Implementation of O2PL Scheduler

Aaryan co21btech11001
Aayush Kumar co21btech11002

1 Introduction

The Ordered Two-Phase Locking (O2PL) protocol is a concurrency control mechanism, which is an extension of the Two-Phase Locking (2PL) protocol, that allows for greater concurrency by introducing the concept of ordered sharing.

O2PL ensures serializability by acquiring locks according to the following rules [2]:

- **OS1**: This rule states that in a schedule s, for any two operations $p_i(x)$ and $q_j(x)$, $i \neq j$, such that $pl_i(x) \rightarrow ql_j(x)$ is permitted, if t_i acquires $pl_i(x)$ before t_j acquires $ql_j(x)$, then the execution of $p_i(x)$ must occur before the execution of $q_j(x)$.
- **OS2**: If we have a relationship $pl_i(x) \rightarrow ql_j(x)$ of ordered sharing between two transactions t_i and t_j, and t_i has not yet released any lock, t_j is called *order dependent* on t_i. If there exists any such transaction t_i, transaction t_j is *on hold*. While a transaction is on hold, it cannot release any of its locks.

Note: It is a standard assumption in the literature that each transaction performs at most one read and one write operation on each item, and we adopt this assumption during the implementation and experimentation of the O2PL protocol.

2 Related Work

In the literature, direct references to the practical or experimental implementation of the Ordered Two-Phase Locking (O2PL) protocol are limited. One of the few works discussing O2PL is by Agrawal et al. [1], which primarily covers the theoretical underpinnings and high-level structure of O2PL. However, it does not offer detailed insights into the full implementation of O2PL. Consequently, we began our work without existing comprehensive code examples or reference implementations. By building this scheduler from scratch, we aim to provide a clearer picture of O2PL's operational intricacies—its data structures,

locking mechanisms, and concurrency control behaviors—offering an end-to-end walk-through that helps fill the gap in detailed documentation for O2PL.

3 Algorithm

3.1 Item Object

The item object is a data structure that represents a database item in the O2PL protocol. It contains the following attributes:

- **read_op_ctr**: If a new read operation is performed on this item, read_op_ctr represents the number of operations that would be in conflict with this read operation.
- **write_op_ctr**: If a new write operation is performed on this item, write_op_ctr represents the number of operations that would be in conflict with this write operation.
- **read_item_ctr**: It represents the number of read operations that are completed on this item.
- **write_item_ctr**: It represents the number of write operations that are completed on this item.
- **read_ulock_item_ctr**: If the current schedule is s, read_ulock_item_ctr represents the number of read operations performed on this item in CP(s).
- **write_ulock_item_ctr**: If the current schedule is s, write_ulock_item_ctr represents the number of write operations performed on this item in CP(s).
- **val**: The value of the item.

The pseudo code for the item object is shown in Algorithm 3.1.

Algorithm 1 Item Class

```
1: class Item
2:     read_op_ctr, write_op_ctr: atomic integers, initially 0
3:     read_item_ctr, write_item_ctr: atomic integers, initially 0
4:     read_ulock_item_ctr, write_ulock_item_ctr: atomic integers, initially 0
5:     val: integer, initially 0
```

3.2 Transaction Object

The transaction object is a data structure that represents a transaction in the O2PL protocol. It contains the following attributes:

- **id**: An integer that uniquely identifies the transaction.

- **operations**: A map that associates each item ID with a list of pairs, where each pair consists of an operation counter and the type of operation (READ or WRITE).

The pseudo code for the transaction object is shown in Algorithm 3.2.

Algorithm 2 Transaction Class

```
1: class Transaction
2:     id: integer
3:     operations: map of (item_id → list of (op_counter, operation_type) pairs)
```

3.3 O2PL Protocol

The `O2PL` class implements the Optimistic Two-Phase Locking (O2PL) concurrency control protocol for a database system. It manages a set of items and handles transaction operations such as reading, writing, and committing. The object contains the following attributes and methods:

- **items**: A vector of pointers to `Item` objects, each representing a unique data item in the system.
- **size**: The number of items in the system.

