Project Report: Practical implementations of O2PL and Optimistic O2PL

Tushita Sharva
CS21BTECH11022

P Gayathri Shreeya
CO21BTECH11010

1 Problem Statement

In this project, we aimed to develop a practical implementation of O2PL scheduler.

Parameters

- n: Number of threads.
- m: Number of shared data items.
- $totalTrans$: Total number of committed transactions required.
- $constVal$: Parameter for simulating increment operations on data items.
- λ: Parameter for simulating complex operations in transactions.
- $numIters$: Number of read and write actions per transaction.
- $readRatio$: Parameter for making transactions read-only.

2 Program Design of O2PL

The project consists of several classes implemented in C++.

2.1 Classes & Their Methods

2.1.1 Logger Class

Handles logging for output and debugging. Provides time-stamped log entries for transaction execution and validation.

2.1.2 Transaction Class

Maintains attributes of a transaction. Key members:

- **transactionId**: Unique transaction identifier.
- **threadId**: thread identifier of the transaction.
- **status**: Transaction state (active, committed, aborted).

2.1.3 Node Classes

Our algorithm enforces an ordered locking mechanism using a linked list, where each node represents a lock request by a transaction. The list of nodes is maintained for each data item, and a transaction is assigned a node depending on the locking table. It is allowed to proceed only when its corresponding node reaches the head of the list. To achieve this, each node contains a *condition variable*. *When a transaction submits its lock request, it receives a reference to its node and then waits on the node's condition variable*. The transaction is unblocked and allowed to execute only when it becomes the head of the list, ensuring strict ordering and preventing race conditions.

```
class Node {
public:
    Node *next;
    OperationType type;

    // Variables for signalling waiting threads
    std::mutex mtx;
    std::condition_variable cv;
    bool isAtHead;

    Node(OperationType t) : type(t) { next = nullptr; isAtHead = false;}
};

class ReadNode : public Node {
public:
    std::set<Transaction*> tlist;
};

class WriteNode : public Node {
public:
    Transaction *transaction;
};
```

The `NodeList` class provides functions that are invoked by the `DataItem` class to manage lock ordering and synchronization. These functions are abstracted to hide the internal details from the `DataItem` class, allowing it to interact with the locking mechanism without knowing its implementation specifics. This design promotes decoupling.

```
class NodeList {
public:
    Node *head;
    Node *tail;

    NodeList();
    ~NodeList();
    Node* addReadNode(Transaction *t);
    Node* addWriteNode(Transaction *t);
    void deleteReadNode(Transaction *t);
    void deleteWriteNode(Transaction *t);
};
```

2.1.4 DataItem Class

As discussed, each `DataItem` maintains a `nodeList` for ordered lock acquisition, along with `readList` and `writeList` sets to support construction of the waits-for graph. A mutex lock is also included to ensure synchronization during concurrent access and modifications.

```
class DataItem {
public:
    int value;
    NodeList *nodeList;
    std::set<int> readList; // will be used by WFG graph
    std::set<int> writeList; // will be used by WFG graph
    std::mutex datalock;

    // Functions to manipulate the nodeList
    Node* addRead(Transaction *t); void deleteRead(Transaction *t);
    Node* addWrite(Transaction *t); void deleteWrite(Transaction *t);
};
```

2.1.5 Scheduler class

The `Scheduler` acts as an interface between transactions and shared data items. When a transaction initiates an operation, it delegates the request to the scheduler. The scheduler then interacts with the relevant `DataItem` to validate and process the operation. Based on the outcome, it either applies the changes and returns control to the transaction, or signals the transaction to abort if the operation violates concurrency control rules.

```
class Scheduler {
public:
    WaitsForGraph* G;
    std::mutex graphLock;
    std::vector<std::shared_ptr<DataItem>> shared;
    std::atomic<int> counter;

    void init(int m);
    Transaction* begin_trans(int threadID);
    bool read(Transaction* t, int index, int &localVal);
    bool write(Transaction* t, int index, int localVal);
    TransactionStatus tryCommit(Transaction* t);
};
```

2.2 Implementation Method

1. The `main` function starts multiple threads. Each thread asks the scheduler to create a new transaction, which it will use to perform operations.
2. Transactions begin and send read/write requests to the scheduler.
3. For each request, the scheduler checks with the `WaitsForGraph` to see if the operation would cause a deadlock.
   (a) If safe, the operation is added to a linked list for that data item. The transaction waits on the node it gets in this list. When its node reaches the head, the operation is performed.
   (b) If not safe, the transaction is aborted.
4. The system collects metrics like commit times and abort counts during execution.

