DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain*

(Annual Progress Seminar)

Parwat Singh Anjana (CS17RESCH11004)

Guided by:
Dr. Sathya Peri, Associate Professor

Department of Computer Science and Engineering,
Indian Institute of Technology Hyderabad, India

*Accepted at Concurrency and Computation: Practice and Experience (CCPE), Wiley, 2021
1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
Introduction: Blockchain

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

2. https://www.ethereum.org/
3. https://www.hyperledger.org/
Introduction: Blockchain

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

![Diagram of blocks with transactions](image)

- Miners add blocks to the blockchain, and validators validate each block added to the blockchain.
- Example: Bitcoin\(^1\), Ethereum\(^2\), Fabric, Sawtooth\(^3\), etc.

---

\(^1\)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\(^2\), Fabric, Sawtooth\(^3\), etc.

---

2. [https://www.ethereum.org/](https://www.ethereum.org/)
3. [https://www.hyperledger.org/](https://www.hyperledger.org/)
1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
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.

<table>
<thead>
<tr>
<th>Transfer function</th>
<th>Concurrent Execution</th>
<th>Serial Execution of transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer (s_id, r_id, amt)</td>
<td>C1: C2:</td>
<td>T1: T2:</td>
</tr>
<tr>
<td></td>
<td>transfer (C, D, $20)</td>
<td>transfer (A, B, $10)</td>
</tr>
<tr>
<td></td>
<td>transfer (A, B, $10)</td>
<td>C1: C2: transfer (C, D, $20)</td>
</tr>
<tr>
<td></td>
<td>Figure 1: Motivation towards concurrent execution over serial Parwat Singh Anjana (CS17RESCH11004) Guided by: Dr. Sathya Peri, Associate Professor DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain*</td>
<td></td>
</tr>
</tbody>
</table>
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.

- By leveraging multiple threads to execute transactions, we can achieve better efficiency and higher throughput.
Serial execution of the transactions by miners and validators fails to harness the power of multi-core processors’, thus degrading throughput.

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

**Listing 1:** Transfer function

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

**Figure 1:** Motivation towards concurrent execution over serial
1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
Parallel Execution Challenges (1/4)

Figure 2: 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.
Parallel Execution Challenges (1/4)

Figure 2: Conflicting access to shared data item.

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

\( \text{Smart Contract A B data item (k) T1 T2 Conflict} \)

Solution: We use Software Transactional Memory Systems (STMs) to solve these challenges.
Identifying the conflicts at run-time is not straightforward. Improper use of locks may lead to deadlock.
Parallel Execution Challenges (1/4)

- 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.

Figure 2: Conflicting access to shared data item.
Parallel Execution Challenges (1/4)

Figure 2: 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.
• Validator may incorrectly reject a valid block proposed by the miner. We call such error the **False Block Rejection (FBR)** error.
• Validator may incorrectly reject a valid block proposed by the miner. We call such error the **False Block Rejection (FBR)** error.
Solution: Miner appends the *Block Graph (BG)*\(^4,5\) in the block to avoid the FBR error.

\(^4\)Dickerson et al., “Adding Concurrency to Smart Contracts.” PODC, 2017
\(^5\)Anjana et al., “An efficient framework for optimistic concurrent execution of smart contracts.” PDP, 2019
**Solution:** Miner appends the *Block Graph (BG)*\(^4,5\) in the block to avoid the FBR error.

---

\(^4\) Dickerson et al., “Adding Concurrency to Smart Contracts.” PODC, 2017

\(^5\) Anjana et al., “An efficient framework for optimistic concurrent execution of smart contracts.” PDP, 2019
Parallel Execution Challenges (4/4)

- A *Malicious miner* can send an incorrect Block Graph to harm the blockchain, missing some edges, e.g., to cause *double spending*. We call such error the **Edge Missing BG (EMB)** error.
A Malicious miner can send an incorrect Block Graph to harm the blockchain, missing some edges, e.g., to cause double spending. We call such error the Edge Missing BG (EMB) error.

Validator's Final State is the same as Miner's Final State so Accepts the block

Validator Final State

Account IS FS
A $10 $5
B $10 $20
C $10 $15

Solution:
We propose a Smart Multi-threaded Validator (SMV) to detect EMB error and rejects the corresponding blocks.
A Malicious miner can send an incorrect Block Graph to harm the blockchain, missing some edges, e.g., to cause double spending. We call such error the **Edge Missing BG (EMB)** error.

**Solution:** We propose a *Smart Multi-threaded Validator (SMV)* to detect EMB error and rejects the corresponding blocks.
1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
Proposed Approach: DiPETrans Framework

- We proposed a *DiPETrans* framework\(^6\) for parallel execution of the transactions at miners and validators, based on transaction shards identified using static analysis.

\(^6\)Accepted at the journal of *Concurrency and Computation: Practice and Experience (CCPE)*, Wiley, 2021.
Proposed Approach: DiPETrans Framework

- We proposed a *DiPETrans* framework\(^6\) for parallel execution of the transactions at miners and validators, based on transaction shards identified using static analysis.

\(^6\) Accepted at the journal of *Concurrency and Computation: Practice and Experience (CCPE)*, Wiley, 2021.
Proposed Approach: DiPETrans Framework

- We proposed a *DiPETrans* framework\(^6\) for parallel execution of the transactions at miners and validators, based on transaction shards identified using static analysis.