Methods:

- `get_op_ctr`: Returns the counter assigned to an operation which denotes the number of operations that would be in conflict with this operation.
  - If the operation is a `READ`, it fetches the `read_op_ctr` and increments `write_op_ctr`, indicating that this read operation will conflict with future writes.
  - If it is a `WRITE`, it fetches `write_op_ctr` and increments both `read_op_ctr` and `write_op_ctr`, reflecting this write operation will conflict with future reads and writes.
- `read`:
  - Gets the operation counter for the read.
  - Waits until all previous write operations (indicated by `write_item_ctr`) are complete. **Note that this waiting is analogous to acquiring a lock**.
  - Reads the value of the item and stores it in `locVal`.
  - Appends the operation and the counter to the transaction's operations list.
  - Increments `read_item_ctr`, indicating that this read operation is complete.

- `write`:
  - Gets the operation counter for the write.
  - Waits until all prior reads and writes (indicated by `write_item_ctr + read_item_ctr`) are complete.
  - Writes the new value into the item.
  - Appends the operation and the counter to the transaction's operations list.
  - Increments `write_item_ctr`, indicating that this write operation is complete.
- `commit`: This method ensures that OS2 is satisfied. It achieves this in the following way:
  - For each operation in the transaction, it checks if the operation is a single read or write. If it is, it waits for the transactions, on which this transaction is order dependent, to unlock.
  - If not, we make sure that the transaction doesn't wait for one of the operations of its own.
  - Waiting is achieved by comparing the operation counter stored in transaction's `operations` map with the `write_ulock_item_ctr` and `read_ulock_item_ctr` of the item.
  - Finally, it increments the `read_ulock_item_ctr` or `write_ulock_item_ctr` of the item based on the operation type. **Note that this is analogous to releasing a lock**.

The pseudo code for the O2PL class is shown in Algorithm 3.3 and Algorithm 3.3.

4 Proof of Correctness

The proof of correctness involves showing that the protocol satisfies the properties of OS1 and OS2. Here are the proofs for both properties:

4.1 Proof of OS1

First, consider the case when $p_i(x)$ and $q_j(x)$ are such that $p_i(x) <_s q_j(x)$ and no other operation on x is executed between $p_i(x)$ and $q_j(x)$. We analyze the following cases:

p = r, q = r : According to O2PL, two read operations on the same item should not be order-dependent. Suppose $p_i(x)$ is assigned a counter `ctr1` in Line 16. Since $p = r$, only `write_op_ctr` of x is incremented in the method `get_op_ctr` (Line 5). Therefore, $q_j(x)$ will also be assigned the counter `ctr1` in Line 16. Since $p_i(x)$ and $q_j(x)$ are assigned the same

Algorithm 3 O2PL Class: Core Methods

```
1: class O2PL
2:     items: array of Item objects
3:     size: integer
4:
5:     private method get_op_ctr(item_id, op)
6:         if op = READ then
7:             op_ctr ← items[item_id].read_op_ctr
8:         else
9:             op_ctr ← items[item_id].write_op_ctr
10:            items[item_id].read_op_ctr++
11:        end if
12:        items[item_id].write_op_ctr++
13:        return op_ctr
14:
15:    method read(t, item_id, locVal)
16:        op_ctr ← get_op_ctr(item_id, READ)
17:        wait until op_ctr > items[item_id].write_item_ctr
18:        locVal ← items[item_id].val
19:        t.operations[item_id].append((op_ctr, READ))
20:        items[item_id].read_item_ctr++
21:
22:    method write(t, item_id, newVal)
23:        op_ctr ← get_op_ctr(item_id, WRITE)
24:        wait until op_ctr > items[item_id].write_item_ctr + items[item_id].read_item_ctr
25:        items[item_id].val ← newVal
26:        t.operations[item_id].append((op_ctr, WRITE))
27:        items[item_id].write_item_ctr++
```