2.3 Proof Sketch: Gen(O2PL) $\subset$ OCSR

Let each transaction T_i be assigned a unique, increasing transaction ID at the time of creation. Define a total order $\prec$ such that $T_i \prec T_j$ iff $i < j$.

Locking Discipline:

- Each `DataItem` maintains a FIFO linked list (via `NodeList`) for read/write operations.
- A transaction T_i is blocked on a node's condition variable until it reaches the head of the queue.
- Thus, operations are executed in the order of lock requests, which reflect the transaction ID order.

Conflict Serialization Graph (CSG):

- If T_i's operation precedes and conflicts with T_j's on the same data item, and $i < j$, then a dependency edge $T_i \to T_j$ is induced.
- Due to the ordered queueing mechanism, T_j can only execute after T_i releases its lock.

Conclusion: All conflicting operations are executed in transaction ID order, and no reverse edges ($T_j \to T_i$ where $j > i$) are possible in the conflict graph.

Hence, the protocol guarantees that the conflict graph is a DAG with edges respecting $\prec$, and thus the schedule is **Ordered Conflict Serializable (OCSR)**.

3 Program Design of Optimistic O2PL

3.1 Class Structure and Modifications

The class architecture of the optimistic O2PL protocol closely mirrors that of the non-optimistic variant, with a few key modifications to enable optimistic execution and dependency tracking:

- **Scheduler:**
  - Maintains a thread-safe `committedTransactions` set, which records all transactions that have either committed or aborted.
  - Uses a `commitLock` to ensure synchronized access to this set.
- **Transaction:**
  - Includes a `dependencySet` to track transactions that must complete before this transaction can commit.
  - Maintains a `localWrites` to hold tentative writes until the commit phase.

3.2 Execution Flow

1. The `main` function spawns multiple threads. Each thread requests a new transaction from the scheduler, which it uses to issue read/write operations.
2. As transactions execute, they issue read and write requests to the scheduler. These are handled optimistically:
   - **Read Request:**
     - The read is performed immediately.
     - The transaction is added to the read list of the accessed data item.
     - To identify dependencies, the transaction inspects the write set of the data item and adds any uncommitted writing transactions to its `dependencySet`.
   - **Write Request:**
     - The write is stored in the transaction's `localWrites` instead of being applied immediately.
     - The transaction is also added to the read list of the data item.
     - Dependencies are determined by inspecting both the read and write sets of the data item. All uncommitted readers and writers are added to the `dependencySet`.
3. When a transaction reaches the `tryCommit` phase:
   - The system checks whether the transaction introduces a cycle in the *waits-for* graph.
   - If a cycle is detected, the transaction is aborted to avoid deadlock.
   - Otherwise, the transaction waits for all transactions in its `dependencySet` to either commit or abort.
   - Once its dependencies are resolved and it is safe to proceed, the transaction acquires locks on all relevant data items and applies its local writes to shared memory.
4. During execution, the system tracks metrics such as transaction commit times, number of aborts, and dependency resolutions for analysis.

3.3 Design Advantage

By deferring actual writes and performing cycle detection only during the commit phase, the system eliminates the need for a centralized `nodelist`, reducing overhead and improving scalability.

3.4 Proof of Correctness: Gen(Optimistic O2PL) $\subset$ CSR

- The protocol tracks dependencies at runtime via each transaction's `dependencySet`.
- Before committing, a transaction T_k checks if adding itself to the current WFG would introduce a cycle.
- If a cycle is detected, T_k is aborted, thereby preventing cyclic dependencies among committed transactions.

