi-threaded Miner Speedup

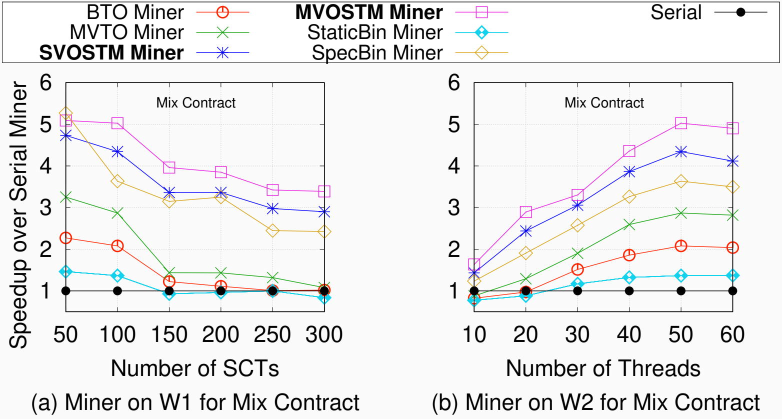


Figure 10: Speedup of Multi-threaded miner over Serial miner

- **MVOSTM, SVOSTM**, MVTO, BTO, Speculative Bin, and Static Bin miner provide an average speedup of **3.91**×**, 3.41**×, 1.98×, 1.5×, 3.02×, and 1.12×, over Serial miner, respectively.

Results: SMV Speedup

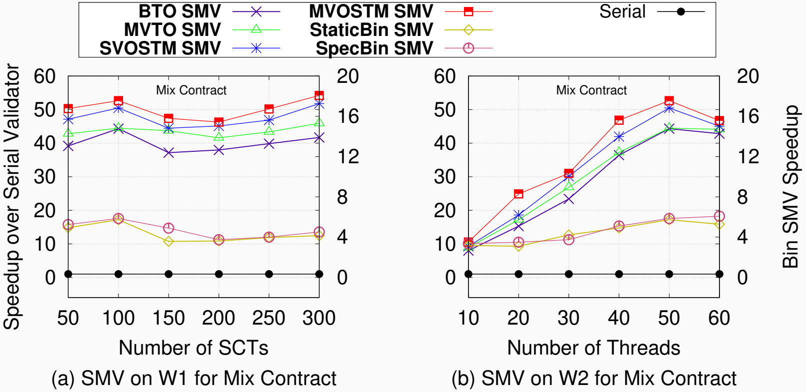


Figure 11: Speedup of SMV over Serial validator

- **MVOSTM, SVOSTM, MVTO, BTO, Speculative Bin,** and **Static Bin Decentralized SMVs** provide an average speedup of **48.45**×, **46.35**×, **43.89**×, **41.44**×, **5.39**×, and **4.81**× over Serial validator, respectively.

Outline

Real-world applications of Blockchain

- Automating and securely storing user records such as land sale documents, vehicle, and insurance records.

Real-world applications of Blockchain

- Automating and securely storing user records such as land sale documents, vehicle, and insurance records.
- Blockchain-based Audit log

Real-world applications of Blockchain

- Automating and securely storing user records such as land sale documents, vehicle, and insurance records.
- Blockchain-based Audit log
- Supply Chain Management.

Real-world applications of Blockchain

- Automating and securely storing user records such as land sale documents, vehicle, and insurance records.
- Blockchain-based Audit log
- Supply Chain Management.
- Health record.
- Decentralized Education System.

Outline

Conclusion

- We developed an efficient framework for concurrent execution of SCTs by a multi-threaded miner using two protocols, SVOSTM and MVOSTM of optimistic STMs[10].

[10] Technical report: https://arxiv.org/abs/1904.00358

Conclusion

- We developed an efficient framework for concurrent execution of SCTs by a multi-threaded miner using two protocols, SVOSTM and MVOSTM of optimistic STMs[10].
- To avoid FBR errors, the multi-threaded miner captures the dependencies among SCTs in the form of a BG.

[10] Technical report: https://arxiv.org/abs/1904.00358

Conclusion

- We developed an efficient framework for concurrent execution of SCTs by a multi-threaded miner using two protocols, SVOSTM and MVOSTM of optimistic STMs[10].
- To avoid FBR errors, the multi-threaded miner captures the dependencies among SCTs in the form of a BG.
- The proposed approach achieves significant performance gain over the state-of-the-art SCTs execution framework.

