Comparison of Timestamp based Optimistic Concurrency Algorithms (Tic Toc and TOCC)

Concurrency Control in Transactional Systems: Spring 2023

K B Vijay Varma
AI20BTECH11012

Pavan Valavala
CH20BTECH11040

I. Introduction

We have implemented two Timestamp based Optimistic Concurrency Control Algorithms namely Tic Toc and Timestamp based OCC (TOCC).

There are two separate files for each of the Algorithms namely tic-toc.cpp, tocc.cpp which when run creates two log files tic-toc-log.txt, tocc-log.txt.

They contain the logs of the Transaction about their reads, writes, aborts, commits etc.

II. Instructions for Running the Codes

- Create a Text file named "inp-params.txt" containing the following values separated by space.
  numThreads m numTrans constVal lambda envNum
- Even though m is present above, m is not input from the above txt file, it is defined in the code as m = 10 (because we have to initialize the Shared Memory).
- The envNum tells the program whether to choose environment 1 or 2 for simulation. Details about Environments are given later.
- Type the following command to run the file tic-toc.cpp which creates an executable file named tic-toc
  "g++ -std=c++20 -g -w tic-toc.cpp -o tic-toc"
- Then run the executable file as follows,
  "./tic-toc"
- This will run the code and we can see the Input Params printed, Average Commit Delay and Average Abort Count, Time of Running the code in the standard output.
- We can also see there is a file created named TIC-TOC-log.txt which contains the logs of the Transactions.
- Similarly for running the Other Algorithm, replace tic-toc with tocc.

III. Paper 1: TicToc: Time Traveling Optimistic Concurrency Control

- In this paper, the authors introduce TicToc, a novel concurrency control method that entirely eliminates the timestamp allocation bottleneck and delivers higher concurrency than cutting-edge T/O schemes.
- The main innovation of TicToc is a method they call Data-Driven Timestamp Management: rather than allocating timestamps to each transaction independently of the data it accesses, TicToc embeds the necessary timestamp data in each of the tuple of the Transactions. This allows each transaction to compute a valid commit timestamp after it has run, just before it commits.
- This approach has two benefits:
  - First benefit is, each transaction determines its timestamp by analyzing the metadata(timestamps) linked to each tuple it reads or writes. The Timestamp allocation bottleneck is eliminated since there is no Centralized Timestamp Allocator and no communication between concurrent transactions accessing different pieces of data.
  - Second benefit is, TicToc establishes a logical-time order that maintains serializability even among transactions that overlap in physical time and would result in aborts in other T/O-based protocols by calculating timestamps lazily at commit time.
- To put it simply, TicToc permits commit timestamps to advance in time to reveal more concurrency than existing techniques without breaking serializability.

A. Implementation Details

- TicToc is an Optimistic Concurrency Control Algorithm. So we used Write Sets and Read Sets to store the Reads and Writes of all transactions. The DataObject contains Read Timestamps, Write Timestamps which are needed for calculating the Commit timestamp of a transaction.
- The Transaction Object contains status, r_set, w_set, r_set_only and status variable to indicate whether it is alive or committed or aborted.
- We have used Locks for locking the data object and then do read and write operations as well as separate locks for Logging.
- And also, for maintaining the Critical Section (Validation and Write Phase), we locked all the data items as mentioned in the Paper.
- We implemented the following phases as per in the Paper:

– **Read Phase**

Algorithm 1: Read Phase

```
Data: read set RS, tuple t
r = RS.get_new_entry()
r.tuple = t
# Atomically load wts, rts, and value
< r.value = t.value, r.wts = t.wts, r.rts = t.rts >
```

– **Validation Phase**

Algorithm 2: Validation Phase

```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in WS ∪ RS do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        commit_ts = max(commit_ts, e.wts)
    end
end
#  Step 3 – Validate the Read Set
for r in RS do
    if r.rts < commit_ts then
        # Begin atomic section
        if r.wts ≠ r.tuple.wts or (r.tuple.rts ≤ commit_ts and
        isLocked(r.tuple) and r.tuple not in W) then
            abort()
        else
            r.tuple.rts = max(commit_ts, r.tuple.rts)
        end
        # End atomic section
    end
end
```

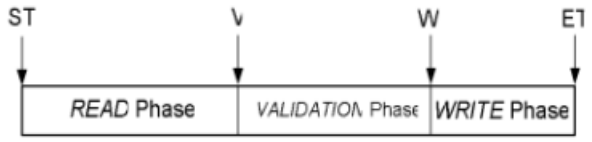
– **Write Phase**

Algorithm 3: Write Phase

```
Data: write set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuple.value, w.value)
    w.tuple.wts = w.tuple.rts = commit_ts
    unlock(w.tuple)
end
```

IV. Paper 2: TOCC: Timestamp Based Optimistic Concurrency Control

- In this paper, the authors introduce TOCC, a Timestamp based Optimistic Concurrency Control method that is better than the Original Optimistic Concurrency Control methods.
- Every transaction will go through three phases, Read, Validation, and Write, same like the Traditional Optimistic concurrency control protocol. Transactions only read data items and update (pre-write) data items in a private work space during the read phase.
- If validation is successful, the database is really updated. At various phases of the transaction, as illustrated in the following diagram IV, they assign some symbols to each transaction.



