Concurrency Control in Transactional Systems CS5280 PROJECT (FINAL REVIEW)

GROUP 5

Indian Institute of Technology, Hyderabad

Introduction

Project Motivation

Our goal is to implement an **efficient O2PL scheduling algorithm**, and an **optimistic variant** that further improves concurrency.

- **OS1: Lock Acquisition Order $\rightarrow$ Execution Order**
  If transaction t_j acquires a conflicting lock $ql_j(x)$ after t_i's lock $pl_i(x)$:

$$pl_i(x) <_s ql_j(x) \Rightarrow p_i(x) <_s q_j(x)$$

- **OS2: Lock Release Rule**
  To preserve order dependencies, t_j (dependent on t_i) cannot release any locks while on hold. This avoids cycles in the precedence graph.

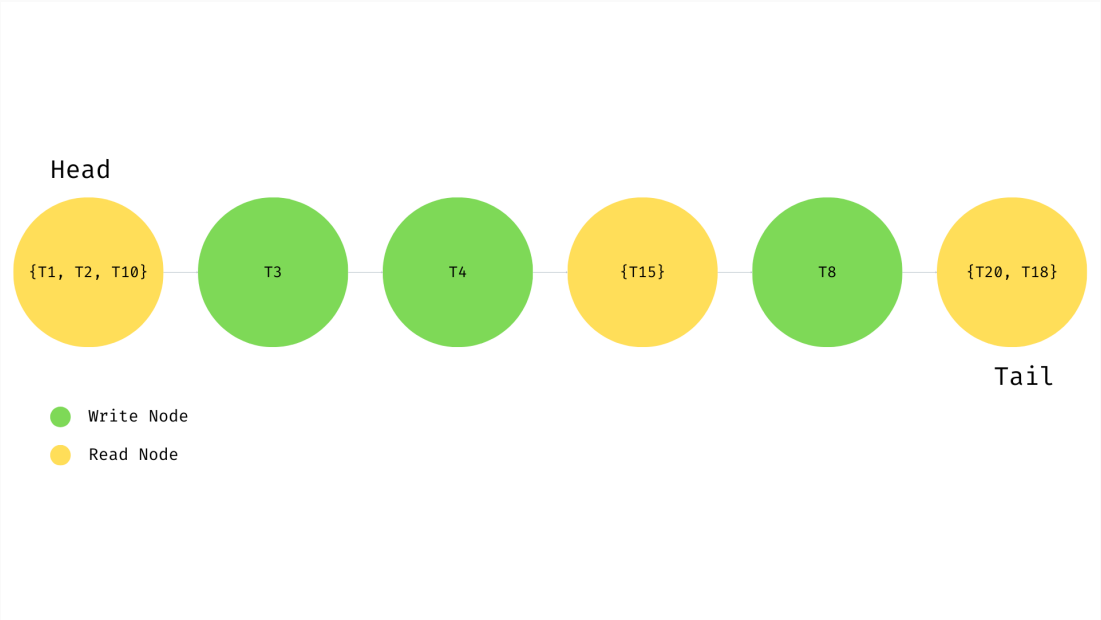
We will see the different high-level ideas for both the implementations.

High level Idea

In O2PL (both pessimistic and optimistic), preserving the order of dependencies is crucial. Both approaches must ensure the **OCSR (Ordered Conflict-Serializable) property**.

Property	Pessimistic O2PL	Optimistic O2PL
Overview	Transactions acquire locks before operations. Internally implemented using a special linked list. The WFG is consulted at every step.	Transactions proceed without acquiring locks, but track dependencies to ensure correctness.
Ordering	OCSR is ensured using the node list structure.	OCSR is ensured using the dependency sets.
Correctness	WFG ensures serializability by validating conflicts during execution.	WFG ensures serializability by validating conflicts post-execution.
Design	Frequent WFG access increases overhead, but early unlock assumptions reduce waiting.	No frequent WFG access or node list, but waiting on dependency sets can introduce delays.