[10] Technical report: https://arxiv.org/abs/1904.00358

Outline

Research Opportunities

- A malicious miner can intentionally append a BG in a block with additional edges to delay other miners. Preventing such a malicious miner would be an immediate future work.

Research Opportunities

- A malicious miner can intentionally append a BG in a block with additional edges to delay other miners. Preventing such a malicious miner would be an immediate future work.
- BG consumes space. So, constructing storage optimal BG is an interesting challenge.

Research Opportunities

- A malicious miner can intentionally append a BG in a block with additional edges to delay other miners. Preventing such a malicious miner would be an immediate future work.
- BG consumes space. So, constructing storage optimal BG is an interesting challenge.
- Implementing our proposed approach in other blockchains such as Bitcoin, Hyperledger, and EOSIO is an exciting exercise.

Research Opportunities

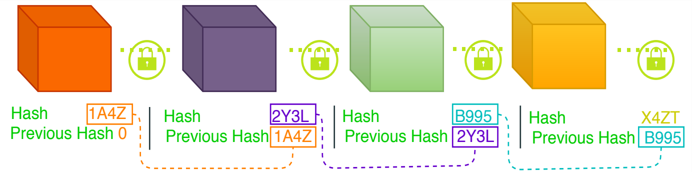
- A malicious miner can intentionally append a BG in a block with additional edges to delay other miners. Preventing such a malicious miner would be an immediate future work.
- BG consumes space. So, constructing storage optimal BG is an interesting challenge.
- Implementing our proposed approach in other blockchains such as Bitcoin, Hyperledger, and EOSIO is an exciting exercise.
- EVM does not support multi-threading, so, another research direction is to design a multi-threaded EVM.

Research Opportunities

- A malicious miner can intentionally append a BG in a block with additional edges to delay other miners. Preventing such a malicious miner would be an immediate future work.
- BG consumes space. So, constructing storage optimal BG is an interesting challenge.
- Implementing our proposed approach in other blockchains such as Bitcoin, Hyperledger, and EOSIO is an exciting exercise.
- EVM does not support multi-threading, so, another research direction is to design a multi-threaded EVM.

Thank You!

Introduction: Blockchain



return

Read-Write STM (RWSTM) v/s Object-based STM (OSTM)

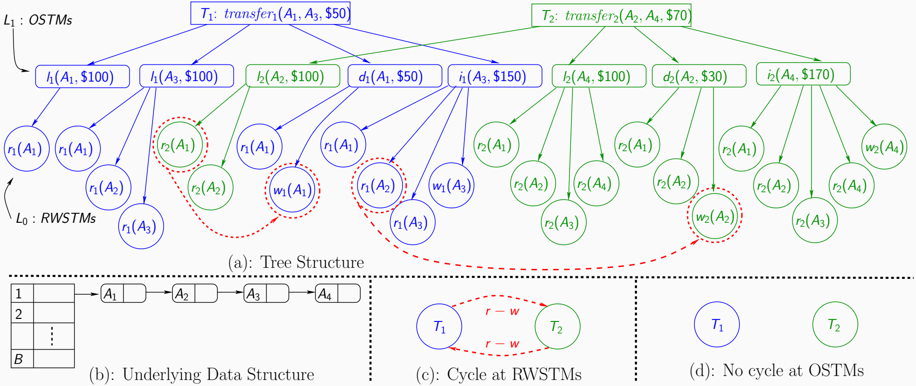
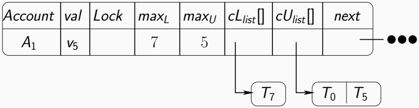
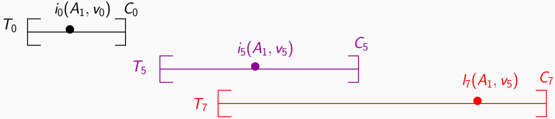


Figure 12: (a) Two SCTs T_1 and T_2 in the form of a tree structure which is working on a hash-table with B buckets where four accounts (shared data items) A_1, A_2, A_3 and A_4 are stored in the form of a list depicted in (b). T_1 transfers \$50 from A_1 to A_3 and T_2 transfers \$70 from A_2 to A_4. After checking the sufficient balance using lookup (l), SCT T_1 deletes (d) \$50 from A_1 and inserts (i) it to A_3 at higher-level (L_1). At lower-level 0 (L_0), these operations involve read (r) and write (w) to both accounts A_1 and A_3. Since, its conflict graph has a cycle either T_1 or T_2 has to abort (see (c)); However, execution at L_1 depicts that both transactions are working on different accounts and the higher-level methods are isolated. So, we can prune this tree and isolate the transactions at higher-level with equivalent serial schedule $T_1 T_2$ or $T_2 T_1$ as shown in (d).