Algorithm 4 O2PL Class: Commit Method

```
1: method commit(t)
2:     for each (item_id, v) in t.operations do
3:         if v.size() = 1 then
4:             ctr ← v[0].first
5:             op ← v[0].second
6:             if op = READ then
7:                 wait until ctr > items[item_id].write_ulock_item_ctr
8:             else
9:                 wait until ctr > items[item_id].write_ulock_item_ctr +
   items[item_id].read_ulock_item_ctr
10:            end if
11:        else
12:            for i = 0 to v.size()-1 do
13:                ctr ← v[i].first
14:                op ← v[i].second
15:                if op = READ then
16:                    wait until ctr > items[item_id].write_ulock_item_ctr +
   (v.back().second = READ ? 1 : 0)
17:                else
18:                    wait until ctr > items[item_id].write_ulock_item_ctr +
   items[item_id].read_ulock_item_ctr + (v.back().second = WRITE ? 1 : 0)
19:                end if
20:            end for
21:        end if
22:    end for
23:
24:    for each (item_id, v) in t.operations do
25:        for i = 0 to v.size()-1 do
26:            ctr ← v[i].first
27:            op ← v[i].second
28:            if op = READ then
29:                items[item_id].read_ulock_item_ctr++
30:            else
31:                items[item_id].write_ulock_item_ctr++
32:            end if
33:        end for
34:    end for
```

counters, they can execute concurrently when `write_item_ctr` becomes equal to `ctr1`.

p = r, q = w : According to O2PL, there will be an ordered sharing of lock $pl_i(x) \rightarrow ql_j(x)$ between $p_i(x)$ and $q_j(x)$. Therefore, $q_j(x)$ should wait for $p_i(x)$ to complete. Suppose, before $p_i(x)$, the value of `read_op_ctr` was rc and `write_op_ctr` was wc. When $p_i(x)$ comes, it is assigned a counter `ctr1` in Line 16 equal to rc. Since $p = r$, `write_op_ctr` of x is incremented to $wc + 1$. Therefore, $q_j(x)$ will be assigned a counter `ctr2` $= wc + 1$ in Line 23. $p_i(x)$ will execute when `write_item_ctr` of x becomes equal to `ctr1`. Just before $p_i(x)$ executes, there are wc operations done, of which rc are write operations. Thus, `write_item_ctr` is rc and `read_item_ctr` is $wc - rc$. After $p_i(x)$ executes, `read_item_ctr` becomes $wc - rc + 1$. Now, $q_j(x)$ checks if `ctr2` is greater than `write_item_ctr` + `read_item_ctr`. After $p_i(x)$ executes, this sum is $rc + wc - rc + 1 = wc + 1$, allowing $q_j(x)$ to execute. This ensures $q_j(x)$ will execute after $p_i(x)$.

p = w, q = r : According to O2PL, there will be an ordered sharing of lock $pl_i(x) \rightarrow ql_j(x)$ between $p_i(x)$ and $q_j(x)$. Therefore, $q_j(x)$ should wait for $p_i(x)$ to complete. Suppose, before $p_i(x)$, `read_op_ctr` was rc and `write_op_ctr` was wc. When $p_i(x)$ comes, it is assigned a counter `ctr1` in Line 23 equal to wc. Since $p = w$, both `read_op_ctr` and `write_op_ctr` are incremented to $rc + 1$ and $wc + 1$ respectively. Therefore, $q_j(x)$ will be assigned a counter `ctr2` $= rc + 1$ in Line 16. $p_i(x)$ will execute when `write_item_ctr` + `read_item_ctr` becomes equal to `ctr1`. Since `read_op_ctr` is incremented only by write operations, before $p_i(x)$ executes, there must be rc write operations. Thus, after all prior operations, `write_item_ctr` is rc. Once $p_i(x)$ executes, `write_item_ctr` becomes $rc + 1$. Now, $q_j(x)$ checks if `ctr2` is greater than `write_item_ctr`. After $p_i(x)$ executes, `write_item_ctr` is $rc + 1$, allowing $q_j(x)$ to execute. This ensures $q_j(x)$ will execute after $p_i(x)$.

