CS5280: Project Report
Concurrent Execution of Blockchain Smart Contract transactions (SCTs)

Darpan Gaur CO21BTECH11004 **Yoshita Kondapalli** CO21BTECH11008

1 Problem Statement

Sequential execution of smart contract transactions (SCTs) in blockchain systems restricts concurrency and limits performance. There is a need for a framework that facilitates concurrent execution of SCTs, enhancing performance by allowing non-conflicting transactions to be processed simultaneously while preserving deterministic order.

2 DAG

In blockchain systems, particularly those handling smart contracts, executing transactions sequentially within a block limits throughput and scalability. To address this, we employ a DAG-based (Directed Acyclic Graph) parallel execution framework. The DAG encodes dependencies between transactions based on their read and write sets, ensuring that only independent transactions are executed concurrently. This approach allows for a significant improvement in execution efficiency.

The DAG represents transactions as nodes and dependencies (conflicts) as directed edges. A transaction T_i has an edge to T_j if T_j accesses data that T_i modifies or vice versa. Three types of conflicts are considered:

- Read-Write conflict: T_i writes to an address that T_j reads.
- Write-Read conflict: T_i reads from an address that T_j writes to.
- Write-Write conflict: Both T_i and T_j write to the same address.

Only transactions with zero incoming edges (i.e., no unresolved dependencies) can be safely executed in parallel. After executing a transaction, its outgoing edges are removed (i.e., dependent transactions' in-degree is decremented), allowing other transactions to become eligible for execution.

3 Design of DAG

Our implementation consists of two main phases: DAG construction and parallel transaction execution. The design is modular, thread-safe, and optimized for performance.

3.1 DAG Construction

The DAG is constructed using multiple threads to parse a list of transactions read from an input file. Each transaction consists of a read set and a write set. The DAG is represented using an adjacency list and an atomic array for in-degree counts. The construction process proceeds as follows:

1. Each thread picks the next transaction index to process using an atomic counter.
2. For each transaction T_i, all subsequent transactions T_j are checked for conflicts:
   - If any address in T_i's write set intersects with T_j's read or write set, or
   - If any address in T_i's read set intersects with T_j's write set,

   then an edge is added from T_i to T_j, and the in-degree of T_j is incremented.

3.2 Parallel Execution

Transaction execution is also handled in parallel by multiple threads. Each thread selects a transaction with zero in-degree (i.e., no pending dependencies) using a two-pass scan over the DAG. The ‘compare_exchange‘ operation ensures atomicity and prevents duplicate execution. Once a transaction is selected:

1. It is executed (simulated in our implementation),
2. Its successors' in-degrees are decremented,
3. The process repeats until all transactions are executed.

3.3 Implementation Details

Our DAG and scheduler are implemented in C++ for concurrency efficiency. Key implementation details include:

- A vector of atomic integers for in-degrees.
- An adjacency list using vectors.
- Use of mutexes to control file output access across threads.
- Thread-safe DAG construction and transaction execution using atomic counters and synchronization.

3.4 Benefits

This design aligns with the framework proposed in *Piduguralla et al. (2023)* and yields significant improvements in throughput compared to serial and tree-based execution models. The adjacency matrix version used here is more performant due to direct access and reduced pointer traversal overhead.

4 Block Pilot

In modern blockchain systems, transaction throughput remains limited by serial execution, especially when validators and proposers operate under different execution contexts. **BlockPilot** addresses this issue by enabling parallel execution of smart contract transactions (SCTs) across both proposer and validator roles. It introduces an *Optimistic Concurrency Control with Write Snapshot Isolation (OCC-WSI)* algorithm for proposers, and a *pipeline scheduling framework* for validators.

In our implementation inspired by BlockPilot, we adopt a proposer-style OCC-WSI mechanism, enabling each thread to execute transactions based on a consistent snapshot and verify for conflicts post-execution. A transaction is committed if no conflicting version updates occurred since its snapshot; otherwise, it is aborted and retried.

Validators execute the committed transactions from the proposed block using a dependency-based parallel scheduler. The scheduler builds a conflict graph and assigns transactions with no interdependencies to different threads, executing subgraphs concurrently while preserving the serial order within each subgraph.

This design enables efficient, parallel execution of SCTs while preserving deterministic execution outcomes, essential for consensus and fault tolerance in blockchain.