Proof:

1. Construct the serialization graph $SG(H)$, where:
   - Each node corresponds to a committed transaction.
   - An edge $T_i \to T_j$ exists if:
     - T_i and T_j access the same data item with at least one being a write, and
     - the conflicting operation of T_i occurs before that of T_j in H.
2. The Optimistic O2PL protocol prevents any committed transaction T_k from creating a cycle in the WFG by checking for cycles before commit.
3. Since only acyclic dependency graphs are permitted to commit, and aborting transactions are excluded from $SG(H)$, the resulting serialization graph is acyclic.

Conclusion: Since every history H generated by the protocol has an acyclic serialization graph, we conclude that:

$$\text{Gen(Optimistic O2PL)} \subseteq \text{CSR}$$

3.5 Proof Sketch: Optimistic O2PL $\subseteq$ OCSR

Let each transaction T_i be assigned a unique, increasing transaction ID at start time. We define a total order $\prec$ such that $T_i \prec T_j$ iff $i < j$.

We aim to prove that the conflict graph produced by the protocol has only edges that respect this total order, i.e., for every edge $T_i \to T_j$, it holds that $i < j$.

Dependency Construction: During execution:

- When T_i performs a `read` or `write` on a data item x, it inspects x's `readList` and `writeList`.
- It adds to its `dependencySet` only those transactions T_j such that $j < i$ and T_j is uncommitted.
- If any such T_j exists with $j > i$, T_i aborts immediately (OCSR enforcement).

Commit Condition: Transaction T_i proceeds to commit only if:

1. It does not introduce a cycle in the waits-for graph.
2. All transactions in its `dependencySet` (i.e., all T_j with $j < i$) have already committed or aborted.

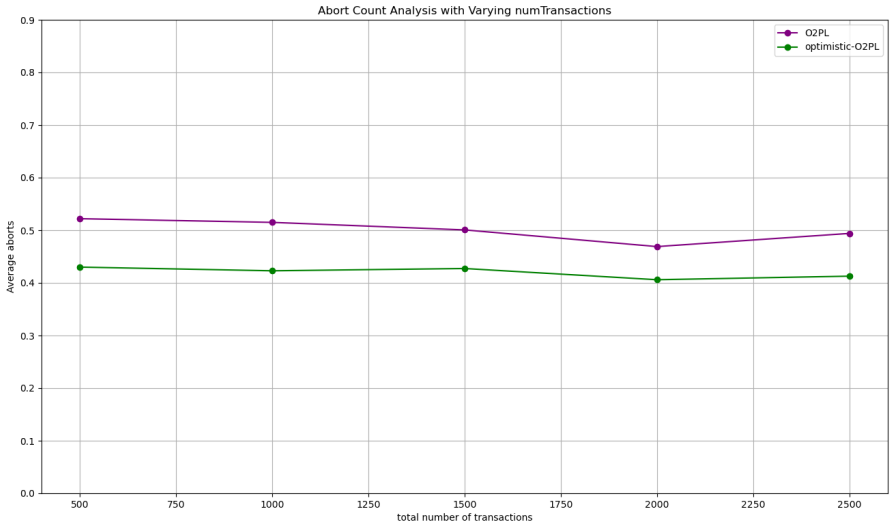
Conflict Graph Properties:

