Project: Concurrency Control in Transactional Systems
Serializable Snapshot Isolation(SSI)

PARUVADA YASASWINI (CS22BTECH11046)
MANE POOJA VINOD (CS22BTECH11035)

Serializable Snapshot Isolation

How It Works

- Reads from a consistent snapshot of the database, similar to SI.
- Introduces **SIREAD locks** — non-blocking locks that track which data items a transaction reads.
- Tracks **read-write (rw) antidependencies** between transactions to monitor conflicting operations.
- Builds a **dependency graph** and identifies **dangerous structures** (cycles) that can lead to non-serializable executions.
- If a dangerous structure is detected, SSI **aborts one of the involved transactions** to maintain serializability.

What is a SIREAD Lock?

A **SIREAD lock** is a special type of non-blocking read lock used in SSI. It allows:

- Tracking of read dependencies without blocking writers.
- Efficient detection of anomalies while preserving high concurrency.

Pseudocode for SSI (Serializable Snapshot Isolation)

```
Transaction T:
    snapshot = take_snapshot() // Same as SI: only committed data visible

    for each read operation:
        value = read(snapshot, key)
        acquire_SIREAD_lock(key) // Record read dependency

    for each write operation:
        buffer the update locally
        on write:
            check for conflicting SIREAD locks
            if (conflict detected) then
                record rw-antidependency (T -> T')

    on commit:
        if (T is part of a dangerous structure)
            abort T (to break cycle)
        else
            apply buffered writes
```

Listing 1: SSI Transaction Logic (Pseudocode)

SSI Benchmark Implementation (C++) – Overview

This C++ program implements the Serializable Snapshot Isolation (SSI) concurrency control mechanism with a focus on performance benchmarking. Below is a high-level explanation of its structure and functionality.

1. Core Concepts Implemented

- **MVCC (Multi-Version Concurrency Control):** Shared variables maintain a list of versions, enabling snapshot reads.
- **SSI (Serializable Snapshot Isolation):** Enhances SI by detecting dangerous rw-antidependency cycles using SIREAD locks.
- **SIREAD Locks:** Track read dependencies without blocking writers.
- **rw-antidependencies:** Read-Write conflicts tracked for each transaction.

2. Transaction Lifecycle

- `beginTransaction()`: Captures a snapshot version and marks the transaction as active.
- `read()`: Reads from the snapshot or local buffer and acquires a SIREAD lock.
- `write()`: Buffers writes, checks for conflicts, and tracks rw-antidependencies.
- `tryCommit()`: If the transaction is part of a dangerous structure (both in and out conflicts), it aborts; otherwise, it commits.

3. Concurrency Management

- `sharedMutex`: Synchronizes access to shared variables.
- `lockManagerMutex`: Manages SIREAD lock structures.
- `activeTransactions`: Tracks currently active transactions for dependency checks.
- `garbageCollectOldVersions()`: Cleans up old MVCC versions.
- `cleanupCommittedTxns()`: Cleans up committed transactions when safe.

4. Benchmark Execution

- `updtMem()`: Function executed by each thread. It runs a fixed number of transactions per thread, retrying up to 10 times on abort.
- `main()`: Initializes shared memory, spawns threads, and runs transactions. Aggregates and reports:
  - Average commit delay (in ms)
  - Abort rate (as a percentage)
  - Throughput (transactions per second)

5. Parameters Used

- `M`: Number of shared variables.
- `totTrans`: Total number of transactions.
- `numThreads`: Number of concurrent threads.
- `constVal, lambdaVal, numIters`: Constants for workload behavior.
- `readOnlyPercent`: Percent of transactions that are read-only.

Conclusion

This SSI implementation provides a simulated environment to evaluate the behavior of serializable isolation under various workloads. It effectively tracks read-write conflicts and manages concurrent access using snapshot isolation extended with dependency tracking.

Results:

In this project, we have compared three concurrency control algorithms: **Serializable Snapshot Isolation (SSI)**, **Multi-Version Timestamp Ordering (MVTO)**, and **Forward Optimistic Concurrency Control (FOCC)**. Our goal is to compare SSI performance with MVTO and FOCC in terms of average commit delay, abort rate, and throughput. **FOCC** is included in the comparison because it follows an optimistic approach like SSI and allows transactions to execute without locking during most of their execution, making it conceptually similar. However, unlike SSI, FOCC validates at commit time without tracking read dependencies, which can lead to different performance and abort characteristics.