5 Correctness of Snapshot Isolation

6 Design of Block Pilot

Our implementation draws directly from the BlockPilot architecture, employing an *OCC-WSI concurrency protocol* and a *thread-based parallel scheduler*. The execution design is comprised of the following core components:

Proposer-Side: OCC-WSI Execution

The proposer initializes a reserve table to track versioning for each data item and assigns each transaction a `snapshot version`. The algorithm proceeds as follows:

1. Each thread selects an available transaction using atomic operations.
2. The transaction reads from a consistent snapshot of the data state.
3. Execution is performed in a local memory buffer (read set and write set).

4. A conflict detection step checks if any data read has been modified since the snapshot version.
5. If validation passes, the transaction's write set is committed, and the reserve table is updated.
6. If validation fails, the transaction is aborted and may be retried.

This behavior mirrors Algorithm 1 in BlockPilot, and the core logic is implemented in our `detectConflict()` method in `scheduler.cpp`.

Pseudo code for the OCC-WSI execution is provided in Listing 1.

Listing 1: Pseudo code for OCC-WSI Execution

```
void threadFunc(int threadID) {
    while (completedTxn < totTrans) {
        int txnID = selectTxn();

        snapshotVersion = state;

        rs, ws = executeTx(txnID, snapshotVersion);

        conflictResult = detectConflict(txnID, rs, ws);
        if (conflictResult == -1) {
            // Transaction aborted
            continue;
        }
        // Transaction committed
        completedTxn++;
    }
}

int detectConflict(int txnID, set<int> rs, set<int> ws) {
    for (int rec: rs) {
        if (Table[rec] > snapshotVersion) {
            // Conflict detected
            return -1;
        }

    }
    state = snapshotVersion + 1;
    for (int rec: ws) {
        Table[rec] = state;
        // update the global memory
    }
    return 0;   // transaction committed
}
```

7 Transaction Generator

The purpose of this program is to generate a specified number of random transactions consisting of read/write operations on multiple accounts. It ensures that the generated transactions match a desired level of dependency (conflict) between them and saves them to a file.

6.1 Inputs

The program takes the following inputs from the user:

- **Target Dependency Percentage (0–100)**: Desired percentage of conflicting transaction pairs.
- **Number of Transactions ($\geq$ 2)**: Total number of transactions to generate.
- **Number of Accounts ($\geq$ 1)**: Number of accounts on which operations are performed.

6.2 Core Functionalities

`generate_transaction(num_accounts)`

Generates a single transaction consisting of 1 to 3 random operations. Each operation is randomly selected as a read (`r`) or write (`w`) on a randomly chosen account.

`generate_transactions(num_transactions, num_accounts)`

Generates a list of transactions by calling `generate_transaction` repeatedly.

`transaction_has_conflict(t1, t2)`

Determines whether two transactions conflict. A conflict exists if both access the same account and at least one operation is a write.

`calculate_dependency(transactions)`

Calculates the percentage of conflicting transaction pairs. It uses the formula:

$$\text{Dependency (\%)} = \frac{\text{Number of Conflicting Pairs}}{\binom{n}{2}} \times 100$$

where n is the number of transactions.

`save_transactions(transactions, filename)`

Saves the transactions to a text file in the format `r(1), w(2), r(3)`.

6.3 Execution Flow

Input Validation

The program prompts the user for valid numeric inputs that meet the following constraints:

- Target dependency percentage must be between 0 and 100.
- Number of transactions must be at least 2.
- Number of accounts must be at least 1.

Attempt Loop

The program makes up to 5 attempts to generate a set of transactions that meets the target dependency percentage within a tolerance of $\pm 1\%$. On a successful attempt:

- The transactions are saved to `transactions.txt`.
- Dependency percentage and attempt details are printed to the console.

If none of the attempts succeed, a failure message is displayed after all 5 tries.

Output

- **File Output:** `transactions.txt`, containing the list of generated transactions.
- **Console Output:** Displays success/failure messages, attempt number, and achieved dependency.

Example Output

For 5 transactions and 3 accounts, a possible output is:

```
r(1), w(3)
w(2), r(3)
r(1)
w(2), w(1)
r(2), w(3)
```

Dependency achieved: 70% (Target: 70%)

8 Experiments

8.1 Number of Transactions

In this experiment number of transactions are varied from 1000 to 5000 in increments of 1000. The number of threads is kept constant at 16 and dependency percentage is kept constant at 40%.

Figure 1 shows the execution time vs number of transactions and Figure 2 shows the number of aborts vs number of transactions. Figure 3 shows the DAG creation time vs number of transactions. For DAG time to execute the transactions is taken and reported in figure, DAG creation time is not included.