- We implement this technique using a distributed *leader–follower* approach within a mining community of servers.

\(^6\)Accepted at the journal of *Concurrency and Computation: Practice and Experience (CCPE)*, Wiley, 2021.
Proposed Approach: DiPETrans Framework

- We proposed a *DiPETrans* framework\(^6\) for parallel execution of the transactions at miners and validators, based on transaction shards identified using static analysis.

- We implement this technique using a distributed *leader–follower* approach within a mining community of servers.

- The leader shards the transactions in the block and the followers concurrently execute (mining) or verify (validation) them.

\(^6\) *Accepted at the journal of “Concurrency and Computation: Practice and Experience (CCPE),”* Wiley, 2021.
Proposed Approach: DiPETrans Framework

- We proposed a *DiPETrans* framework\(^6\) for parallel execution of the transactions at miners and validators, based on transaction shards identified using static analysis.

- We implement this technique using a distributed *leader–follower* approach within a mining community of servers.

- The leader shards the transactions in the block and the followers concurrently execute (mining) or verify (validation) them.

- When mining, the PoW is also partitioned and solved in parallel by the members of the community.

---

\(^6\)* Accepted at the journal of *Concurrency and Computation: Practice and Experience (CCPE)*, Wiley, 2021.
Proposed Approach: Sharding of Block Transactions

- DiPETrans groups the block transactions into independent shards and executes them parallelly in a distributed fashion using a leader-follower approach.

**Figure 3:** Sharding of transactions in a block using static graph analysis
DiPETrans Architecture: Miner Community

(a) Community Acting as Miner

1. Sequence During Proposing a Block
2. Leader Node
3. Block
4. Shards
5. Shard Tx Execution
6. Chain/Global State
7. Nonce
8. Pending Transaction Queue

- MI: Miner ID
- TS: Timestamp
- D: Difficulty
- BH: Block Hash
- PH: Previous Hash
- FS: Final State
- LS: Local State
- O: Other Information
- S: Shard
- m: # Shards
- k: # Transactions
- p: # Nonce Set
- g: Genesis Block
- n: # Followers at Miner
- v: # Followers at Validator

Parwat Singh Anjana
DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain
DiPETrans Architecture: Validator Community

Sequence During Validating a Block

(b) Community Acting as Validator

Chain/Global State

Mi: Miner ID
TS: Timestamp
D: Difficulty
BH: Block Hash
PH: Previous Hash
FS: Final State
LS: Local State
O: Other Information
S: Shard
m: # Shards
k: # Transactions
p: # Nonce Set
g: Genesis Block
v: # Followers at Validator
n: # Followers at Miner

Parwat Singh Anjana

DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain
DiPETrans: Theoretical Running Time Complexity

- Analyze() takes $O(n)$ to build transaction graph with $n$ edges and between $2 - 2n$ vertices. So, static analysis using WCC takes $O(n)$.

- With $m$ shards and $f$ follower nodes in the community, the LoadBalance() takes $O(m \cdot \log(m))$ to sort the shards.

- Using a priority queue to load balance shards (transactions) assigned to each follower, we get a time complexity of $O(m \cdot \log(f))$.

- For the LoadBalance phase, the combined time complexity is $O(m \cdot (\log(m) + \log(f)))$.

- So overall time complexity of $O(n + m \cdot (\log(m) + \log(f)))$. Usually, with $m > f$, expected complexity is $O(n + m \cdot \log(m))$.

- The worst-case time complexity for transaction execution is $O(n \cdot t_x)$ and the best-case time complexity is $\Omega(n f \cdot t_x)$, where, $t_x$ is a transaction execution time.

7 The time to complete the transaction execution is limited by the follower with the most number of transactions.
DiPETrans: Theoretical Running Time Complexity

- Analyze() takes $O(n)$ to build transaction graph with $n$ edges and between $2 - 2n$ vertices. So, static analysis using WCC takes $O(n)$.

- With $m$ shards and $f$ follower nodes in the community. The LoadBalance() takes $O(m \cdot \log(m))$ to sort the shards.

7 The time to complete the transaction execution is limited by the follower with the most number of transactions.
DiPETrans: Theoretical Running Time Complexity

- Analyze() takes $\mathcal{O}(n)$ to build transaction graph with $n$ edges and between $2 - 2n$ vertices. So, static analysis using WCC takes $\mathcal{O}(n)$.

- With $m$ shards and $f$ follower nodes in the community. The LoadBalance() takes $\mathcal{O}(m \cdot \log(m))$ to sort the shards.

- Using a priority queue to load balance shards (transactions) assigned to each follower, we get a time complexity of $\mathcal{O}(m \cdot \log(f))$.

The time to complete the transaction execution is limited by the follower with the most number of transactions.
DiPETrans: Theoretical Running Time Complexity

- Analyze() takes $O(n)$ to build transaction graph with $n$ edges and between $2 - 2n$ vertices. So, static analysis using WCC takes $O(n)$.
- With $m$ shards and $f$ follower nodes in the community. The LoadBalance() takes $O(m \cdot \log(m))$ to sort the shards.
- Using a priority queue to load balance shards (transactions) assigned to each follower, we get a time complexity of $O(m \cdot \log(f))$.
- For the LoadBalance phase the combined time complexity is $O(m \cdot (\log(m) + \log(f)))$.

