

Turnitin Originality Report

Processed on: 06-May-2025 13:28 IST

ID: 2667957140

Word Count: 1985

Submitted: 1

ProjectReport-
CS21BTECH11022.pdf By Tushita
Sharva

Similarity Index

1%

Similarity by Source

Internet Sources:	0%
Publications:	1%
Student Papers:	0%

1% match (Seog Park. "Alternative correctness criteria for multiversion concurrency control and its applications in advanced database systems", Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat No 98EX130) DEXA-98, 1998)

[Seog Park. "Alternative correctness criteria for multiversion concurrency control and its applications in advanced database systems", Proceedings Ninth International Workshop on Database and Expert Systems Applications \(Cat No 98EX130\) DEXA-98, 1998](#)

< 1% match (Amr El Abbadi. "Locks with constrained sharing (extended abstract)", Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS 90 PODS 90, 1990)

[Amr El Abbadi. "Locks with constrained sharing \(extended abstract\)", Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS 90 PODS 90, 1990](#)

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. 1 class Node { 2 public: 3 Node *next; 4 OperationType type; 5 6 // Variables for signalling waiting threads 7 std::mutex mtx; 8 std::condition_variable cv; 9 bool isAtHead; 10 11 Node(OperationType t) : type(t) { next = nullptr; isAtHead = false; } 12 }; 13 14 class ReadNode : public Node { 15 public: 16 std::set<list> tlist; 17 }; 18 19 class WriteNode : public Node { 20 public: 21 Transaction *transaction; 22 }; 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 Dataltem class, allowing it to interact with the locking mechanism without knowing its implementation specifics. This design promotes decoupling.

```

1 class NodeList {
2 public:
3 Node *head;
4 Node *tail;
5
6 NodeList();
7 ~NodeList();
8
9 Node* addReadNode(Transaction *t);
10 Node* addWriteNode(Transaction *t);
11 void deleteReadNode(Transaction *t);
12 void deleteWriteNode(Transaction *t);
13 };

```

2.1.4 Dataltem Class

As discussed, each Dataltem 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.

```

1 class Dataltem {
2 public:
3 int value;
4 NodeList *nodeList;
5 std::set readList; // will be used by WFG graph
6 std::set writeList; // will be used by WFG graph
7 std::mutex datalock;
8
9 // Functions to manipulate the nodeList
10 Node* addRead(Transaction *t); void deleteRead(Transaction *t);
11 Node* addWrite(Transaction *t); void deleteWrite(Transaction *t);
12 };

```

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 Dataltem 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.

```

1 class Scheduler {
2 public:
3 WaitsForGraph* G;
4 std::mutex graphLock;
5 std::vector< shared > shared;
6 std::atomic counter;
7
8 void init(int m);
9 Transaction* begin_trans(int threadID);
10 bool read(Transaction* t, int index, int &localVal);
11 bool write(Transaction* t, int index, int localVal);
12 TransactionStatus tryCommit(Transaction* t);
13 };

```

2.2 Implementation Method

- The main function starts multiple threads. Each thread asks the scheduler to create a new transaction, which it will use to perform operations.
- Transactions begin and send read/write requests to the scheduler.
- For each request, the scheduler checks with the WaitsForGraph to see if the operation would cause a deadlock.
 - 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.
 - If not safe, the transaction is aborted.
- The system collects metrics like commit times and abort counts during execution.

2.3 Proof Sketch: $\text{Gen}(\text{O2PL}) \subseteq \text{OCSR}$

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

Locking Discipline:

- Each Dataltem 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 $T_i \rightarrow T_j$ 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 ($T_j \rightarrow T_i$ where $j > i$) are possible in the conflict graph. Hence, the protocol guarantees that the conflict graph is a DAG with edges respecting $<$, 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

- The main function spawns multiple threads. Each thread requests a new transaction from the scheduler, which it uses to issue read/write operations.
- 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: $\text{Gen}(\text{Optimistic O2PL}) \subseteq \text{CSR}$ - 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 $\text{SG}(H)$, where: - Each node corresponds to a committed transaction. - An edge $T_i \rightarrow T_j$ 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 $\text{SG}(H)$, the resulting serialization graph is acyclic. Conclusion: Since every history H generated by the protocol has an acyclic serialization graph, we conclude that: $\text{Gen}(\text{Optimistic O2PL}) \subseteq \text{CSR}$ 3.5 Proof Sketch: $\text{Optimistic O2PL} \subseteq \text{OCSR}$ Let each transaction T_i be assigned a unique, increasing transaction ID at start time. We define a total order $<$ such that $T_i < T_j$ 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 $T_i \rightarrow T_j$, 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 $T_i \rightarrow T_j$ satisfy $j < i$. - Hence, all edges in the conflict graph respect the total order $<$. 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 Results4.1 Experiment 1: Varying number of transactions (16 threads) Explanation Optimistic O2PL aborts fewer transactions because it checks conflicts only at commit time, therefore unnecessary 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. 1 2 3 4 5 6 7 8 9