- Each transaction will read from and/or write to some data items while it is being executed. These reads and writes are enrolled in the ReadSet and WriteSet sets, respectively.
- Tuples of data objects conjugated with the time the data item was accessed make up the members of ReadSet and WriteSet. Specifically, ReadSet and WriteSet shall take the form
(αm, tn), (αp, tq), ... — data item αm was accessed at time tn and so on .
- The further layer of inspection will be applied when transactions have been validated. The transaction is validated and verified if condition 1 is true. In the absence of that, the validity of conditions 2 and 3 should be examined. The conditions are:
  - Is the validating transaction in serial position with conflicting transaction?
  - Is there any write-write conflict from validating to conflicting transactions?
  - Is there any read-write conflict from validating to conflicting transactions?
- They set our conflict checking precisely such that validating transactions should not be aborted for non-serious conflicts.
- They have a Correctness Proof for TOCC which is based on the fact that an Acyclic Serialization Graph produces only schedules which are in CSR.
- They made sure that their Algorithms produced Serialization Graphs that are Acyclic. Ultimately their Algorithm TOCC (Timestamp based Optimistic Concurrency Control) exhibits serializability.

A. *Implementation Details*

- TOCC is also an Optimistic Concurrency Control Algorithm. So we used Write Sets and Read Sets to store the Reads and Writes of all transactions.
- The DataObject contains only the value of the object.
- The Transaction Object contains t_id, start_time, valid_time, end_time, r_set, w_set, w_set_val and status variable to indicate whether it is alive or committed or aborted.
- We have used Locks for locking the read and write operations as well as separate locks for Logging.
- And also, for maintaining the Critical Section (Validation and Write Phase), we locked the tryCommit as mentioned in the Paper.
- We implemented the following phases as per in the Paper:
  - **Read Phase**
    - In the Read Phase, we stored all the Reads and Writes of the Transaction along with the corresponding Time stamps as mentioned in the paper in their respective private Read and Write Sets.
  - **Validation Phase**

```
1  if (Ti(ET) < Tj (ST))  Valid = TRUE
2  else{
3       Valid =TRUE
4       if ((WriteSet (Tj)∩WriteSet (Ti) ≠ Ø) and
5                                        (Ti(ET) > Tj(V)))
6            Valid = FALSE
7        else if (ReadSet(Tj)∩WriteSet (Ti) = β) {
8            for every element ε of β {
9                 if  (ε(Tj) <Ti(ET)) {
10                     Valid = FALSE
11                     Break
12                     }
13                }
14     }
15 }
```

  - **Write Phase**
    - On successful validation from the above phase, we wrote all the Writes into the Database (Shared Memory) by applying suitable locks so that the next translation reads only the Latest Version of data items.

V. Input Parameters

We took the values of n (numThreads) = 10,20,30,40,50,...,100, m (Number of Data Items) = 10, numTrans = 100, constVal = 100, lambda = 20, environment = 1,2 and plotted the Average Commit Delay as well as Average Abort Count.

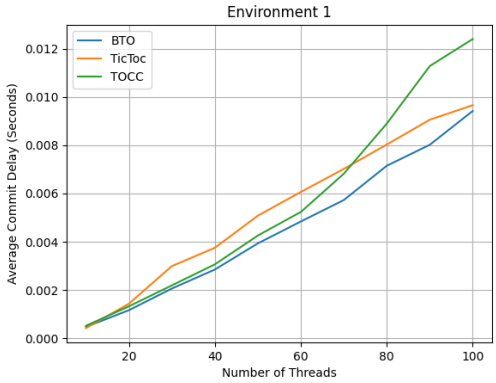
VI. Testing Environment

- We created two Environments for Testing the above two Algorithms along with BTO (Optimistic Implementation).
- The First Environment is a standard one used for the Programming Assignments. In this, each thread simulates numTrans and in each transaction, randIters Operations are performed. Also a Read Operation is always followed by a Write Operation in this environment and the write is the same as the read data index. The Thread restarts the Transaction if it aborts and runs until its numTrans are committed.
- The Second Environment is the Modified First Environment. In this, everything is the same as the first environment but the Write selects a random data index for writing rather than the read data index.