7 The time to complete the transaction execution is limited by the follower with the most number of transactions.
DiPETrans: Theoretical Running Time Complexity

• Analyze() takes $O(n)$ to build transaction graph with $n$ edges and between 2 - $2n$ vertices. So, static analysis using WCC takes $O(n)$.

• With $m$ shards and $f$ follower nodes in the community. The LoadBalance() takes $O(m \cdot \log(m))$ to sort the shards.

• Using a priority queue to load balance shards (transactions) assigned to each follower, we get a time complexity of $O(m \cdot \log(f))$.

• For the LoadBalance phase the combined time complexity is $O(m \cdot (\log(m) + \log(f)))$.

• So overall time complexity of $O(n + m \cdot (\log(m) + \log(f)))$. Usually, with $m > f$, expected complexity is $O(n + m \cdot \log(m))$. 

\[7\text{ The time to complete the transaction execution is limited by the follower with the most number of transactions.}\]
DiPETrans: Theoretical Running Time Complexity

- Analyze() takes $O(n)$ to build transaction graph with $n$ edges and between $2 - 2n$ vertices. So, static analysis using WCC takes $O(n)$.
- With $m$ shards and $f$ follower nodes in the community. The LoadBalance() takes $O(m \cdot \log(m))$ to sort the shards.
- Using a priority queue to load balance shards (transactions) assigned to each follower, we get a time complexity of $O(m \cdot \log(f))$.
- For the LoadBalance phase the combined time complexity is $O(m \cdot (\log(m) + \log(f)))$.
- So overall time complexity of $O(n + m \cdot (\log(m) + \log(f)))$. Usually, with $m > f$, expected complexity is $O(n + m \cdot \log(m))$.
- The worst-case time complexity for transaction execution is $O(n \cdot t_x)$ and the best-case time complexity is $\Omega(\frac{n}{f} \cdot t_x)$, where, $t_x$ is a transaction execution time.\(^7\)

\(^7\) The time to complete the transaction execution is limited by the follower with the most number of transactions.
Outline

1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
• We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.
We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.

- We extracted \( \approx 80K \) blocks consisting of 5,170,597 total transactions.
We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.

- We extracted \( \approx 80K \) blocks consisting of 5,170,597 total transactions.
- There are two types of transactions: *monetary* and *smart contracts*. 
We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.

- We extracted \( \approx 80K \) blocks consisting of 5,170,597 total transactions.
- There are two types of transactions: monetary and smart contracts.

We used a commodity cluster to run the leader and followers.
• We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.
  • We extracted \( \approx 80K \) blocks consisting of 5,170,597 total transactions.
  • There are two types of transactions: monetary and smart contracts.
• We used a commodity cluster to run the leader and followers.
  • The implementation is in C++ using Apache thrift cross-platform micro-services library.
We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.

- We extracted \( \approx 80K \) blocks consisting of 5,170,597 total transactions.
- There are two types of transactions: monetary and smart contracts.

We used a commodity cluster to run the leader and followers.

- The implementation is in C++ using Apache thrift cross-platform micro-services library.
- Each node in the cluster has an 8-core AMD CPU with 32 GB memory, running CentOS, and connected using 1 Gbps Ethernet.
DiPETrans: Experimental Evaluation

- We empirically evaluated DiPETrans using 5 million actual transactions from the Ethereum blockchain.
  - We extracted \(\approx 80\text{K} \) blocks consisting of 5,170,597 total transactions.
  - There are two types of transactions: monetary and smart contracts.
- We used a commodity cluster to run the leader and followers.
  - The implementation is in C++ using Apache thrift cross-platform micro-services library.
  - Each node in the cluster has an 8-core AMD CPU with 32 GB memory, running CentOS, and connected using 1 Gbps Ethernet.
- Depending on the experiment configuration, a community has a leader running on one node and between 1 to 5 followers running on separate nodes.
Table 1: Summary of transactions in experiment workload