p = w, q = w : According to O2PL, two write operations on the same item will be order-dependent. Suppose, before $p_i(x)$, `read_op_ctr` was rc and `write_op_ctr` was wc. When $p_i(x)$ comes, it is assigned a counter `ctr1` in Line 23 equal to wc. Since $p = w$, both `read_op_ctr` and `write_op_ctr` are incremented to $rc + 1$ and $wc + 1$ respectively. Therefore, $q_j(x)$ will be assigned a counter `ctr2` $= wc + 1$ in Line 23. $p_i(x)$ will execute when `write_item_ctr` + `read_item_ctr` becomes equal to `ctr1`. After it executes, `write_item_ctr` is incremented by 1. When $q_j(x)$ executes, it checks if `ctr2` is greater than `write_item_ctr` + `read_item_ctr`. After $p_i(x)$ executes, this sum is incremented by 1, so it becomes equal to `ctr2`, allowing $q_j(x)$ to execute. This ensures $q_j(x)$ will execute after $p_i(x)$.

Proof of Correctness of O2PL for $n > 2$ operations: We have already proved that the protocol correctly handles $n = 2$ adjacent operations. We now extend the proof to any arbitrary $n > 2$ using mathematical induction.

Base Case: As shown above, the correctness holds for $n = 2$ operations.

Inductive Hypothesis: Assume that for some $k \geq 2$, the O2PL protocol ensures the correct execution order for any k operations on a data item x, preserving the ordering constraints specified by O2PL (e.g., reads not ordered among themselves, reads before writes, writes before reads, writes ordered among themselves).

Inductive Step: We now prove that the protocol ensures correct execution for $k + 1$ adjacent operations.

Consider a set of $k+1$ adjacent operations $\{o_1(x), o_2(x), \ldots, o_k(x), o_{k+1}(x)\}$. By the inductive hypothesis, the first k operations execute correctly according to O2PL constraints.

Now, when the $(k + 1)$-th operation $o_{k+1}(x)$ arrives, it is assigned an operation counter based on its type (`read` or `write`) using the rules outlined earlier:

- If $o_{k+1}(x)$ is a read, it is assigned a counter equal to the current `read_op_ctr`, and `write_op_ctr` is incremented.
- If $o_{k+1}(x)$ is a write, it is assigned a counter equal to the current `write_op_ctr`, and both `read_op_ctr` and `write_op_ctr` are incremented.

Depending on the type of $o_{k+1}(x)$ and its interactions with previous operations:

- If $o_{k+1}(x)$ is a read:
  - It waits until all writes before it (which have lower counters) complete.
  - It does not need to wait for other reads with the same counter value (concurrent reads are allowed).
- If $o_{k+1}(x)$ is a write:
  - It waits for all previous reads and writes (having lower counter values) to complete.

Since the counters are assigned monotonically (increasing with each operation according to the protocol rules) and each operation waits appropriately based on these counters, $o_{k+1}(x)$ will only execute after all operations it must be ordered after have completed, and it will execute concurrently only with operations it is allowed to share execution with (e.g., reads).

Thus, the correctness extends from k operations to $k + 1$ operations.

Conclusion: By the principle of mathematical induction, O2PL correctly enforces the required execution ordering constraints for any number of operations $n \geq 2$. □

4.2 Proof of OS2

The proof of OS2 shows that a transaction t_j correctly waits for all transactions t_i on which it is order-dependent, before releasing any locks.

In the O2PL protocol:

- During the locking phase, every operation performed by a transaction is assigned an operation counter using `get_op_ctr`. This counter represents the number of prior conflicting operations.
- This assigned counter is stored in the transaction's `operations` map at the time the operation is performed.
- During the unlocking phase (commit phase), for each operation performed by a transaction, the protocol waits until the corresponding unlock counter of the item (`read_ulock_item_ctr` or `write_ulock_item_ctr`) reaches the operation's assigned counter value.
- Waiting for unlock is achieved through the condition checks at Lines 16 and 18.
- The increment of `read_ulock_item_ctr` or `write_ulock_item_ctr` at Lines 29 and 31 is equivalent to unlocking the respective operation.