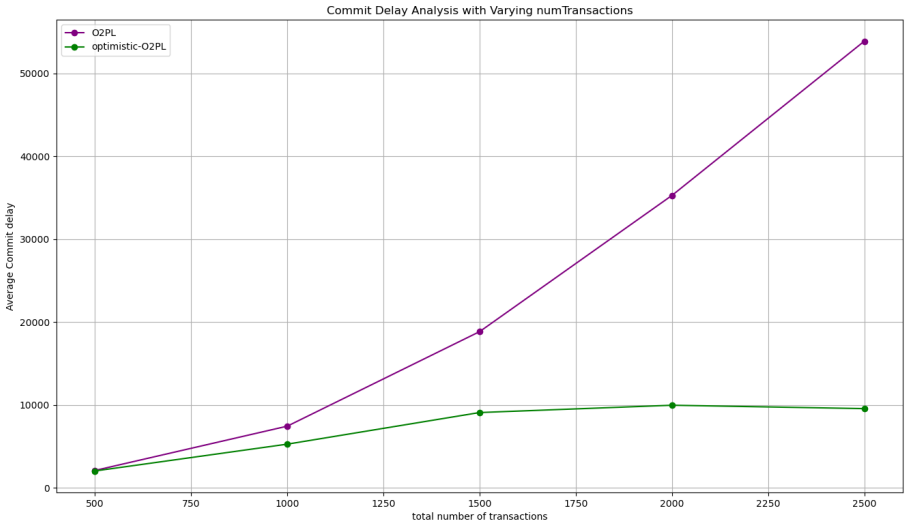
- Edges in the conflict graph correspond to dependencies.
- By construction, all dependencies $T_i \to T_j$ satisfy $j < i$.
- Hence, all edges in the conflict graph respect the total order $\prec$.

Conclusion: Since the conflict graph has only forward edges with respect to transaction IDs and is acyclic (cycle check during commit), the resulting schedule is **Ordered Conflict Serializable (OCSR)** under the ID order.

4 Experimental Results

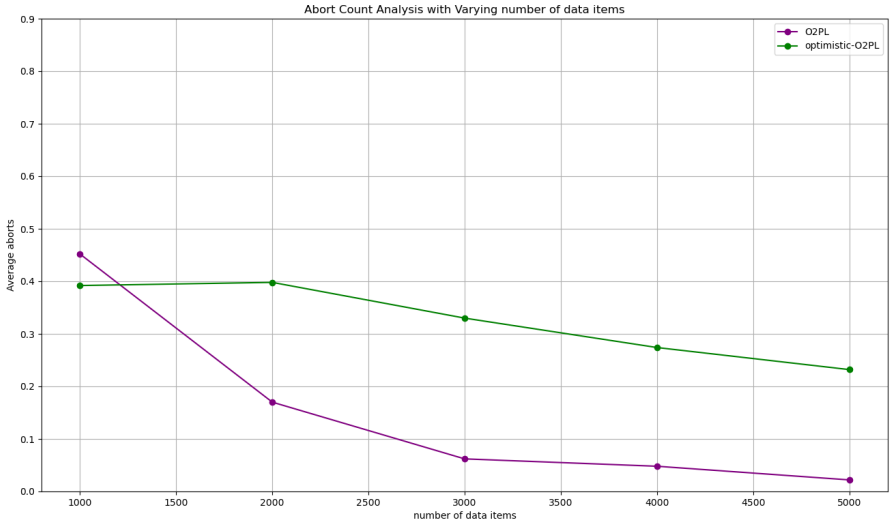
4.1 Experiment 1: Varying number of transactions (16 threads)