MVTO is a widely studied multi-version concurrency control algorithm that provides a baseline for comparison. MVTO maintains multiple versions of data and uses timestamps to order transactions. This makes it suitable for evaluating how version management affects performance in contrast to SSI's dependency tracking approach.

1. Varying Number of Transactions

Input Format: `<n> <m> <constVal> <lambda> <readOnlyPercent> <totTrans> <numIters>` **Input:** `16 <n=1000> 1000 20 80 <totTrans=1000 to 5000> 20`

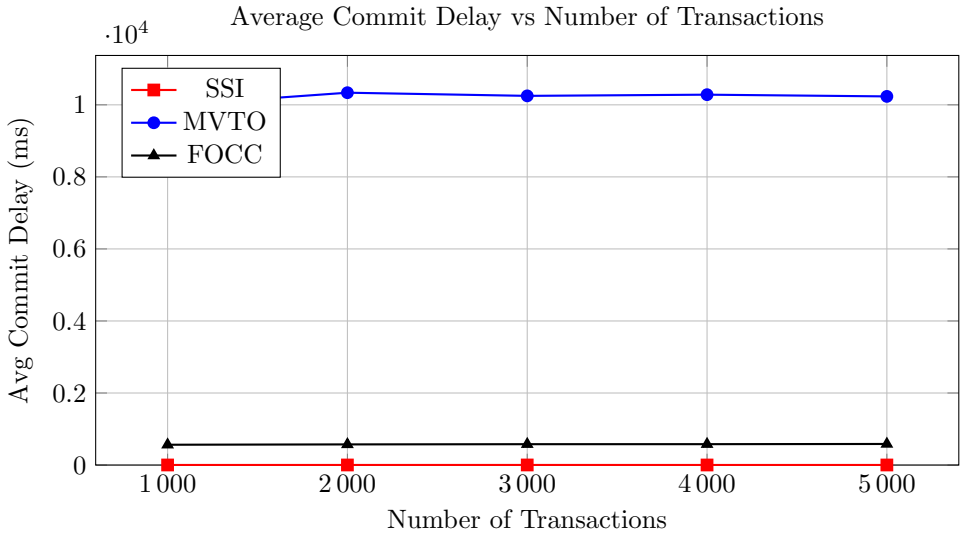

Figure 1: Commit delay rises steeply for MVTO, while SSI stays extremely low. FOCC is intermediate but increases slightly.

Observation: MVTO shows extremely high and nearly constant commit delay (10,000 ms), regardless of the number of transactions, due to version maintenance overhead. SSI consistently achieves the lowest delays (around 2 ms), making it highly efficient. FOCC falls in between, with moderate delays (570 ms) that slightly increase with more transactions.

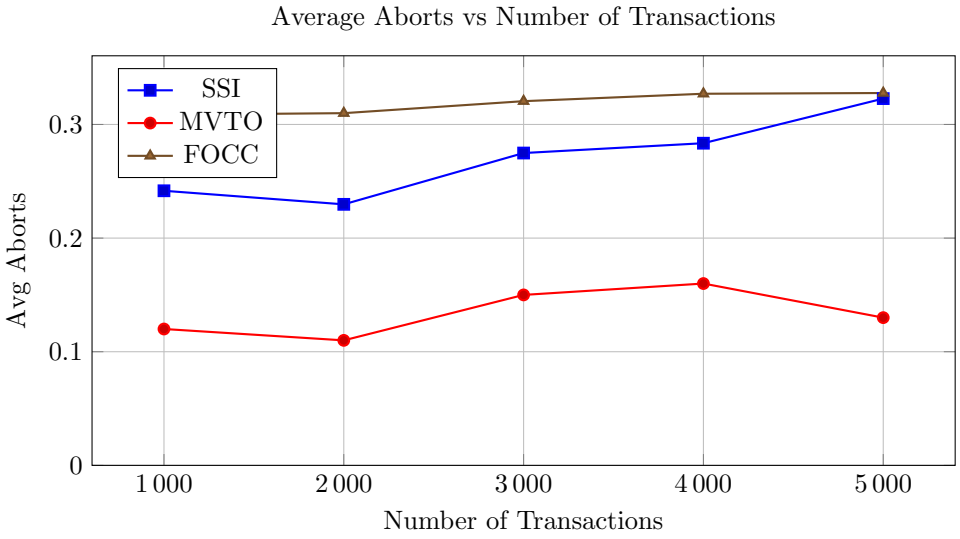

Figure 2: MVTO consistently has fewer aborts; SSI and FOCC show higher abort rates as transactions increase.