The unlocking mechanism is carefully designed to satisfy OS2:

- Since the order of unlocking is determined by the operation counters (which were assigned monotonically during the locking phase based on conflict ordering), a transaction t_j will naturally wait for all transactions t_i (where t_i performed conflicting operations earlier) to unlock their locks before proceeding.
- The protocol ensures that a transaction does not wait for its own operation to unlock by using an if-else condition that verifies whether the operation's counter is associated with the transaction itself. This avoids deadlocks or unnecessary waiting on itself.
- Therefore, if t_j is order dependent on t_i, t_j will be put on hold until t_i has released the corresponding lock (i.e., incremented the appropriate unlock counter).

Since the unlocking counters are incremented in the same order as the execution of conflicting operations (as proven in OS1), and since transactions wait based on these counters, the mechanism for ensuring OS2 is structurally similar to that used for ensuring OS1. Hence, the correctness proof for OS2 follows a similar inductive argument:

- For two conflicting operations, the transaction unlocking follows the execution order ensured by their operation counters.

- By induction, for any sequence of operations and any number of transactions, the unlock ordering will preserve the necessary waiting and unlock rules imposed by OS2.

Thus, O2PL satisfies OS2: A transaction on hold waits for all transactions on which it is order dependent, and no transaction releases any lock until all such dependencies are cleared.

□

5 Experiments

Our experiments were conducted on **AMD EPYC 7452 32-Core Processor** with **256** GB RAM. The code was compiled using **g++ 7.5.0** without any compiler optimizations. All the times reported are in microseconds.

5.1 Input

The input schedules were generated using a protocol similar to BTO in the following way:

- We maintain r_max and w_max for each data item as the maximum timestamp of the transaction that has read or written the item.
- Each transaction tries to access 20 random data items.
- For each read operation, we check if the transaction is late by comparing it's timestamp with w_max of the item. If it is late, we skip the read operation.
- Similarly, for each write operation, we check if the transaction is late by comparing it's timestamp with $max(r_max, w_max)$ of the item. If it is late, we skip the write operation.
- If the operation is executed, we update the value of r_max or w_max of the item.

This protocol ensures that the input schedule does not have cycles, and hence avoids deadlocks.

5.2 Experiment 1

We fix the number of threads to 16, number of items to 5000, write probability to 0.2 and vary the total number of transactions from 1000 to 5000 in increments of 1000. The execution times for the four algorithms are shown in Table 1 and Figure 1.

We can see that O2PL is faster than SS2PL, BOCC and FOCC. The execution time of all the algorithms remains almost constant with the number of transactions for all the algorithms. This is since the performance is determined

Input	O2PL	SS2PL	BOCC	FOCC
1000	306.478	333.327	321.639	326.939
2000	305.191	316.737	314.339	318.856
3000	303.895	316.302	317.859	318.748
4000	305.738	312.062	313.729	320.449
5000	311.911	319.306	314.960	322.756

Table 1: Execution times for changing total number of transactions.

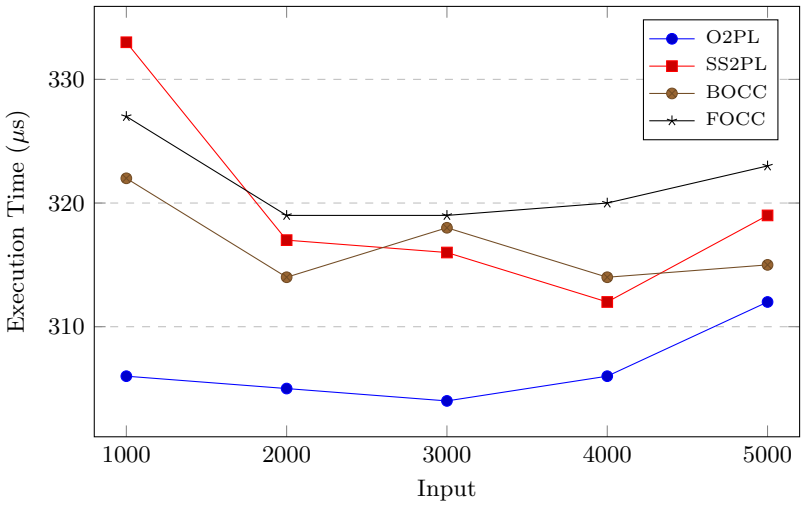

Figure 1: Execution times for changing total number of transactions.

mainly by the number of concurrent transactions and not the total number of transactions. As the number of transactions increases, total time taken increases linearly but the average time per transaction remains constant.