- As number of transactions increases, the execution time also increases.
- Block Pilot performs better than DAG in terms of execution time.
- Number of aborts also increases with the number of transactions.
- DAG creation time also increases with the number of transactions.

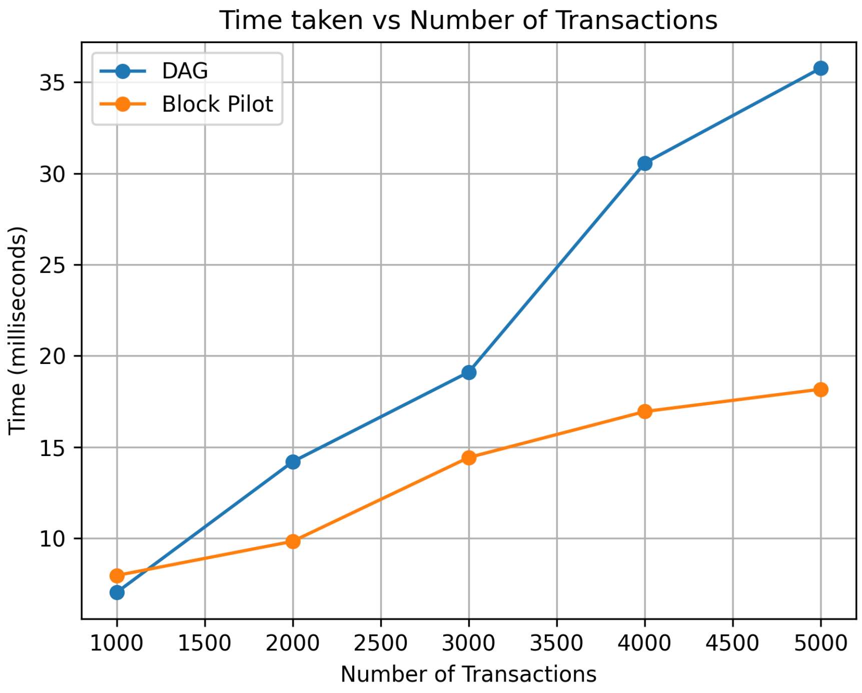

Figure 1: Execution Time vs Number of Transactions

8.2 Dependency Percentage

In this experiment dependency percentage is varied from 0% to 80% in increments of 20%. The number of transactions is kept constant at 3000 and number of threads is kept constant at 16.
Figure 4 shows the execution time vs dependency percentage and Figure 5 shows the number of aborts vs dependency percentage. Figure 6 shows the DAG creation time vs dependency percentage. For DAG time to execute the transactions is taken and reported in figure, DAG creation time is not included.

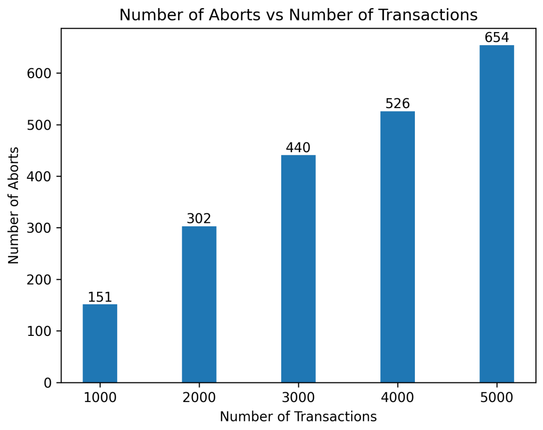

Figure 2: Number of Aborts vs Number of Transactions

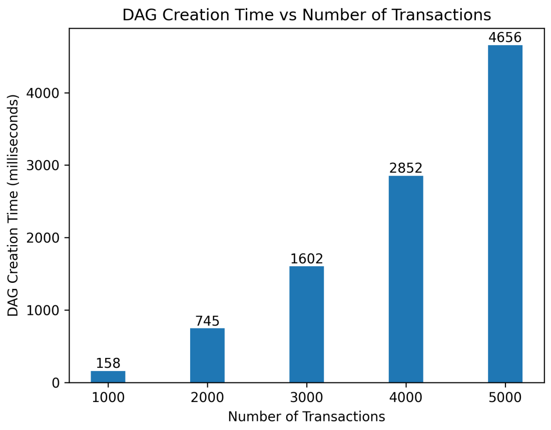

Figure 3: DAG Creation Time vs Number of Transactions

- As dependency percentage increases, the execution time also increases.
- Block Pilot performs better than DAG in terms of execution time.
- Number of aborts also increases with the dependency percentage.
- DAG creation time also increases with the dependency percentage, but the rate of increase decreases as the dependency percentage increases and having almost equal time for 60% and 80%.