List for (pessimistic) O2PL



Notice that the read node has a set of transactions, whereas the write node only has a transaction. All compatible operations will be grouped in the set. And whenever a conflicting operation is encountered, a new node is added to the queue.

Data Structures Required for Pessimistic O2PL Implementation

Data Structures Required

```
1   class Node {
2       Node *next;
3       int type;
4   }
5
6   // Inherited from base class
7   class rnode extends Node {
8       set<Transaction> tlist;
9       type = 0; // read node
10  }
11
12  class wnode extends Node {
13      Transaction txn;
14      type = 1; // write node
15  }
```

Data Structures Required

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

    // Methods
    addReadNode(Transaction txn);
    addWriteNode(Transaction txn);
    deleteReadNode(Transaction txn);
    deleteWriteNode(Transaction txn);
    getHead();
    getTail();
}
```

Data Structures Required

```
1   class DataItem {
2       int value;
3       NodeList* nodeList;
4       set<int> readList;
5       set<int> writeList;
6       mutex lock;
7
8       addRead(Transaction t);
9       addWrite(Transaction t);
10      deleteRead(Transaction t);
11      deleteWrite(Transaction t);
12  }
```

Data Structures Required

```
class Transaction {
    int tid;
    int threadId;
    status; // init ongoing
}
```

Additionally, we will also declare conditional variables for each thread.

```
atomic<int> conditinals[num_threads];
```

These will later be changed to conditional variables to avoid spinning on atomic variables. These will be accessed by the list data item to notify that the transaction which a thread is executing has become head node and that it can now execute.

Data Structures Required

```
class WaitsForGraph {
    set<int> vertices; // = transactions
    vector<vector<int>> adjacencyList; // = conflicts

    // Methods
    addOperation(int transId, int ind, int op);
    // transactionId; dataItem; read=0, write=1
    detectCycle();
    garbageCollect();
}
```

Scheduler Algorithms for Pessimistic O2PL

Read operations - O2PL

```
bool read(transaction t, int index, int *localVal) {
    bool permission = WFG.addOperations(t, index, 0); // 0 ->
        read op
    if(permission == 0) {
        t.status = aborted;
        return false;
    }
    shared[index].addRead(t);
    while(conditional[t->threadId] != true) { ; // spinning,
        will be changed }
    localVal = shared[index].value;
    shared[index].deleteRead(t);
    return true;
}
```

```
void addRead(Transaction t) {
    this->lock.lock(); // Lock data item
    this->nodeList->addReadNode(t);
    this->lock.unlock();
}
```

```
void addReadNode(Transaction t) {
    if(this->nodeList->getHead() == nullptr) {
        Node *r = new rnode();
        head = r;
        tail = r;
    }
    if(this->nodeList->getTail().type == 0) {
        tail->tlist.insert(t);
        if(this->nodeList->getTail() == this->nodeList->getHead
            ()) {
            conditional[t->threadId] = true;
        }
    } else {
        Node *r = new rnode();
        r.tlist.insert(t);
        tail->next = r;
        tail = r;
    }
    this.readList.add(t->tid)
}
```

```
void deleteRead(Transaction t) {
    this->lock.lock(); // Lock data item
    this.nodeList.deleteReadNode(t);
    this->lock.unlock();
}
```

```
void deleteReadNode(Transaction t) {
    // If it is here, it means it is head. It will be in the set
         of the tlist of head.
    auto iter = this.head.tlist->find(t);
    this.head.tlist.erase(t);

    if(head.tlist.size() == 0) {
        prev = head;
        head = head->next;
        delete prev;

        // Definitely next should be write node
        conditional[head.transaction->threadId] = true;
    }
}
```

Write Operations - O2PL

```
1   bool write(transaction t, int index, int localVal) {
2       bool permission = WFG.addOperations(t, index, 1); // 1
            -> write op
3       if (permission == 0) {
4           t.status = aborted;
5           return false;
6       }
7       shared[index].addWrite(t);
8       while(conditional[t->threadId] != true) { ; }
9       shared[index].value = localVal;
10      shared[index].deleteWrite(t);
11      return true;
12  }
```