Figure 3: Performance vs Number of Transactions

Observation: MVTO has the lowest abort rate (0.1–0.16), showing stability under increased load. SSI's abort rate increases steadily (from 0.24 to 0.32) due to higher concurrency conflicts. FOCC exhibits a slightly higher and gradually increasing abort rate (0.31 to 0.33), indicating more contention as transaction volume grows.

2. Varying Number of Variables

Input: `16 <n=1000 to 5000> 1000 20 80 <totTrans=1000> 20`

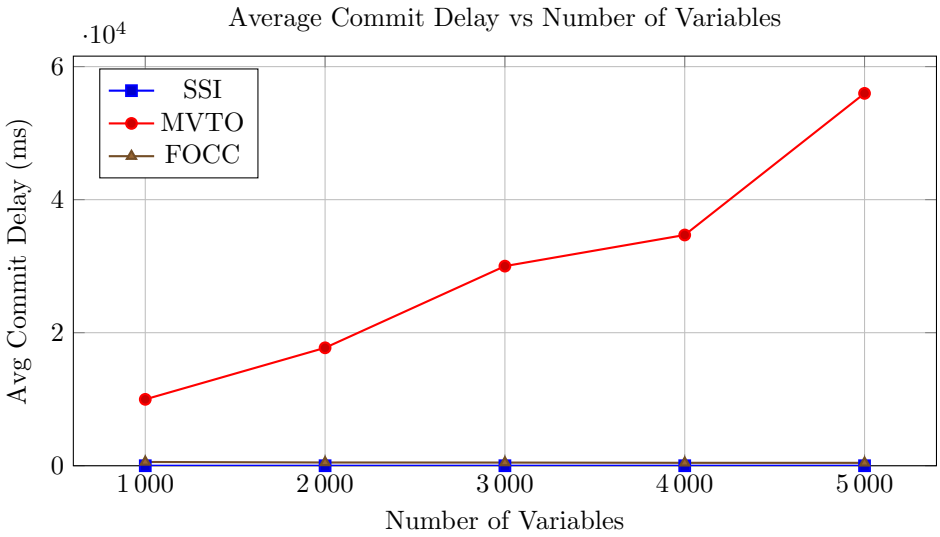

Figure 4: MVTO's delay grows rapidly with more variables. SSI and FOCC are stable and efficient.

Observation: MVTO's commit delay increases drastically (from 10,000 to 56,000 ms) as the number of variables grows, likely due to increased version management overhead. SSI and FOCC maintain low and

stable delays throughout, indicating better scalability with respect to variable count.

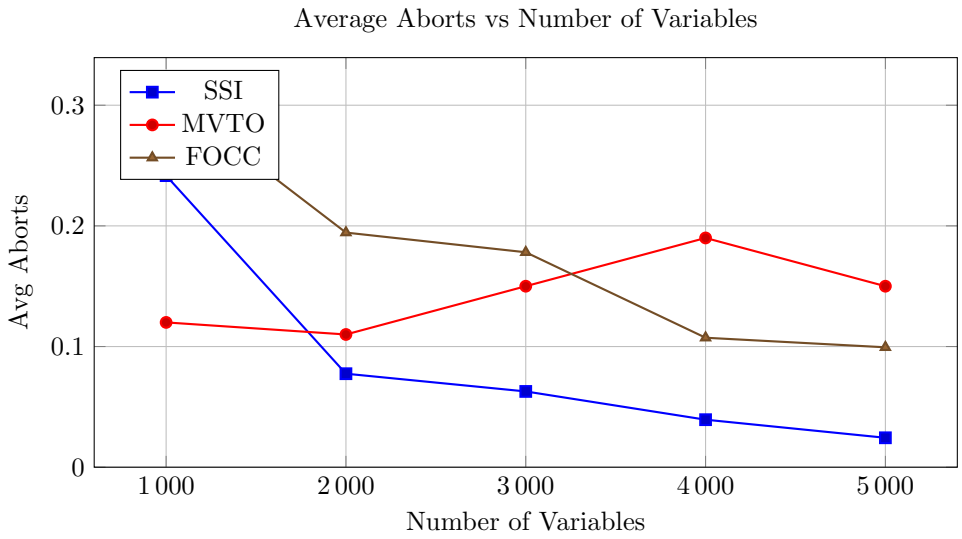

Figure 5: SSI shows fewer aborts as variables increase; MVTO and FOCC remain mostly steady. **Observation:** SSI's abort rate significantly decreases (from 0.24 to 0.2) with more variables, indicating fewer conflicts. FOCC also improves (from 0.31 to 0.1). MVTO stays relatively stable, reflecting low conflict sensitivity but poor efficiency.