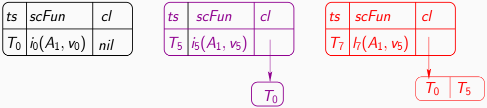
Data Structure of SVOSTM to Maintain Conflicts



(a) Structure of Shared data-item



(b) Timeline View



(c) Transactions Conflict List

Figure 13: Underlying Data Structure of SVOSTM

return

Block Graph: Components

- MVOSTM uses multiple versions and satisfies opacity.

Block Graph: Components

- MVOSTM uses multiple versions and satisfies opacity.
- In **MVOSTM** two types of edges based on *mvoconflicts*:

Block Graph: Components

- MVOSTM uses multiple versions and satisfies opacity.
- In **MVOSTM** two types of edges based on *mvoconflicts*:
  1. **Return value from (rvf) edge:** If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$ then there exist an *rvf edge* from T_i to T_j, i.e., $T_i \rightarrow T_j$;

Block Graph: Components

- MVOSTM uses multiple versions and satisfies opacity.
- In **MVOSTM** two types of edges based on *mvoconflicts*:
  1. **Return value from (rvf) edge:** If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$ then there exist an *rvf edge* from T_i to T_j, i.e., $T_i \rightarrow T_j$;
  2. **Multi-version (mv) edge:** consider a triplet, $STM\_tryC_i(), rv_m(k, v)$, $STM\_tryC_j()$ in which $(updSet(T_i) \cap updSet(T_j) \cap rvSet(T_m) \neq \emptyset)$, (two committed transactions T_i and T_j update the key k with value v and u respectively) and $(u, v \neq \mathcal{A})$; then

Block Graph: Components

- MVOSTM uses multiple versions and satisfies opacity.
- In **MVOSTM** two types of edges based on *mvoconflicts*:
  1. **Return value from (rvf) edge:** If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$ then there exist an *rvf edge* from T_i to T_j, i.e., $T_i \rightarrow T_j$;
  2. **Multi-version (mv) edge:** consider a triplet, $STM\_tryC_i(), rv_m(k, v)$, $STM\_tryC_j()$ in which $(updSet(T_i) \cap updSet(T_j) \cap rvSet(T_m) \neq \emptyset)$, (two committed transactions T_i and T_j update the key k with value v and u respectively) and $(u, v \neq \mathcal{A})$; then

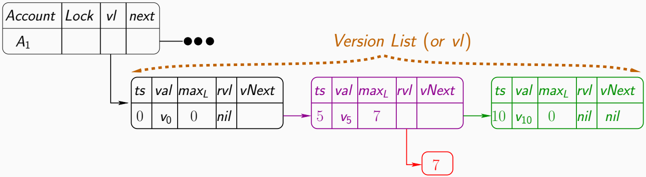
     2.1 If $STM\_tryC_i() <_H STM\_tryC_j()$ then there exist a *mv edge* from T_m to T_j.

Block Graph: Components

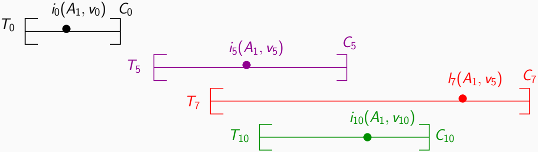
- MVOSTM uses multiple versions and satisfies opacity.

- In **MVOSTM** two types of edges based on *mvoconflicts*:
  1. **Return value from (rvf) edge:** If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$ then there exist an *rvf edge* from T_i to T_j, i.e., $T_i \rightarrow T_j$;
  2. **Multi-version (mv) edge:** consider a triplet, $STM\_tryC_i(), rv_m(k, v)$, $STM\_tryC_j()$ in which $(updSet(T_i) \cap updSet(T_j) \cap rvSet(T_m) \neq \emptyset)$, (two committed transactions T_i and T_j update the key k with value v and u respectively) and $(u, v \neq \mathcal{A})$; then
     - 2.1 If $STM\_tryC_i() <_H STM\_tryC_j()$ then there exist a *mv edge* from T_m to T_j.
     - 2.2 If $STM\_tryC_j() <_H STM\_tryC_i()$ then there exist a *mv edge* from T_j to T_i.