```
1   void addWrite(Transaction t) {
2       this->lock.lock(); // Lock data item
3       this->nodeList->addWriteNode(t);
4       this->lock.unlock();
5   }
```

```
void addWriteNode(Transaction t) {
    if(this->nodeList->getHead() == nullptr) {
        Node *w = new wnode();
        head = w;
        tail = w;
        conditional[t->threadId] = true;
    } else {
        Node *w = new wnode();
        w.txn = t;
        tail->next = w;
        tail = w;
    }

    this.writeList.add(t->tid)
}
```

```
void deleteWrite(Transaction t) {
    this->lock.lock(); // Lock data item
    this.nodeList.deleteWriteNode(t);
    this->lock.unlock();
}
```

```
void deleteWriteNode(Transaction t) {
    // If it is here, it means it is head.
    prev = head;
    head = head->next;
    delete prev;

    // Next might be read/write nodes
    // read -> iterate through all threadIds of transactions and
         signal
    // If write node, just signal the transaction of the node
    // conditional[head.transaction->threadId] = true;
}
```

Commit Operation - O2PL

```
1  TransactionStatus tryCommit(Transaction t) {
2      if (t.status == aborted) {
3          return aborted;
4      }
5      return committed;
6  }
```

Optimistic O2PL

Data Structures required for Optimistic O2PL

The class architecture of the optimistic O2PL protocol is very similar to 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.

Read Operations - Optimistic O2PL