5.3 Experiment 2

We fix the number of threads to 16, number of transactions to 5000, write probability to 0.2 and vary the number of items in the database from 1000 to 5000 in increments of 1000. The execution times for the four algorithms are shown in Table 2 and Figure 2.

It was initially expected that increasing the database size would lead to fewer conflicts and therefore reduced execution time. However, due to the way the input is generated, a higher number of conflicts results in more operations being skipped. As a result, execution time actually increases when conflicts decrease. This highlights a limitation of the input generation method. However, the performance of O2PL remains consistently better than the other algorithms.

Input	O2PL	SS2PL	BOCC	FOCC
1000	305.957	308.123	301.884	307.034
2000	304.228	311.364	310.661	309.742
3000	306.526	314.767	309.182	316.388
4000	304.204	314.477	311.336	317.936
5000	308.571	315.156	318.675	319.067

Table 2: Execution times for changing number of items in the database.

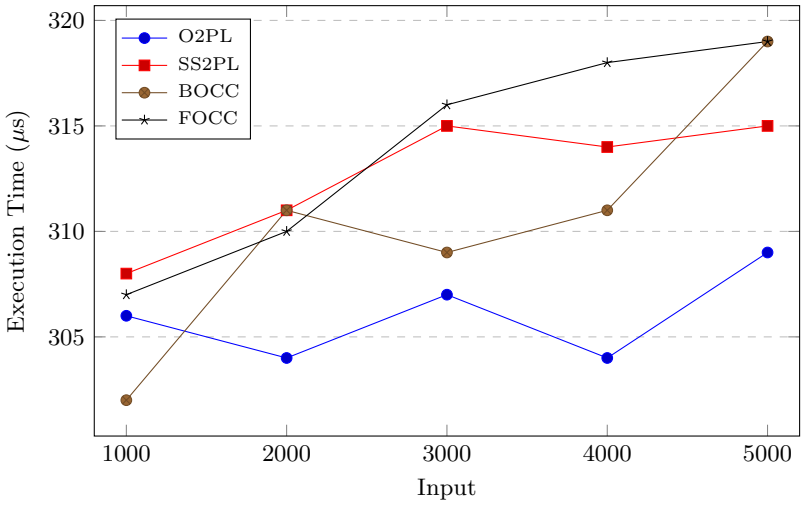

Figure 2: Execution times for changing number of items in the database.

5.4 Experiment 3

We fix the number of transactions to 5000, number of items to 5000, write probability to 0.2 and vary the number of threads from 2 to 64 in increments of 2. The execution times for the four algorithms are shown in Table 3 and Figure 3.

The average time taken per transaction increases with the number of threads for all the algorithms. This is because more threads lead to increased probability of conflicts between transactions. However, O2PL consistently outperforms the other algorithms on average.

5.5 Experiment 4

We fix the number of transactions to 5000, number of items to 5000, number of threads to 16 and vary the write probability from 0.1 to 0.5 in increments of 0.1. The execution times for the four algorithms are shown in Table 4 and Figure 4.