Figure 6: Performance vs Number of Variables

3. Varying Number of Threads

Input: `<threads=2 to 32 (powers of 2)> 1000 1000 20 80 1000 20`

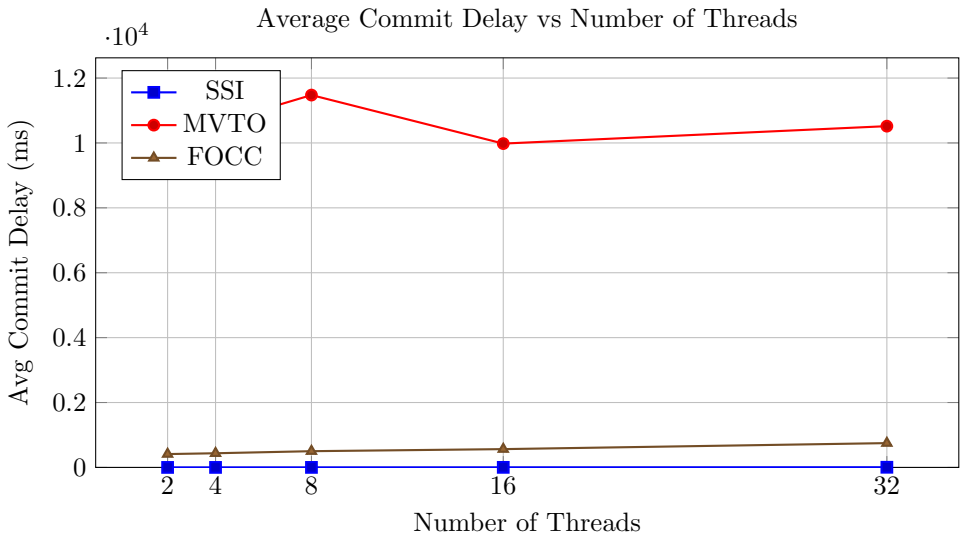

Figure 7: SSI remains extremely efficient even as threads increase. FOCC rises gradually. MVTO stays high throughout.

Observation: SSI has minimal delay at low thread counts (0.09 ms), but delay increases with concurrency (up to 4.85 ms). MVTO remains consistently high (10,000–11,000 ms), and FOCC delay grows from 410 to 750 ms, affected by increased validation overhead.

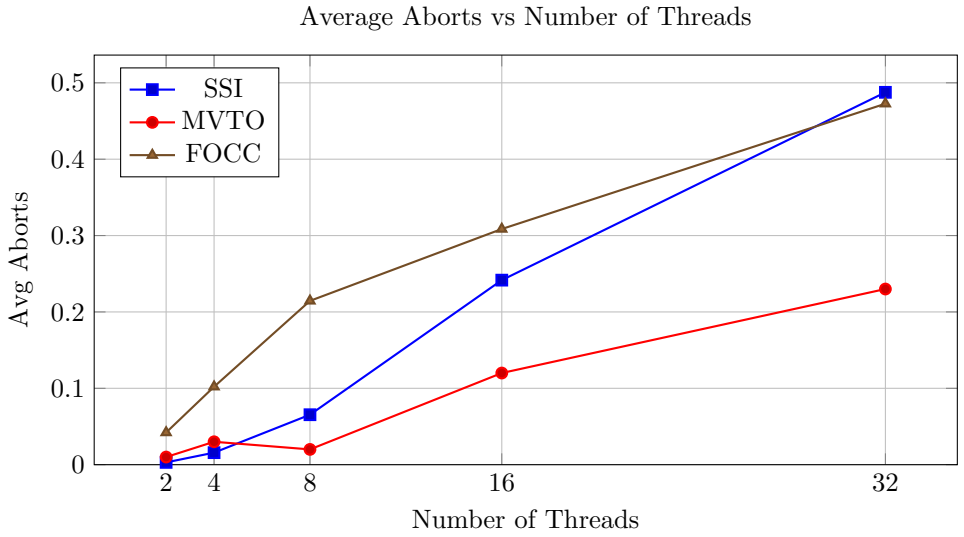

Figure 8: All algorithms show increased aborts with more threads, SSI growing fastest at high concurrency.