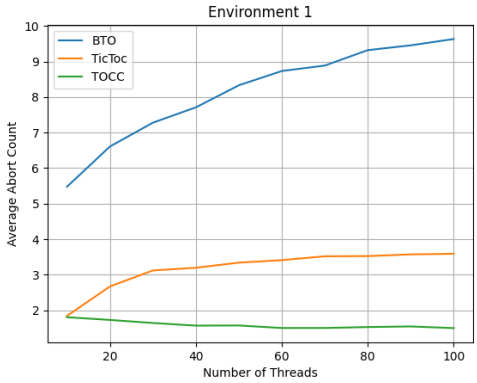
VII. Comparison Graphs

A. *Environment-1*

- **Graphs of BTO, TICTOC, TOCC for Average Commit Delay vs Number of Threads**

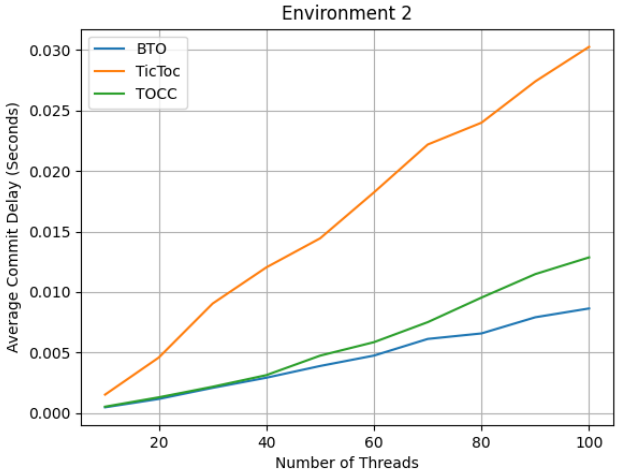


- **Graphs of BTO, TICTOC, TOCC for Average Abort Count vs Number of Threads**

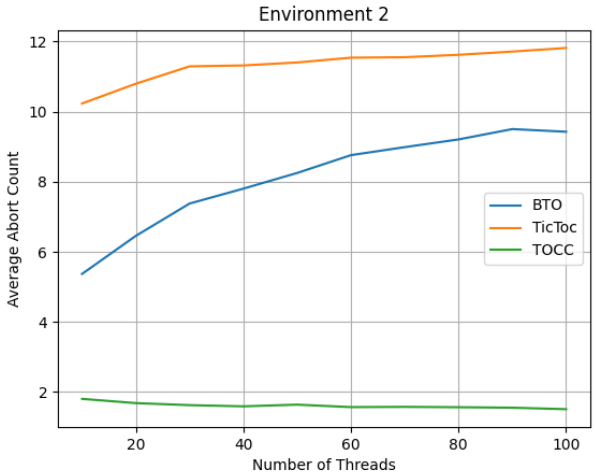


B. Environment-2

- **Graphs of BTO, TICTOC, TOCC for Average Commit Delay vs Number of Threads**



- **Graphs of BTO, TICTOC, TOCC for Average Abort Count vs Number of Threads**



VIII. Comparison of Performance of Graphs of BTO, TICTOC, TOCC

- Environment-1
  - From the Environment-1 1st graph, we can see that all the three Algorithms are performing equivalently and they are increasing in proportional to Number of threads. But BTO has less Average Commit Delay compared to others.
  - From the Environment-1 2nd graph, we can see that the New Algorithms TicToc, TOCC are performing far better than BTO as they have less abort count.
- Environment-2
  - From the Environment-2 1st graph, we can see that TicToc has more Average Commit Delay than the other two and of the other two, BTO is performing better.
  - From the Environment-2 2nd graph, we can see that TOCC has the Lowest Abort count followed by TicToc and then BTO.

IX. Conclusion

- The algorithms we developed outperform comparable Timestamp Based Optimistic Concurrency Control Algorithms that are presently in use, such as BTO in the sense of minimizing the Aborts.
- For managing concurrent access to shared resources, Timestamp-Based Optimistic Concurrency Control algorithms are a strong and well-known method.
- There are various varieties of these algorithms, each with its own advantages and disadvantages.
- When selecting a TOCC algorithm, it's important to take into account a number of important factors, such as the projected workload characteristics, the anticipated system contention level, and the level of precision and consistency needed by the application.
- Overall, TOCC algorithms are a popular option for many real-world systems because they provide a reasonable compromise between concurrency and accuracy.
- However, as with any concurrency control strategy, it's crucial to thoroughly assess the unique requirements of your system and select the proper algorithm in accordance with those needs.
- And likewise the True Performance of the above Algorithms can only be reached or known when employed in Real life systems.

References

[1] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, Srinivas Devadas, "TicToc: Time Traveling Optimistic Concurrency Control", 2016.

[2] Quazi Mamun, Hidenori Nakazato, "Timestamp Based Optimistic Concurrency Control", 2015.