The average time taken per transaction increases with the write probability

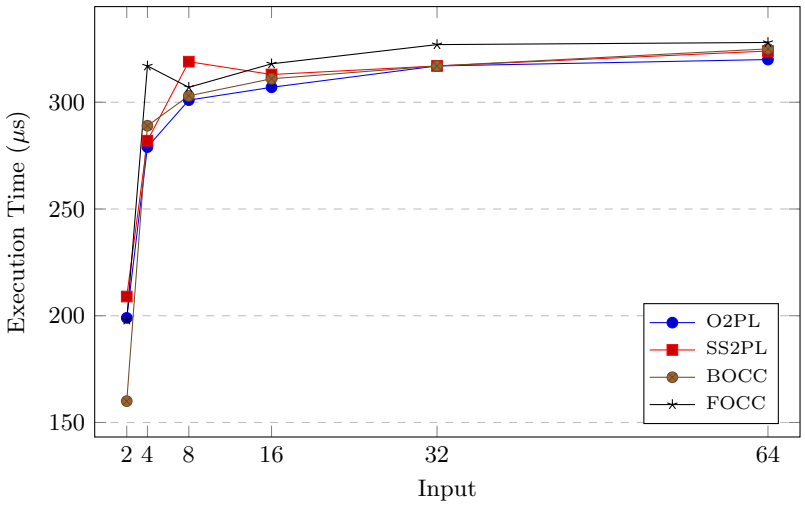

Figure 3: Execution times for changing number of threads.

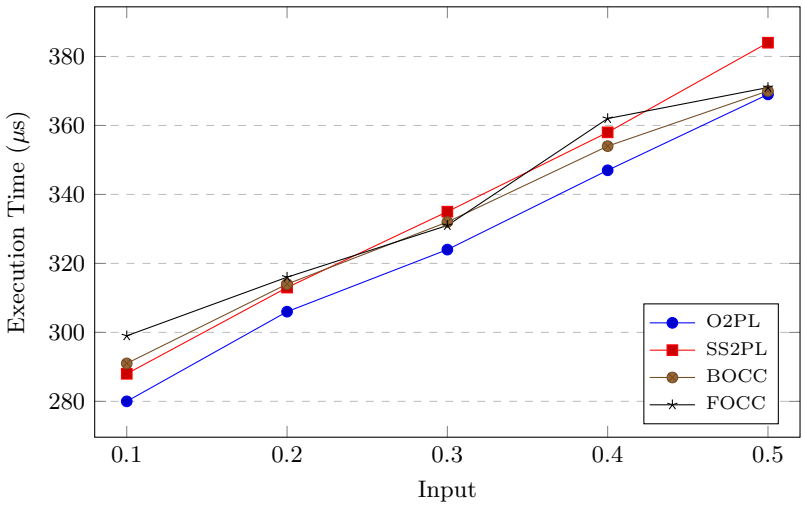

Figure 4: Execution times for changing write probability.

Input	O2PL	SS2PL	BOCC	FOCC
2	199.081	208.971	160.389	198.227
4	279.347	281.953	289.283	317.335
8	300.568	318.788	303.208	307.180
16	307.004	313.341	310.887	317.714
32	317.243	316.688	317.160	326.857
64	320.229	323.540	324.832	327.556

Table 3: Execution times for changing number of threads.

Input	O2PL	SS2PL	BOCC	FOCC
0.1	280.482	287.817	290.903	299.312
0.2	306.166	312.656	313.671	316.416
0.3	324.185	335.363	332.147	331.184
0.4	347.154	358.157	354.376	361.595
0.5	369.166	384.276	369.866	370.970

Table 4: Execution times for changing write probability.

for all the algorithms. This is because more writes lead to increased probability of conflicts between transactions. However, O2PL consistently outperforms the other algorithms on average.

6 Conclusion and Future Work

We evaluated the performance of O2PL, SS2PL, BOCC, and FOCC under varying workloads. Overall, O2PL consistently outperformed the other algorithms across all scenarios. While some results followed expected trends—such as increased contention with higher thread count or write probability—others, like the effect of database size, revealed unexpected behavior due to the input generation method. These findings highlight both the robustness of O2PL and the importance of realistic workload modeling when benchmarking concurrency control mechanisms.

Overall, these results suggest that O2PL offers robust performance and is less sensitive to conflict-heavy environments, making it a strong candidate for systems requiring scalable concurrency control.

There are several directions for future work:

- Explore different input generation methods to better understand the impact of database size on performance.
- Compare O2PL with other concurrency control algorithms not covered in this study.
- Analyze the scalability of O2PL in distributed systems or cloud environments.

References

[1] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. Ordered shared locks for real-time databases. *The VLDB Journal*, 4(1):87–126, 1995.

[2] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.