Figure 9: Performance vs Number of Threads

Observation: All methods experience higher abort rates as threads increase. SSI's aborts grow from 0.003 to 0.49, showing high contention at 32 threads. FOCC shows a steady rise (0.04 to 0.47), while MVTO's abort rate increases more gradually (0.01 to 0.23).

4. Varying Read-Only Percentage

Input Format: `<threads> <n> <constVal> <lambda> <readOnlyPercent> <totTrans> <numIters>`
Input: `16 1000 1000 20 <readOnlyPercent=20 to 100> 1000 20`

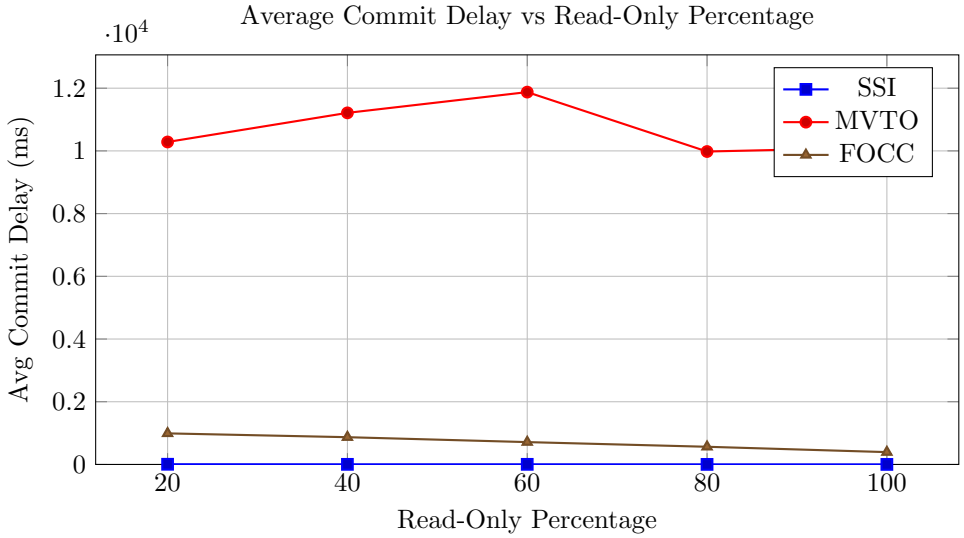

Figure 10: Commit delay decreases as more transactions are read-only. SSI maintains the lowest delay; MVTO stays high.

Observation: As the percentage of read-only transactions increases, SSI and FOCC show clear reductions in commit delay, with SSI remaining consistently lowest. MVTO's delay stays high across the board, with minimal improvement due to its reliance on version tracking rather than workload mix.

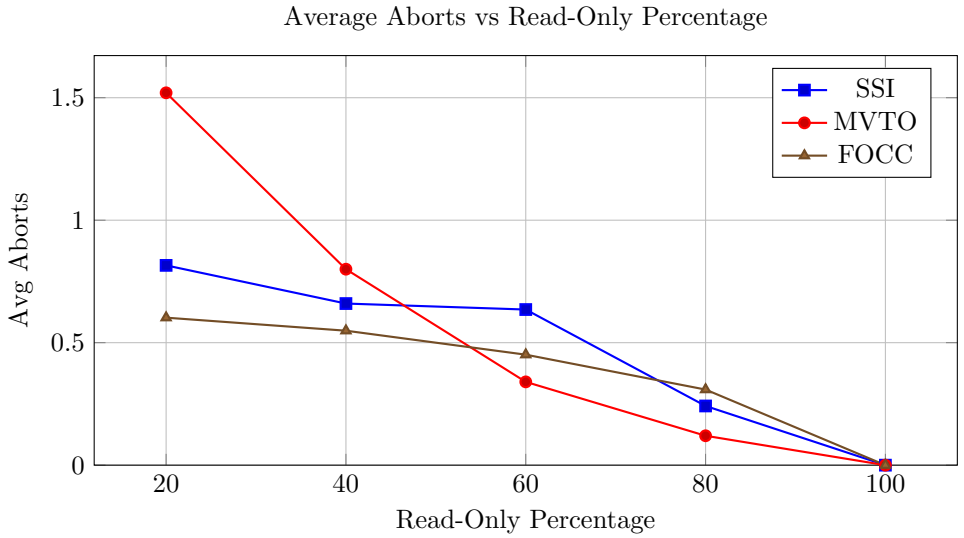

Figure 11: Abort rates decline as read-only transactions increase. SSI shows the steepest improvement.