```
1  bool read((Transaction t, int index, int *localVal) {
2      bool permission = WFG.addOperations(t, index, 0); // 0 ->
           read op
3      if (permission == 0) {
4          t.status = aborted;
5          return false;
6      }
7      localVal = shared[index].value;
8      shared[index].readList.insert(t.transactionId);
9      // Adding the transactions that previously wrote to this
           item as dependencies
10     for (tid in shared[index].writeList) {
11         if (tid != t.transactionId) {
12             t.dependencySet.insert(tid);
13         }
14     }
15     return true;
16 }
```

Write Operation - Optimistic O2PL

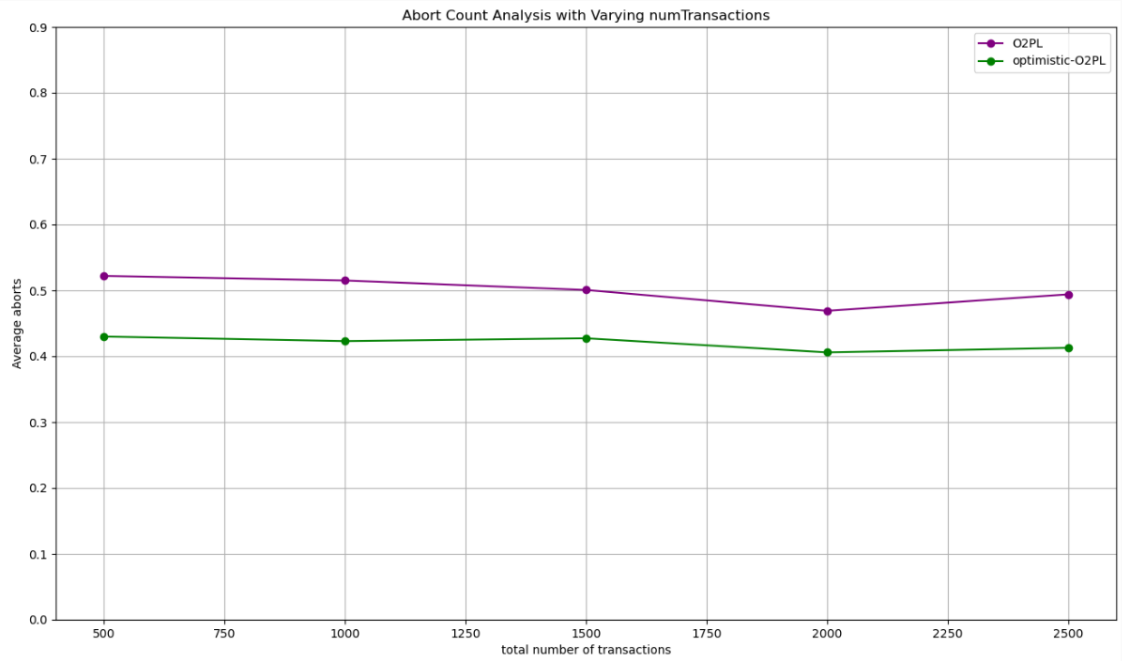
```
bool write(Transaction t, int index, int localVal) {
    t.localWrites[index] = localVal;
    // Adding t to the writeList of the data item
    shared[index].writeList.insert(t.transactionId);
    // Updating DependencySet of t
    for (tid in shared[index].readList) {
        if (tid != t.transactionId) {
            t.dependencySet.insert(tid);
        }
    }
    for (tid in shared[index].weriteList) {
        if (tid != t.transactionId) {
            t.dependencySet.insert(tid);
        }
    }
}
```

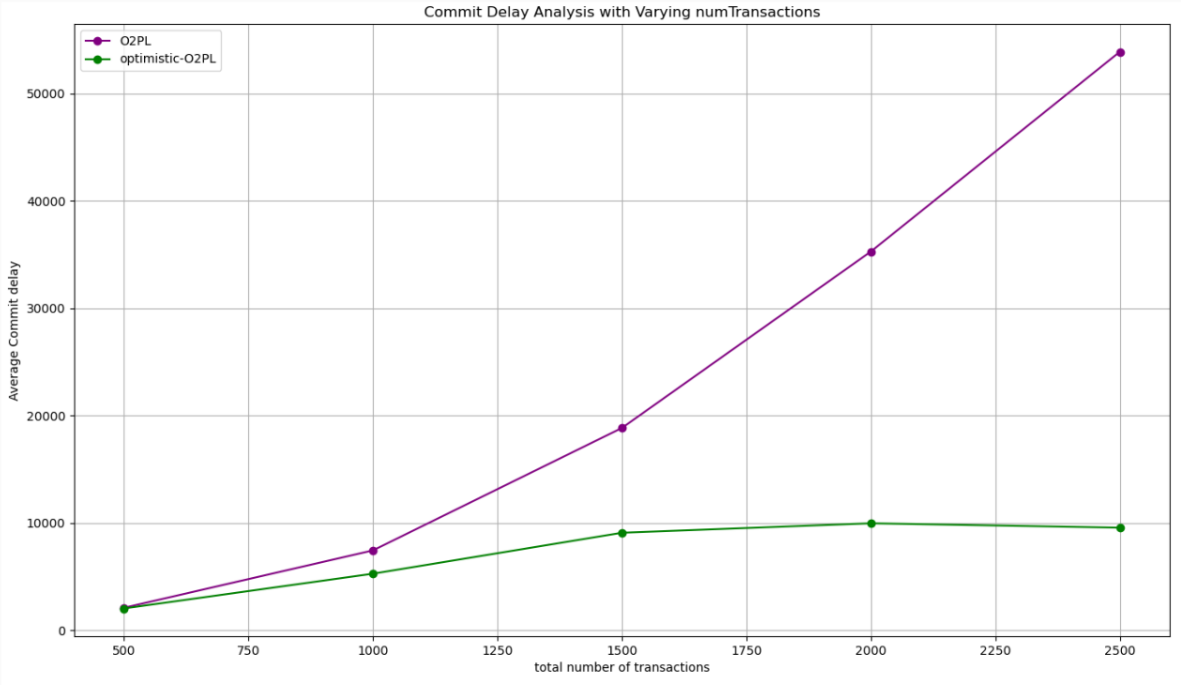
Commit Operation - Optimistic O2PL