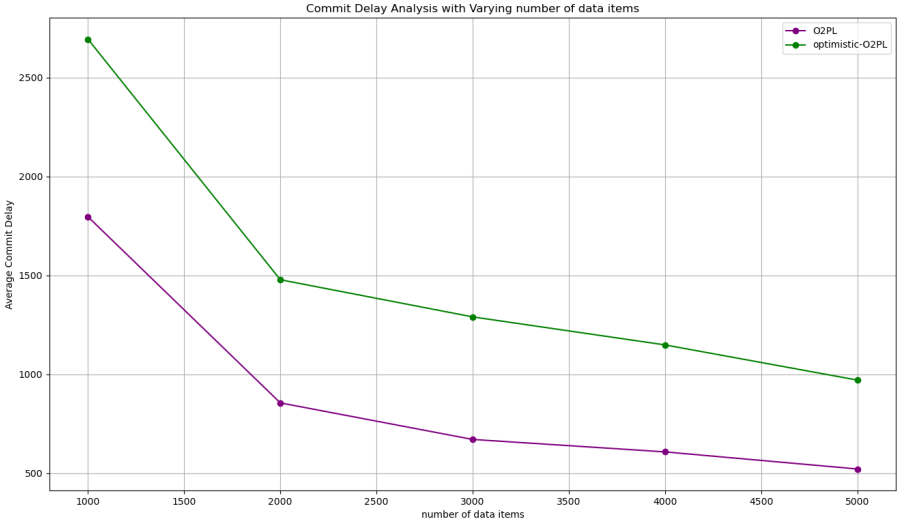




Explanation Optimistic O2PL aborts fewer transactions because it checks conflicts only at commit time, therefore unecessary aborts are avoided. It also has lower commit delays since transactions don't block during execution.

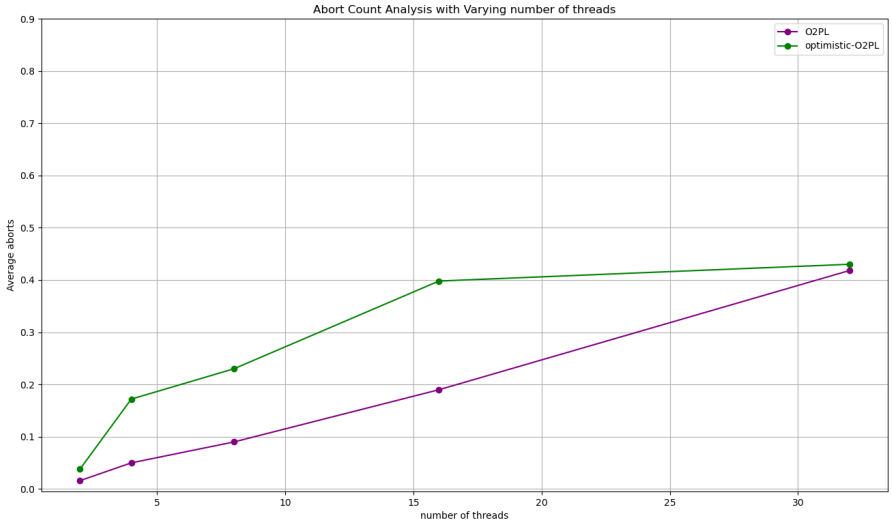
4.2 Experiment 2: Varying number of shared variables

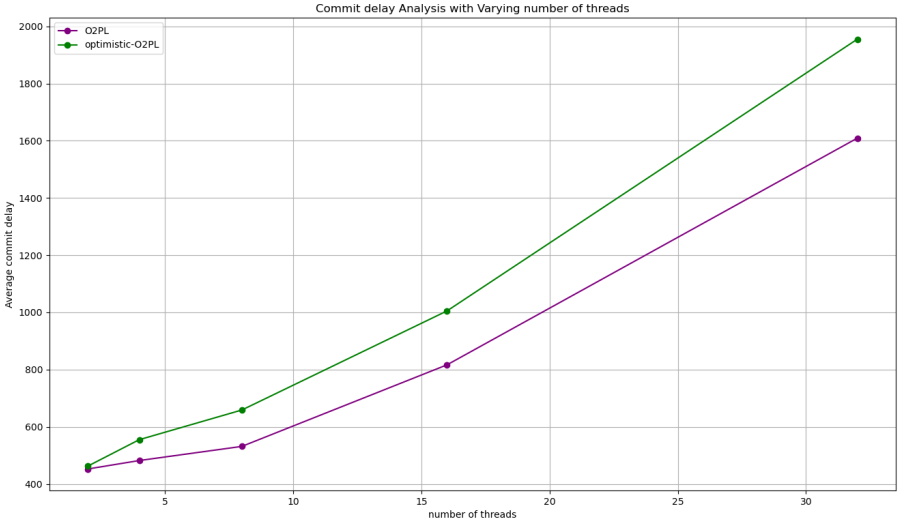




Explanation IN both O2PL and Optimistic O2PL protocols, the abort count and commit delay reduce with increased data items, which is expected because the more the data items, the more we are decreasing the likelihood of conflicts among the transactions and thus reducing aborts and commit delays. However, we observed O2PL performed better than optimistic version, which is not expected, since we are allowing them to pass and are checking only at the end. One possible explanation is that they have to wait for all dependencies to clear, which was not required in O2PL, because we assume without loss of generality that all the unlocks happen at the end of schedule.