Observation: All algorithms experience fewer aborts as the percentage of read-only transactions increases. SSI shows the most dramatic improvement. FOCC and MVTO follow similar trends but with smaller reductions.

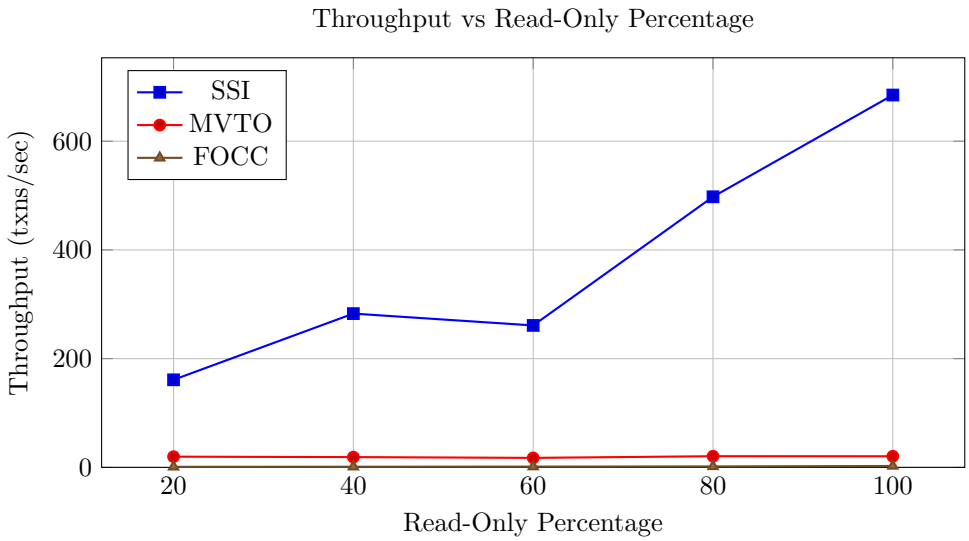

Figure 12: SSI throughput scales dramatically with more read-only transactions; FOCC improves slowly; MVTO remains mostly constant.

Observation: SSI shows significant throughput gains as the read-only workload increases — reaching nearly 685 txns/sec when all transactions are read-only. FOCC improves modestly, while MVTO remains nearly flat, as version overhead dominates regardless of workload type.

Throughput Observation: For input `16 1000 1000 20 80 1000 20`, the observed throughput values were:
MVTO: 20.52 txns/sec, **SSI:** 497.76 txns/sec, **FOCC:** 1.77 txns/sec.

Throughput Observation: For 1,000 transactions with 1,000 variables and 16 threads, **SSI significantly outperforms** both MVTO and FOCC in terms of throughput. While **SSI achieves nearly 498 transactions/sec**, MVTO lags far behind at just **20.52 transactions/sec**, and FOCC performs the worst at **1.77 transactions/sec**. This demonstrates that SSI is highly optimized for concurrency and low-latency processing under light-to-moderate transaction loads.

Conclusion

Through extensive experimentation, we have evaluated the performance of **Serializable Snapshot Isolation (SSI)** against **Multi-Version Timestamp Ordering (MVTO)** and **Forward Optimistic Concurrency Control (FOCC)** across multiple system parameters — including the number of transactions, variables, threads, and the percentage of read-only transactions.

The results consistently show that **SSI delivers superior throughput and the lowest average commit delays** across most scenarios. Its performance scales especially well as the workload becomes more read-heavy, thanks to efficient non-blocking reads, low aborts in read-only transactions, and its ability to avoid unnecessary validation delays.

MVTO, while exhibiting lower abort rates in some cases, suffers from consistently high commit delays and poor scalability due to the overhead of version maintenance. Its throughput remains mostly flat, even when the percentage of read-only transactions increases.

FOCC provides moderate performance and scalability, but it is impacted by a higher and steadily growing abort rate under concurrent workloads. While it shows some improvement with higher read-only percentages, its overall throughput remains significantly lower than SSI.

In summary, **SSI emerges as the most robust and efficient concurrency control algorithm among the three**, offering a compelling combination of serializability, low latency, and high throughput — especially under workloads with high read-only transaction percentages and varying concurrency levels.