<table>
<thead>
<tr>
<th>Block type</th>
<th>$\rho$</th>
<th># Txns/ block</th>
<th># Blocks</th>
<th>$\sum # \text{Contract txns}$</th>
<th>$\sum # \text{Non-contract txns}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-1-1-100</td>
<td>$\frac{1}{1}$</td>
<td>100</td>
<td>3,880</td>
<td>193,959</td>
<td>194,000</td>
</tr>
<tr>
<td>data-1-1-200</td>
<td>200</td>
<td>1,940</td>
<td>193,959</td>
<td>194,000</td>
<td>194,000</td>
</tr>
<tr>
<td>data-1-1-300</td>
<td>300</td>
<td>1,294</td>
<td>193,959</td>
<td>194,000</td>
<td>194,100</td>
</tr>
<tr>
<td>data-1-1-400</td>
<td>400</td>
<td>970</td>
<td>193,959</td>
<td>194,000</td>
<td>194,000</td>
</tr>
<tr>
<td>data-1-1-500</td>
<td>500</td>
<td>776</td>
<td>193,959</td>
<td>194,000</td>
<td>194,000</td>
</tr>
<tr>
<td>data-1-2-100</td>
<td>$\frac{1}{2}$</td>
<td>100</td>
<td>5,705</td>
<td>193,959</td>
<td>376,530</td>
</tr>
<tr>
<td>data-1-2-200</td>
<td>200</td>
<td>2,895</td>
<td>193,959</td>
<td>385,035</td>
<td>385,035</td>
</tr>
<tr>
<td>data-1-2-300</td>
<td>300</td>
<td>1,940</td>
<td>193,959</td>
<td>388,000</td>
<td>388,000</td>
</tr>
<tr>
<td>data-1-2-400</td>
<td>400</td>
<td>1,448</td>
<td>193,959</td>
<td>385,168</td>
<td>385,168</td>
</tr>
<tr>
<td>data-1-2-500</td>
<td>500</td>
<td>1,162</td>
<td>193,959</td>
<td>386,946</td>
<td>386,946</td>
</tr>
<tr>
<td>data-1-4-100</td>
<td>$\frac{1}{4}$</td>
<td>100</td>
<td>9,698</td>
<td>193,959</td>
<td>775,840</td>
</tr>
<tr>
<td>data-1-4-200</td>
<td>200</td>
<td>4,849</td>
<td>193,959</td>
<td>775,840</td>
<td>775,840</td>
</tr>
<tr>
<td>data-1-4-300</td>
<td>300</td>
<td>3,233</td>
<td>193,959</td>
<td>775,840</td>
<td>775,840</td>
</tr>
<tr>
<td>data-1-4-400</td>
<td>400</td>
<td>2,425</td>
<td>193,959</td>
<td>776,000</td>
<td>776,000</td>
</tr>
<tr>
<td>data-1-4-500</td>
<td>500</td>
<td>1,940</td>
<td>193,959</td>
<td>776,000</td>
<td>776,000</td>
</tr>
<tr>
<td>data-1-8-100</td>
<td>$\frac{1}{8}$</td>
<td>100</td>
<td>16,164</td>
<td>193,959</td>
<td>1,422,432</td>
</tr>
<tr>
<td>data-1-8-200</td>
<td>200</td>
<td>8,434</td>
<td>193,959</td>
<td>1,492,818</td>
<td>1,492,818</td>
</tr>
<tr>
<td>data-1-8-300</td>
<td>300</td>
<td>5,705</td>
<td>193,959</td>
<td>1,517,530</td>
<td>1,517,530</td>
</tr>
<tr>
<td>data-1-8-400</td>
<td>400</td>
<td>4,311</td>
<td>193,959</td>
<td>1,530,405</td>
<td>1,530,405</td>
</tr>
<tr>
<td>data-1-8-500</td>
<td>500</td>
<td>3,464</td>
<td>193,959</td>
<td>1,538,016</td>
<td>1,538,016</td>
</tr>
<tr>
<td>data-1-16-100</td>
<td>$\frac{1}{16}$</td>
<td>100</td>
<td>32,327</td>
<td>193,959</td>
<td>3,038,738</td>
</tr>
<tr>
<td>data-1-16-200</td>
<td>200</td>
<td>16,164</td>
<td>193,959</td>
<td>3,038,832</td>
<td>3,038,832</td>
</tr>
<tr>
<td>data-1-16-300</td>
<td>300</td>
<td>10,776</td>
<td>193,959</td>
<td>3,038,832</td>
<td>3,038,832</td>
</tr>
<tr>
<td>data-1-16-400</td>
<td>400</td>
<td>8,082</td>
<td>193,959</td>
<td>3,038,832</td>
<td>3,038,832</td>
</tr>
<tr>
<td>data-1-16-500</td>
<td>500</td>
<td>6,466</td>
<td>193,959</td>
<td>3,039,020</td>
<td>3,039,020</td>
</tr>
</tbody>
</table>
Figure 4: Workload-1: speedup by community miner and validator over serial miner and validator.

- With 5 followers, the peak speedup achieved by the community miners’ is $2.18 \times$, the speedup efficiency is sub-optimal at about 51% for 4 followers and 44% for 5 followers, with 500 transactions/blocks.
- The default community validators’ average speedup is $1.25 \times$, and their peak is $2.03 \times$ with 5 followers and 500 transactions per block.
Figure 5: Workload-2: speedup by community miner and validator over serial miner and validator.

- For the community miners’ a peak speedup of $2.7\times$ is achieved with 5 followers and a favorable speedup efficiency of 73% with 3 followers is achieved when $\rho = \frac{1}{4}$.

- For the default community validators’ a peak speedup of $2.5\times$ is achieved with 5 followers.
DiPETrans Results: End-to-end Mining Speedup

Figure 6: Average end-to-end block creation speedup by community miner over serial miner.

- In Workload 1, a speedup of $1.15 \times$ to $4.91 \times$ for 1–5 followers that remain stable as the block size increases, with a speedup efficiency of 57.5 to 81.83%.

- We achieve a maximum speedup of $1.17 \times$ to $4.82 \times$ for 1–5 followers, with a speedup efficiency of 58.5 to 80.33% in Workload 2.
DiPETrans Results: Throughput

**Figure 7:** Throughput with varying transactions per block and varying $\rho$.

- In Workload 1, the maximum throughput is 1577 tps in a community with 5 followers at 500 transactions/block, which is $2.05 \times$ higher than that of serial execution.
- In Workload 2, we achieve a maximum throughput of 2147 tps that is $1.49 \times$ over serial when ratio $\rho = \frac{1}{16}$ for 5 followers, with 500 transactions/blocks. The sweet spot of maximum throughput is $2.52 \times$ with 1690 tps when $\rho = \frac{1}{4}$. 