8.3 Number of Threads

In this experiment, the number of threads is varied from 2 to 32 in increments of multiples of 2. The number of transactions is kept constant at 3000 and dependency percentage is kept constant at 40%. Figure 7 shows the execution time vs number of threads and Figure 8 shows the number of aborts vs number of threads. Figure 9 shows the DAG creation time vs number of threads. For DAG time to execute the transactions is taken and reported in figure, DAG creation time is not included.

- As number of threads increases, the execution first decreases and then increases.
- Block Pilot performs better than DAG in terms of execution time.
- For Block Pilot as number of threads increases, so number of aborts hence execution time starts increases after 8 threads.
- Number of aborts also increases with the number of threads.
- DAG creation time decreases with the number of threads, as more threads are available to process the transactions.

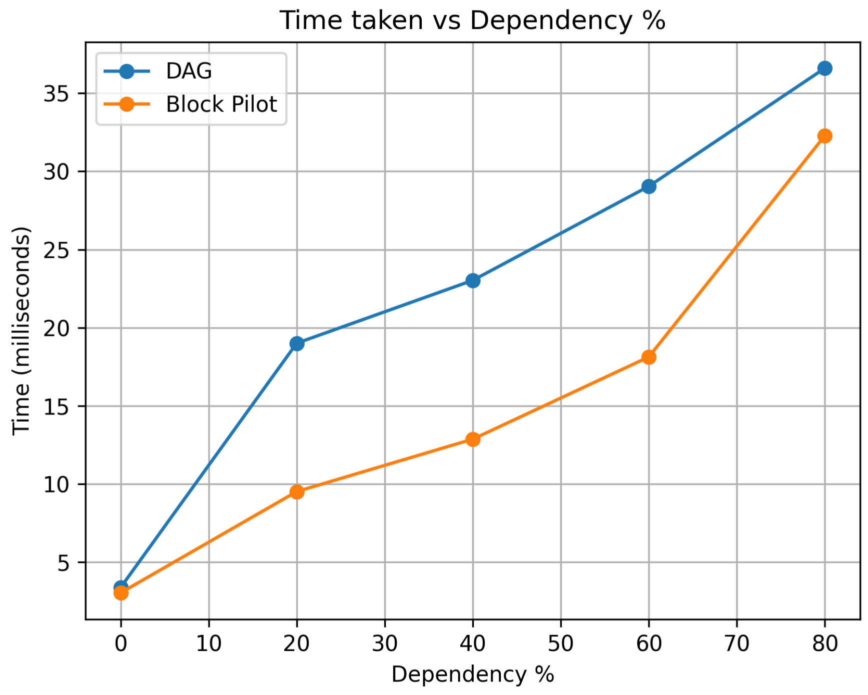

Figure 4: Execution Time vs Dependency Percentage

9 Observation and Conclusion

10 References

1. Haowen Zhang, Jing Li, He Zhao, Tong Zhou, Nianzu Sheng, and Hengyu Pan. *Block-Pilot: A Proposer-Validator Parallel Execution Framework for Blockchain.* In 52nd International Conference on Parallel Processing (ICPP), 2023. DOI: 10.1145/3605573.3605621.

2. Parwat Singh Anjana et al. *An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts.* In EuroMicro PDP, 2019. DOI: 10.1007/978-3-031-39698-4_13.