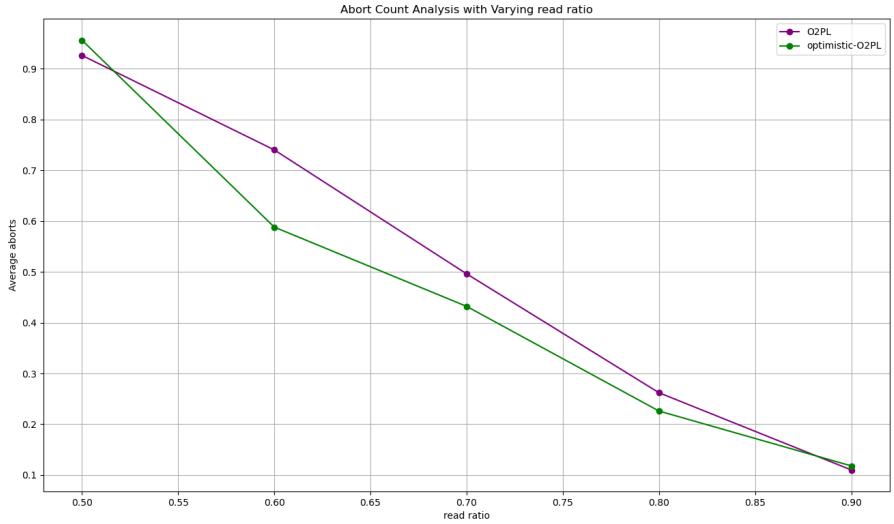
4.3 Experiment 3: Varying number of threads

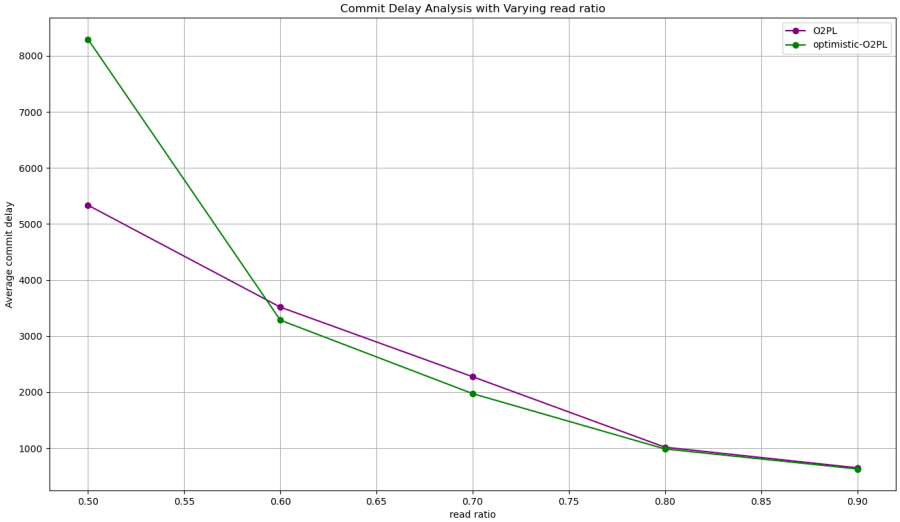




Explanation By increasing the threads, we see the abort count and commit delays increase, which is expected because we are increasing the number of concurrent transactions, not dividing work but we are creating more competition. Hence the result. In comparison, we see that pessimistic is performing better than optimistic and the possible explanation is the need for optimistic O2PL to wait on dependency set.

4.4 Experiment 4: Varying value of read ratio





Explanation Optimistic O2PL performs well with high read ratios because reads don't block and cause fewer conflicts. With more read operations, each transaction's dependency set becomes smaller, making validation faster and also reducing aborts.