Parwat Singh Anjana (CS17RESCH11004)
Guided by: Dr. Sathya Peri, Associate Professor DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain*
Figure 8: Transaction execution time by a follower and accumulative followers idle time on W1 and W2.

- The optimal community size depends on several parameters: \# transactions/block, \# shards formed, the mix of contractual and monetary transactions/shard.

- With an optimal community size, the idle time will be minimized, hence, the average execution time will be similar to the maximum execution time.
Outline

1. Introduction

2. Bottleneck in Existing Blockchain Design

3. Challenges in Executing Transactions Parallelly

4. Current Progress

5. Experimental Evaluation

6. Conclusion and Future Work
DiPETrans Conclusion

• We proposed DiPETrans framework to execute block transactions efficiently in parallel by leveraging distributed resources using leader-follower approach.
DiPETrans Conclusion

- We proposed DiPETrans framework to execute block transactions efficiently in parallel by leveraging distributed resources using leader-follower approach.

- The proposed techniques prevent transaction parallelization errors such as FBR, EMB, and FBin.
• We proposed DiPETrans framework to execute block transactions efficiently in parallel by leveraging distributed resources using leader-follower approach.

• The proposed techniques prevent transaction parallelization errors such as FBR, EMB, and FBin.

• We achieve a maximum speedup of $2.2 \times$ and $2.0 \times$ and an average speedup of $1.6 \times$ and $1.5 \times$ for the miner and the validator, respectively, with 100 to 500 transactions per block when using 6 machines in the community.
Future Work

• Exploring the possibilities of integrating our ideas into existing order-execute-based blockchain platforms like Bitcoin, Sawtooth, Tezos, and EOS is an exciting direction to pursue.
Future Work

• Exploring the possibilities of integrating our ideas into existing order-execute-based blockchain platforms like Bitcoin, Sawtooth, Tezos, and EOS is an exciting direction to pursue.

• We plan to integrate it with Ethereum blockchain by deploying a DiPETrans community smart contract.
Future Work

- Exploring the possibilities of integrating our ideas into existing order-execute-based blockchain platforms like Bitcoin, Sawtooth, Tezos, and EOS is an exciting direction to pursue.

- We plan to integrate it with Ethereum blockchain by deploying a DiPETrans community smart contract.

- Another interesting direction is to apply concurrency in the nested execution of SCTs.
Thanks!
Publications (1/2)

**Journal Papers:**

**Conference Papers:**

**Short Papers:**
Manuscripts under review/preparation:


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 9:** Clients send Transaction T1, T2 and T3 to Miner (Peer4)
**Introduction: Ethereum High Level Design**

**Figure 10:** Miner forms a block $B_4$ and computes final state ($FS$) sequentially
Figure 11: Miner broadcasts the block B4
Figure 12: Validators (Peer 1, 2, and 3) compute current state (CS) sequentially
Figure 13: Validators verify the FS and reach the consensus protocol
Figure 14: Block B4 successfully added to the blockchain
We have used two protocols implemented in IITH-STM library for concurrent execution of the smart contracts by miner.

Basic Time-stamp Ordering (BTO) Protocol

If $p_i(x)$ and $q_j(x)$, $i \neq j$, are operations in conflict, the following has to hold:

- $p_i(x)$ is executed before $q_j(x)$ iff $ts(t_i) < ts(t_j)$.

![Figure 15: BTO](image)

Figure 15: BTO

---

Multi-Version Time-stamp Ordering (MVTO) Protocol

- MVTO maintains multiple versions corresponding to each shared data-objects.
- It reduces the number of aborts and improves the throughput.

Figure 16: BTO

Figure 17: MVTO

---

Concurrent Validator: Fork-Join Approach

**On completion of Task (AU) execution =>**

- AU.InCnt = -1;
- AU.OutEdge.InCnt -= 1;
- ExeCount++;

**Status**
- -1 => Join
- 0 => Available
- 1 => Execute Task

**Figure 18: Fork-Join Approach**
Concurrent Validator: Decentralized Approach

List of Atomic Units (AUs)

On completion of AU Execution =>
AU.InCnt = -1;
AU.OutEdge.InCnt -= 1;
ExeCount++;

Threads keep on traversing the conflict graph to find AU nodes with InCount 0; If find a node then claim node and execute corresponding AU.

Threads join when all AUs executed (ExeCount == NumAUs)

Figure 19: Decentralized Approach
Proposed Methodologies: OptSmart

- Since static analysis fails to identify the conflicts precisely.
Proposed Methodologies: OptSmart

- Since static analysis fails to identify the conflicts precisely.
- We introduce *OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts* to exploit multi-processing on a multi-core system to improve throughput.
Proposed Methodologies: OptSmart

- Since static analysis fails to identify the conflicts precisely.
- We introduce *OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts* to exploit multi-processing on a multi-core system to improve throughput.
- Miners and validators use multiple threads to parallelly execute smart contract transactions (SCTs) in a block.
Proposed Methodologies: OptSmart

- Since static analysis fails to identify the conflicts precisely.
- We introduce *OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts* to exploit multi-processing on a multi-core system to improve throughput.
- Miners and validators use multiple threads to parallelly execute smart contract transactions (SCTs) in a block.
- A miner concurrently executes SCTs using optimistic read-write software transactional memory systems (RWSTMs) and saves the non-conflicting SCTs in the concurrent bin and conflicting SCTs in the block graph (BG).
Proposed Methodologies: OptSmart

- Since static analysis fails to identify the conflicts precisely.
- We introduce *OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts* to exploit multi-processing on a multi-core system to improve throughput.
- Miners and validators use multiple threads to parallelly execute smart contract transactions (SCTs) in a block.
- A miner concurrently executes SCTs using optimistic read-write software transactional memory systems (RWSTMs) and saves the non-conflicting SCTs in the concurrent bin and conflicting SCTs in the block graph (BG).
- Later, decentralized validators re-execute SCTs deterministically in parallel to validate the block by using information appended by the concurrent miner.
OptSmart Results

Figure 20: Speedup achieved by optimized concurrent miner and validator over serial miner and validator.