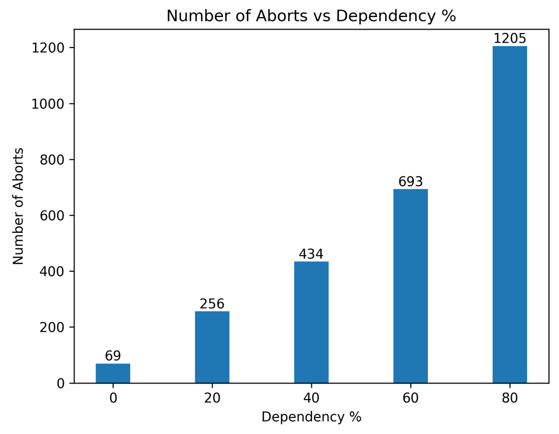

Figure 5: Number of Aborts vs Dependency Percentage

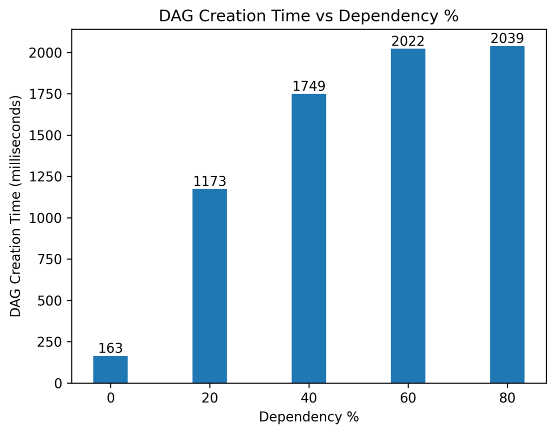

Figure 6: DAG Creation Time vs Dependency Percentage

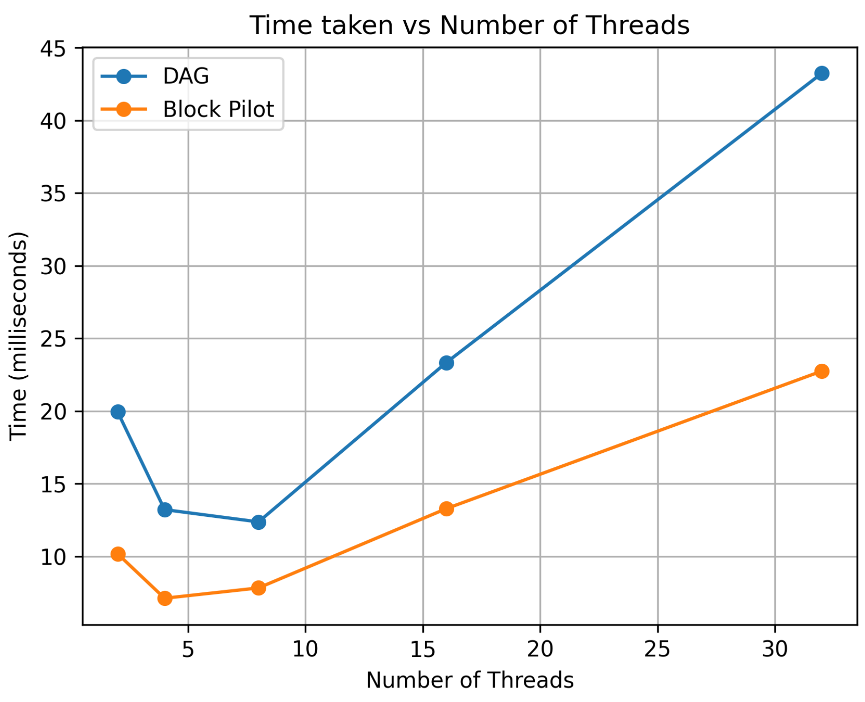

Figure 7: Execution Time vs Number of Threads

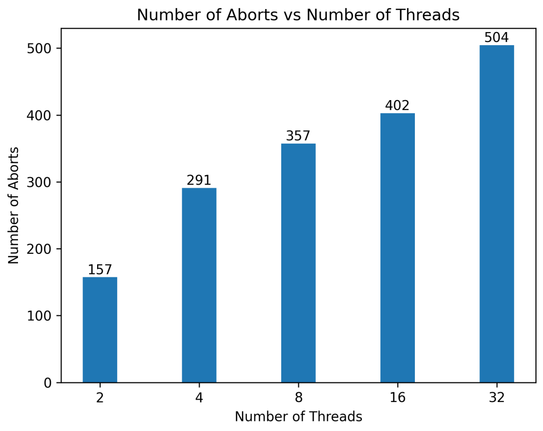

Figure 8: Number of Aborts vs Number of Threads

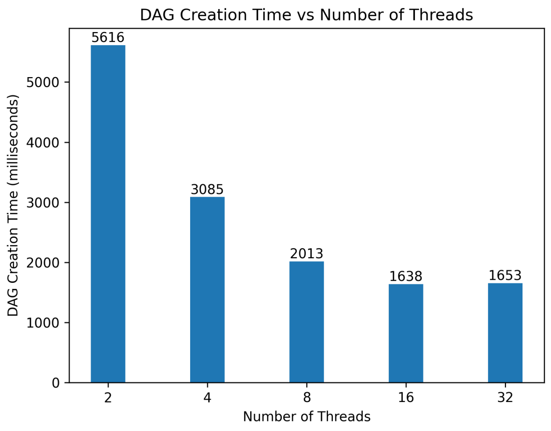

Figure 9: DAG Creation Time vs Number of Threads