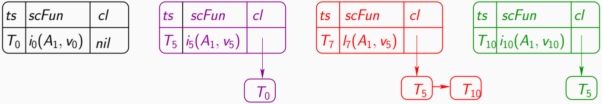
Data Structure of MVOSTM to Maintain Conflicts



(a) Structure of Shared data-item with Version List



(b) Timeline View



(c) Transactions Conflict List

Figure 14: Underlying Data Structure of SVOSTM

Single-version v/s Multi-version OSTMs

- *Multi-version OSTMs (MVOSTMs)* maintain multiple versions for each shared data item (object) and provide greater concurrency relative to traditional *single-version OSTMs (SVOSTMs)*.

Single-version v/s Multi-version OSTMs

- *Multi-version OSTMs (MVOSTMs)* maintain multiple versions for each shared data item (object) and provide greater concurrency relative to traditional *single-version OSTMs (SVOSTMs)*.

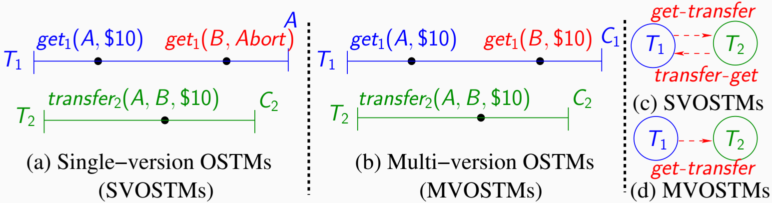


Figure 15: (a) Transaction T_1 gets the balance of two accounts A and B (both initially \$10), while transaction T_2 transfers \$10 from A to B and T_1 aborts. Since, its conflict graph has a cycle (see (c)); (b) When T_1 and T_2 are executed by MVOSTM, T_1 can read the old versions of A and B. This can be serialized, as shown in (d).

Correctness Criteria: Opacity

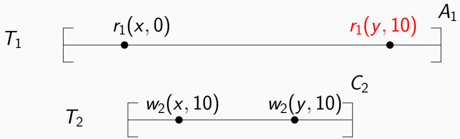


Figure 16: History H is not Opaque

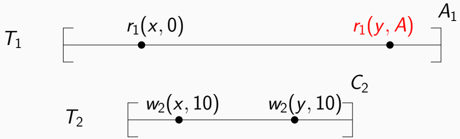


Figure 17: Opaque History H

return

Smart Multi-threaded Validator

SMV maintains two global counters (gUC: global update counter and gLC: global lookup counter) and two local counters (lUC and lLC) for each shared data item k to identifies the EMB error.

Smart Multi-threaded Validator

SMV maintains two global counters (gUC: global update counter and gLC: global lookup counter) and two local counters (lUC and lLC) for each shared data item k to identifies the EMB error.

Lookup(k):

- **If**(k.gUC == k.lUC)
  1. Atomically increment the global lookup counter, k.gLC.
  2. Increment k.lLC by 1.
  3. Lookup key k from a shared memory.

  else miner is malicious.

Smart Multi-threaded Validator

SMV maintains two global counters (gUC: global update counter and gLC: global lookup counter) and two local counters (lUC and lLC) for each shared data item k to identifies the EMB error.

Lookup(k):

- **If**(k.gUC == k.lUC)
  1. Atomically increment the global lookup counter, k.gLC.
  2. Increment k.lLC by 1.
  3. Lookup key k from a shared memory.

  else miner is malicious.

Insert(k, v)/Delete(k):

- **If**(k.gLC == k.lLC && k.gUC == k.lUC)
  1. Atomically increment the global update counter, k.gUC.
  2. Increment k.lUC by 1.
  3. Insert/delete key k to/from shared memory.

  else miner is malicious.