- OptSmart achieves an average speedup of $4.49 \times$ and $5.21 \times$ for optimized concurrent miners using BTO (Opt-BTO) and MVTO STM (Opt-MVTO) protocol than a serial miner.

- Optimized decentralized BTO and MVTO concurrent validator outperform average $7.68 \times$ and $8.60 \times$ than serial validator.

- The proposed efficient BG saves an average of $2.29 \times$ block space over existing approaches.
Read-Write STM (RWSTM) v/s Object-based STM (OSTM)

Figure 21: (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).
Proposed Methodology: ObjSC

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic Object-Based STMs (OSTMs).\textsuperscript{10}

---

Proposed Methodology: ObjSC

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic Object-Based STMs (OSTMs).\textsuperscript{10}
- STMs are convenient programming paradigms for a programmer to access shared memory using multiple threads.

Proposed Methodology: ObjSC

• We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.\textsuperscript{10}
• 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)*.

Proposed Methodology: ObjSC

• We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.\(^{10}\)

• 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.

---

Proposed Methodology: ObjSC

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.\(^\text{10}\)
- 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.

---

Proposed Methodology: ObjSC

- We develop an efficient framework for the concurrent execution of SCTs by miners using an optimistic *Object-Based STMs (OSTMs)*.\(^\text{10}\)
- 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.
- Hash Table based *OSTMs* export the following methods:
  - STM\_begin()
  - STM\_insert()
  - STM\_delete()
  - STM\_lookup()
  - STM\_tryC()
  - STM\_Abort()