```
bool tryCommit(Transaction t) {
    for ([index, value] in t.localWrites) {
        bool perm = WFG.addWriteOperation(t, index, 1);
        if (!perm) {
            t.status = aborted;
            WFG.deleteNode(t.transactionId);
            // delete t from writeList of items it wrote to
            finishedTransactions.insert(t.transactionId);
            return aborted;
        }
    }
    // Wait until all dependencies are committed
    while(true) {
        std::set<int> remaining = std::set_difference(
            dependencySet, finishedTransactions);
        if (remaining.empty()) break;
    }
    for ([index, value] in t.localWrites) {
        shared[index].value = value;
    }
    finishedTransactions.insert(t.transactionId);
    return committed;
}
```

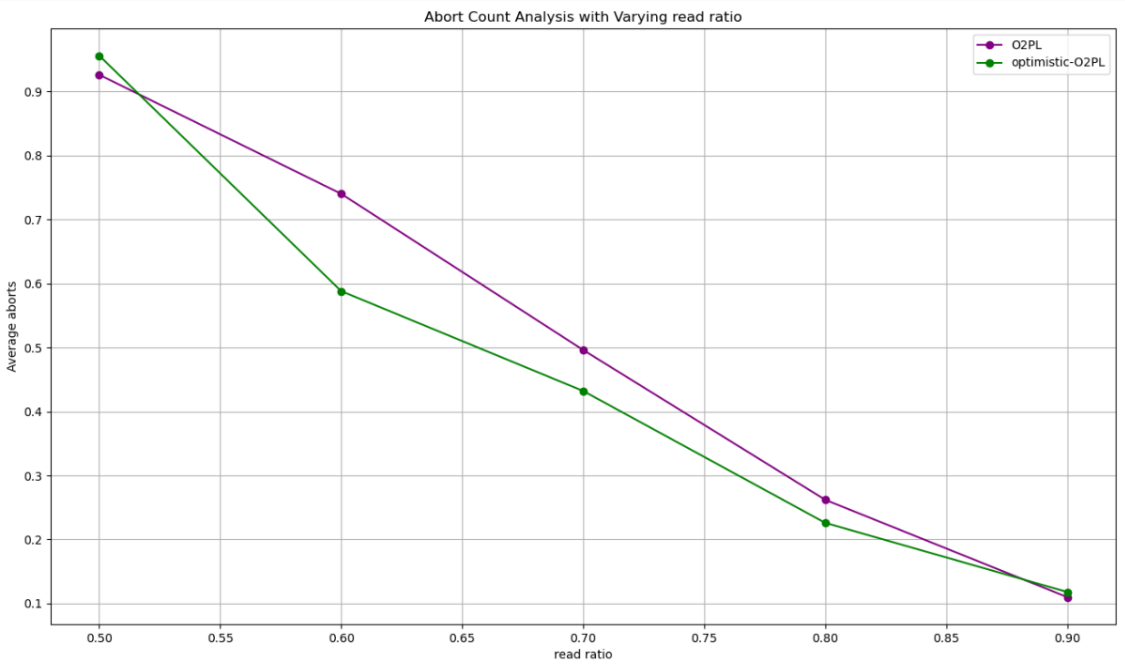
Experimental Results

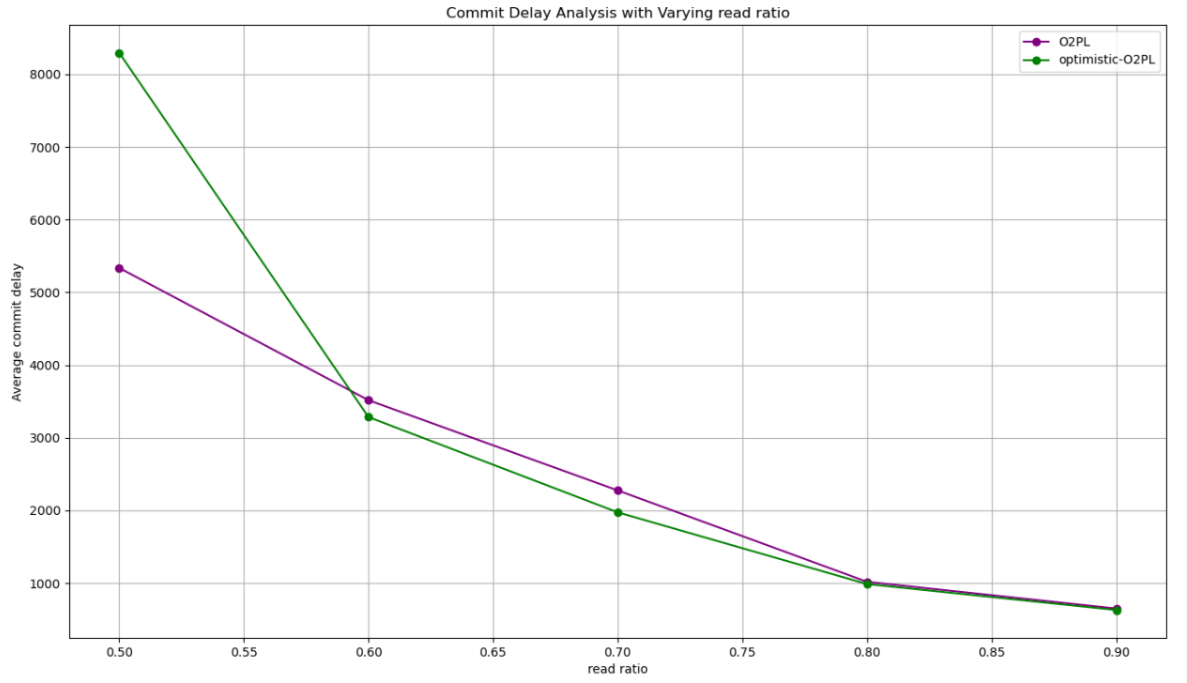
Experiment 1



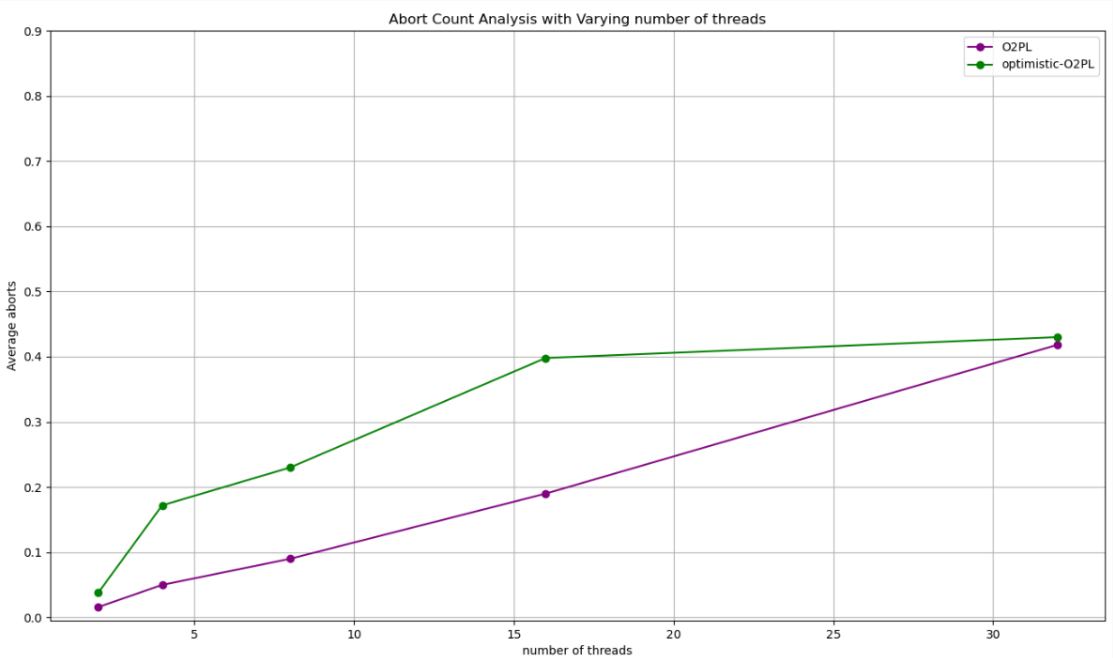


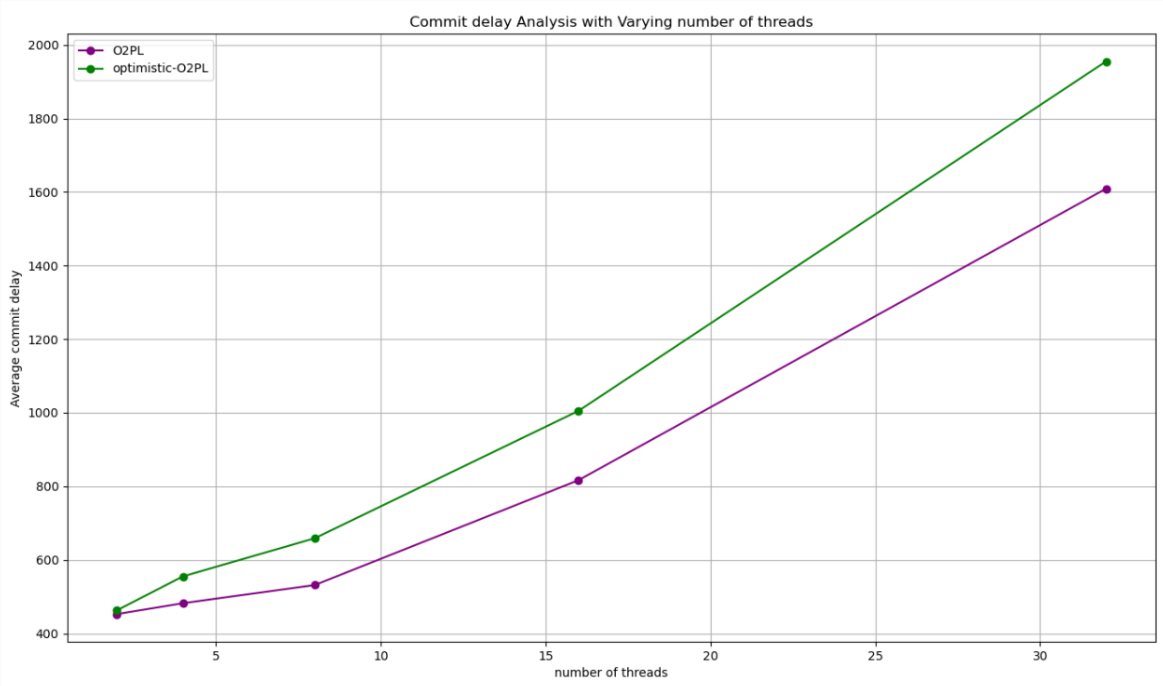
Experiment 2