return

Algorithm 1: SMV(scFun): Execute scFun with atomic global lookup/update counter.

```
// scFun is a list of steps.
while (scFun.steps.hasNext()) do
    curStep = scFun.steps.next(); //Get the next step to execute.
    switch (curStep) do
        case lookup(k): do
            // Check for update counter (uc) value.
            if (k.gUC == k.lUC_i) then
                Atomically increment the global lookup counter, k.gLC;
                Increment k.lLC_i by 1;//Maintain k.lLC_i in transaction local log.
                Lookup k from a shared memory;
            end
            else
                return ⟨Miner is malicious⟩;
            end
        end
        case insert(k, v): do
            // Check lookup/update counter value.
            if ((k.gLC == k.lLC_i) && (k.gUC == k.lUC_i)) then
                Atomically increment the global update counter, k.gUC;
                Increment k.lUC_i by 1;//Maintain k.lUC_i in transaction local log.
                Insert k in shared memory with value v;
            end
            else
                return ⟨Miner is malicious⟩;
            end
        end
    end
end
Atomically decrements the k.gLC and k.gUC corresponding to each shared data-item key k;
```

▸ return

```
// scFun is a list of steps.
while (scFun.steps.hasNext()) do
    curStep = scFun.steps.next(); //Get the next step to execute.
    switch (curStep) do
        case delete(k): do
            // Check lookup/update counter value.
            if ((k.gLC == k.lLC_i) && (k.gUC == k.lUC_i)) then
                Atomically increment the global update counter, k.gUC;
                Increment k.lUC_i by 1; //Maintain k.lUC_i in transaction local.
                Delete k in shared memory;
            end
            else
                return ⟨Miner is malicious⟩;
            end
        end
    end
end
Atomically decrements the k.gLC and k.gUC corresponding to each shared data-item key k;
```

▸ return

Results: BG Depth

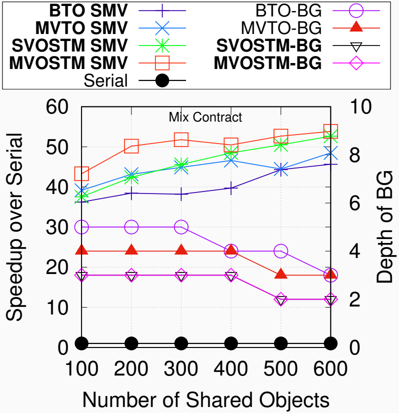


Figure 18: Speedup of SMV over serial and depth of BG for W3

Results: Dependencies in BG

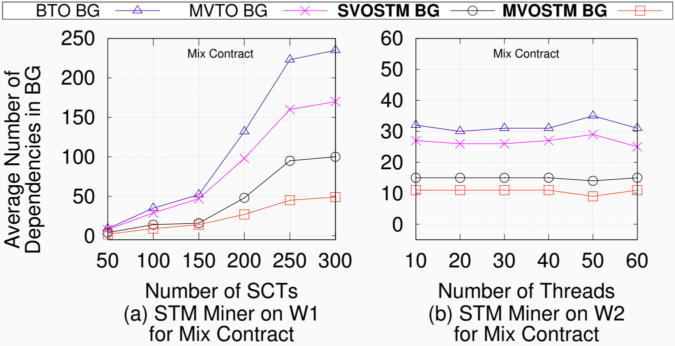


Figure 19: Average number of dependencies in BG for mix contract on W1 and W2

Results: Average Speedup by Multi-threaded Miner

Table 1: Overall average speedup on all workloads by multi-threaded miner over serial miner

Contract	Multi-threaded Miner					
	BTO Miner	**MVTO Miner**	**SVOSTM Miner**	**MVOSTM Miner**	**StaticBin Miner**	**SpecBin Miner**
Coin	1.596	1.959	4.391	5.572	1.279	6.689
Ballot	0.960	1.065	2.229	2.431	1.175	2.233
Auction	2.305	2.675	3.456	3.881	1.524	2.232
Mix	1.596	2.118	3.425	3.898	1.102	3.080
Total Avg. Speedup	*1.61*	*1.95*	*3.38*	*3.95*	*1.27*	*3.56*

Results: Average Speedup by Smart Multi-threaded Validator

Table 2: Overall average speedup on all workloads by SMV over serial validator

Contract	Smart Multi-threaded Validator (SMV)					
	BTO SMV	**MVTO SMV**	**SVOSTM SMV**	**MVOSTM SMV**	**StaticBin SMV**	**SpecBin SMV**
Coin	26.576	28.635	30.344	32.864	5.296	7.565
Ballot	26.037	28.333	33.695	36.698	3.570	3.780
Auction	27.772	31.781	29.803	32.709	4.694	5.214
Mix	36.279	39.304	42.139	45.332	4.279	4.463
Total Avg. Speedup	*29.17*	*32.01*	*34.00*	*36.90*	*4.46*	*5.26*

▸ return