Listing 1: Transfer function

```c
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  }
```
ObjSC: 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 }
```
ObjSC: Working of multi-threaded miner

1. Execute the SCTs concurrently using object semantic
2. Return the Conflict list
3. Create BG
4. Compute the hash of the previous block
5. Create a Block

(a). Multi-threaded Execution
(b). Balance Details
(c). Block Graph
(d). Compute Previous Hash
(e). Proposed Block

<table>
<thead>
<tr>
<th>Account</th>
<th>IS</th>
<th>FS_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$100</td>
<td>$80</td>
</tr>
<tr>
<td>B</td>
<td>$100</td>
<td>$110</td>
</tr>
<tr>
<td>C</td>
<td>$100</td>
<td>$110</td>
</tr>
<tr>
<td>D</td>
<td>$100</td>
<td>$50</td>
</tr>
<tr>
<td>E</td>
<td>$100</td>
<td>$150</td>
</tr>
</tbody>
</table>

Figure 22: Working of multi-threaded miner
• Miner maintains the BG in the form of the adjacency list, where vertices correspond only to committed SCTs.
- 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.
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 OSTM:

\[
\text{Conflicting Operations} = \begin{cases} 
STM\_lookup_i() & - & STM\_tryC_j() \\
STM\_delete_i() & - & STM\_tryC_j() \\
STM\_tryC_i() & - & STM\_tryC_j() \\
STM\_tryC_i() & - & STM\_delete_j() \\
STM\_tryC_i() & - & STM\_lookup_j()
\end{cases}
\]
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 OSTM

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

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

\[
\text{Conflicting Operations} = \begin{cases} 
STM\_lookup_i() & \rightarrow \ STM\_tryC_j() \\
STM\_delete_i() & \rightarrow \ STM\_tryC_j() \\
STM\_tryC_i() & \rightarrow \ STM\_tryC_j() \\
STM\_tryC_i() & \rightarrow \ STM\_delete_j() \\
STM\_tryC_i() & \rightarrow \ STM\_lookup_j()
\end{cases}
\]  

- Multi-threaded miner uses addVert() and addEdge() methods of BG.
- Later, validators re-execute the same SCTs concurrently and deterministically relying on the BG.
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 = \[
\begin{align*}
STM\_lookup_i() & \rightarrow STM\_tryC_j() \\
STM\_delete_i() & \rightarrow STM\_tryC_j() \\
STM\_tryC_i() & \rightarrow STM\_tryC_j() \quad (1) \\
STM\_tryC_i() & \rightarrow STM\_delete_j() \\
STM\_tryC_i() & \rightarrow STM\_lookup_j()
\end{align*}
\]

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.
• **SMV** uses `searchGlobal()` and `decInCount()` methods of BG.

---

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

![Block Graph Diagram]

**Figure 23:** Data structure of BG

---

**ObjSC: Block Graph (2/2)**

- **SMV** uses `searchGlobal()` and `declInCount()` methods of BG.

![Diagram of Block Graph](image)

(a). Underlying Representation of Block Graph

(b). Block Graph

**Figure 23**: Data structure of BG

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

---

ObjSC: Block Graph (2/2)

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

![Figure 23: Data structure of BG](image)

- OSTM\textsuperscript{11} 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.

ObjSC: Data Structure of SVOSTM to Maintain Conflicts

(a) Structure of Shared data-item

(b) Timeline View

(c) Transactions Conflict List

Figure 24: Underlying Data Structure of SVOSTM
ObjSC: 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)*.
ObjSC: 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 25:** (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).
ObjSC: Multi-Version OSTM based Miner

- Multi-Version OSTM (MVOSTM)\textsuperscript{12} maintain multiple versions for each shared data item and provide greater concurrency relative to Single-Version OSTM (SVOSTM).

ObjSC: Multi-Version OSTM based Miner

- *Multi-Version OSTMs (MVOSTMs)*\(^{12}\) 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.

ObjSC: Data Structure of MVOSTM to Maintain Conflicts

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

(b) Timeline View

(c) Transactions Conflict List

Figure 26: Underlying Data Structure of SVOSTM
ObjSC Correctness Criteria: Opacity

Figure 27: History H is not Opaque

Figure 28: Opaque History H
Fig. 29: Working of multi-threaded validator

1. Execute the SCTs concurrently using BG
2. Return the final state $FS_v$
3. Compare $FS_m$ given by miner and $FS_v$ computed by itself
4. If ($FS_m = FS_v$) then Block is Valid, else Invalid Block

(a). Block Graph
(b). Multi-threaded Execution
(c). Balance Details
(d). Compare the Final States

<table>
<thead>
<tr>
<th>Account</th>
<th>$IS$</th>
<th>$FS_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$100$</td>
<td>$80$</td>
</tr>
<tr>
<td>B</td>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td>C</td>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td>D</td>
<td>$100$</td>
<td>$50$</td>
</tr>
<tr>
<td>E</td>
<td>$100$</td>
<td>$150$</td>
</tr>
</tbody>
</table>
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.
ObjSC: Smart Multi-threaded Validator (SMV)

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.**
ObjSC: Smart Multi-threaded Validator (SMV)

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 identify 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.
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.IUC) then
        Atomically increment the global lookup counter, k.gLC;
        Increment k.IUC by 1; // Maintain k.IUC 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.ILC) && (k.gUC == k.IUC)) then
        Atomically increment the global update counter, k.gUC;
        Increment k.IUC by 1; // Maintain k.IUC in transaction local log.
        Insert k in shared memory with value v;
      end
      else
        return ⟨Miner is malicious⟩;
      end
    end
  end
Atomically decrements the k.gLC and k.gUC corresponding to each shared data-item key k;
// 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.lLCi) && (k.gUC == k.lUCi)) then
                Atomically increment the global update counter, k.gUC;
                Increment k.lUCi by 1; //Maintain k.lUCi in transaction local.
                Delete k in shared memory;
            end
        else
            return ⟨Miner is malicious⟩;
        end
    end
end
Atomically decrements the k.gLC and k.gUC corresponding to each shared data-item key k;
In Ethereum blockchain, smart contracts are written in Solidity language, which runs on Ethereum Virtual Machine (EVM).
• In Ethereum blockchain, smart contracts are written in Solidity language, which runs on Ethereum Virtual Machine (EVM).

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

EVM does not support multi-threading.

We converted smart contracts from Solidity to C++ language for multi-threaded execution.