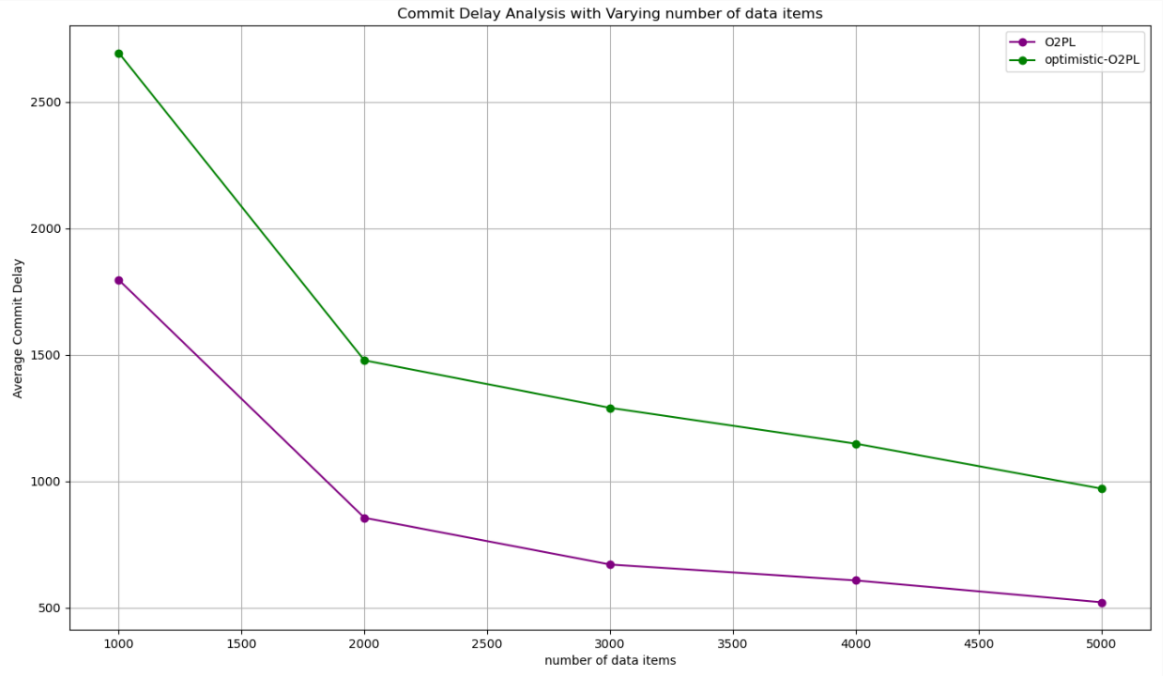


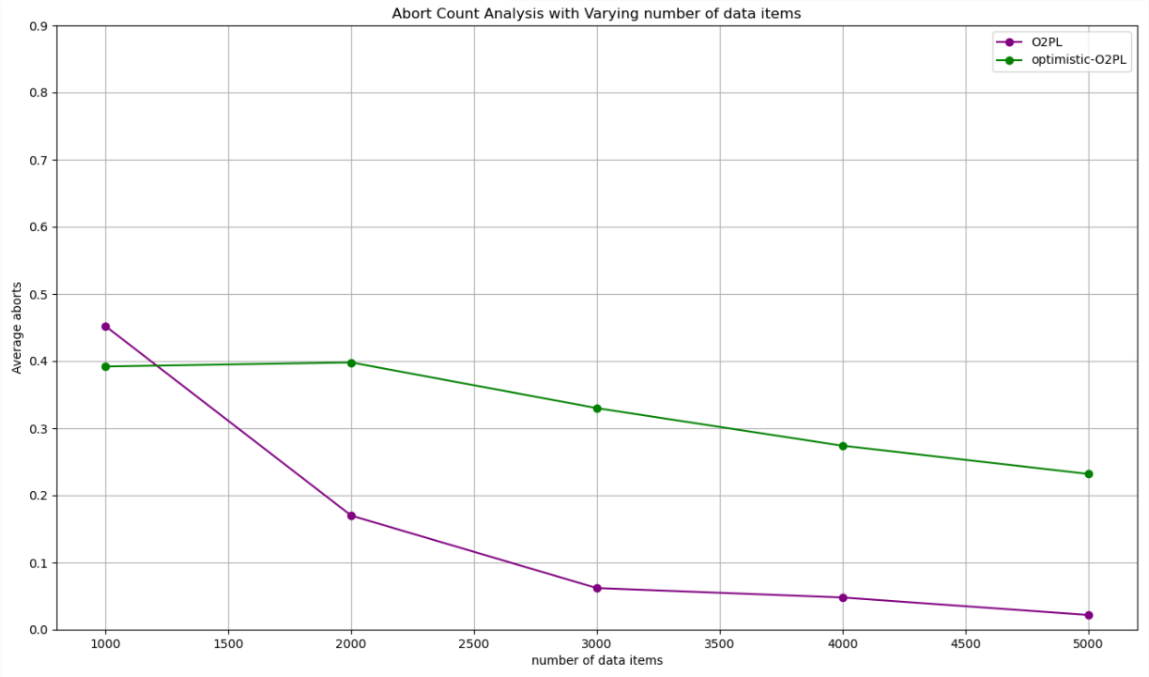
Experiment 3





Experiment 4





TO-DO

- Consider garbage collection for the wait-for graph to prevent unbounded growth.
- Explore lock-free implementations for data item lists to minimize contention.
- Exploring concurrent graph algorithms for implementing WFG.