We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.

<table>
<thead>
<tr>
<th>Workload</th>
<th>SCTs</th>
<th>Threads</th>
<th>Shared data items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload 1</td>
<td>50 - 300</td>
<td>50</td>
<td>500</td>
</tr>
<tr>
<td>Workload 2</td>
<td>100</td>
<td>10 - 60</td>
<td>500</td>
</tr>
</tbody>
</table>
We consider four benchmark contracts Coin, Ballot, Simple Auction, and Mix from Solidity documentation.

1. **Coin**: A simple cryptocurrency contract.
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.
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.
• 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 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 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:

<table>
<thead>
<tr>
<th>Workload</th>
<th>SCTs</th>
<th>Threads</th>
<th>Shared data items</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>50 - 300</td>
<td>50</td>
<td>500</td>
</tr>
<tr>
<td>W2</td>
<td>100</td>
<td>10 - 60</td>
<td>500</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Workload</th>
<th>SCTs</th>
<th>Threads</th>
<th>Shared data items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload 1 (W1)</td>
<td>50 - 300</td>
<td>50</td>
<td>500</td>
</tr>
<tr>
<td>Workload 2 (W2)</td>
<td>100</td>
<td>10 - 60</td>
<td>500</td>
</tr>
</tbody>
</table>
• **MVOSTM, SVOSTM, MVTO, BTO, Speculative Bin, and Static Bin miner** provide an average speedup of $3.91 \times$, $3.41 \times$, $1.98 \times$, $1.5 \times$, $3.02 \times$, and $1.12 \times$, over Serial miner, respectively.
### Table 2: Overall average speedup on all workloads by multi-threaded miner over serial miner

<table>
<thead>
<tr>
<th>Contract</th>
<th>BTO Miner</th>
<th>MVTO Miner</th>
<th>SVOSTM Miner</th>
<th>MVOSTM Miner</th>
<th>StaticBin Miner</th>
<th>SpecBin Miner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coin</td>
<td>1.596</td>
<td>1.959</td>
<td>4.391</td>
<td>5.572</td>
<td>1.279</td>
<td>6.689</td>
</tr>
<tr>
<td>Ballot</td>
<td>0.960</td>
<td>1.065</td>
<td>2.229</td>
<td>2.431</td>
<td>1.175</td>
<td>2.233</td>
</tr>
<tr>
<td>Auction</td>
<td>2.305</td>
<td>2.675</td>
<td>3.456</td>
<td>3.881</td>
<td>1.524</td>
<td>2.232</td>
</tr>
<tr>
<td>Mix</td>
<td>1.596</td>
<td>2.118</td>
<td>3.425</td>
<td>3.898</td>
<td>1.102</td>
<td>3.080</td>
</tr>
<tr>
<td>Total Avg. Speedup</td>
<td><strong>1.61</strong></td>
<td><strong>1.95</strong></td>
<td><strong>3.38</strong></td>
<td><strong>3.95</strong></td>
<td><strong>1.27</strong></td>
<td><strong>3.56</strong></td>
</tr>
</tbody>
</table>
ObjSC Results: SMV Speedup

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

![Graph showing SMV Speedup over Serial validator](image)

**Figure 31**: Speedup of SMV over Serial validator
**Table 3:** Overall average speedup on all workloads by SMV over serial validator

<table>
<thead>
<tr>
<th>Contract</th>
<th>BTO SMV</th>
<th>MVTO SMV</th>
<th>SVOSTM SMV</th>
<th>MVOSTM SMV</th>
<th>StaticBin SMV</th>
<th>SpecBin SMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coin</td>
<td>26.576</td>
<td>28.635</td>
<td>30.344</td>
<td>32.864</td>
<td>5.296</td>
<td>7.565</td>
</tr>
<tr>
<td>Ballot</td>
<td>26.037</td>
<td>28.333</td>
<td>33.695</td>
<td>36.698</td>
<td>3.570</td>
<td>3.780</td>
</tr>
<tr>
<td>Auction</td>
<td>27.772</td>
<td>31.781</td>
<td>29.803</td>
<td>32.709</td>
<td>4.694</td>
<td>5.214</td>
</tr>
<tr>
<td>Mix</td>
<td>36.279</td>
<td>39.304</td>
<td>42.139</td>
<td>45.332</td>
<td>4.279</td>
<td>4.463</td>
</tr>
<tr>
<td><strong>Total Avg. Speedup</strong></td>
<td><strong>29.17</strong></td>
<td><strong>32.01</strong></td>
<td><strong>34.00</strong></td>
<td><strong>36.90</strong></td>
<td><strong>4.46</strong></td>
<td><strong>5.26</strong></td>
</tr>
</tbody>
</table>
Acceptance of even a single malicious block result in the blockchain going into inconsistent state.
Figure 33: Average number of dependencies in BG for mix contract on W1 and W2
**ObjSC Results: BG Depth**

![Graph showing speedup of SMV over serial and depth of BG for W3.](image)

**Figure 34:** Speedup of SMV over serial and depth of BG for W3
• We developed an efficient framework for concurrent execution of SCTs by a multi-threaded miner using two protocols, SVOSTM and MVOSTM of optimistic STMs\textsuperscript{13}.

\textsuperscript{13}Technical report: https://arxiv.org/abs/1904.00358
We developed an efficient framework for concurrent execution of SCTs by a multi-threaded miner using two protocols, SVOSTM and MVOSTM of optimistic STMs\textsuperscript{13}.

To avoid FBR errors, the multi-threaded miner captures the dependencies among SCTs in the form of a BG.

\textsuperscript{13}Technical report: https://arxiv.org/abs/1904.00358
ObjSC 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\textsuperscript{13}.

- To avoid FBR errors, the multi-threaded miner captures the dependencies among SCTs in the form of a BG.

- To handle EMB error, we proposed SMV that re-executes SCTs concurrently relying on the BG provided by the miner.

\textsuperscript{13} Technical report: https://arxiv.org/abs/1904.00358
Objective 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.\textsuperscript{13}

- To avoid FBR errors, the multi-threaded miner captures the dependencies among SCTs in the form of a BG.

- To handle EMB error, we proposed SMV that re-executes SCTs concurrently relying on the BG provided by the miner.

- The proposed approach achieves significant performance gain over the state-of-the-art SCTs execution framework.

\textsuperscript{13} Technical report: https://arxiv.org/abs/1904.00358
ObjSC Future Work

- 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.
ObjSC Future Work

- 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.
ObjSC Future Work

- 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.
ObjSC Future Work

- 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.
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.

Another interesting direction is to apply concurrency in the nested execution of SCTs.
Collaborators

Parwat Singh Anjana
Ph.D. Student
IIT Hyderabad, India
cs17resch11004@iith.ac.in

Hagit Attiya
Professor
Technion, Israel
hagit@cs.technion.ac.il

Sweta Kumari
Postdoc Fellow
Technion, Israel
sweta@cs.technion.ac.il

Sathya Peri
Associate Professor
IIT Hyderabad, India
sathyap@cse.iith.ac.in

Yogesh Simmhan
Professor
IISc, Bangalore, India
simmhan@iisc.ac.in

Archit Somani
Postdoc Fellow
Technion, Israel
archit@cs.technion.ac.il

Shrey Baheti
Software Engineer
Cargill Digital Labs, India
shrey_baheti@cargill.com

Sachin Rathor
Software Engineer
Microsoft, India
cs18mtech01002@iith.ac.in

Thanks!