Comparison of BTO, MVTO, MVTO-gc, k-MVTO

Name: Tushita Sharva
Roll Number: CS21BTECH11022

April 11, 2025

1 Problem Statement

This project implements and compares the scheduling algorithms: BTO, MVTO, k-MVTO, MVTO-gc. The performance is measured in terms of:

- The average number of times a transaction is aborted before committing (abort count).
- The average time taken for a transaction to commit (commit delay).

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.

2 Program Design

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:

- **id**: Unique transaction identifier.
- **status**: Transaction state (active, committed, aborted).
- **localSharedMemory**: Stores local modifications before committing.

2.1.3 DataItem Class

Encapsulates shared data items and tracks transactions accessing them. I implemented separate classes, one for BTO protocol, called `PrimitiveDataItem.h` and one for variants of MVTO protocol, called `DataItem.h`. `DataItem.h` uses another class `VersionedDataItem.h`, which has the details of specific version. Classes are as follows:

```
class DataItem { // For BTO
public:
    int value;
    int maxRead;
    int maxWrite;
    std::mutex datalock;

    DataItem() {
        value = 0;
        maxRead = INT_MIN;
        maxWrite = INT_MIN;
    }
};
```

```
class VersionedDataItem {
public:
    int version;
    int valueInThisVersion;
    std::set<int> readList;

    int getLargestInReadList();
};
```

```
class DataItem { // M
public:
    std::map<int, VersionedDataItem> versions;  // key = version
    std::mutex datalock;

    DataItem() { // Initialize with a 0th version }
    int getGreatestVersionLessThanMyId(int id);
};
```

3 Validation logics for the protocols

BTO:

- Whenever a transaction reads a data item, scheduler checks with the maxWrite of the data item and if it's time stamp is less, it is aborted.
- Whenever a tranction writes a data item, scheduler checks with both maxRead and maxWrite of the data item, and if it's time stamp is less and it hasn't written to it before, it is aborted.
- **Here, when a second read/write comes, I am not checking if it is validity, but directly accessing the local memory. This is because since we are going in an optimistic way and we are writing at the end of transaction, why don't we modify the local value and write the same at the end? It seems logical even because page model allows only one read and one write per data item per transaction.**

MVTO:

- When we read, we see if it is local shared memory, else we read from the smallest version less than ourselves. There won't be any aborts caused by a `read` in this algorithm.

- When we write, we see if the write is valid, and if not, we abort. If it is valid, we write to the local memory.
- When the transaction is trying to commit, we check if the transaction is ongoing. If it is not, we abort. If it is valid, we write to the shared memory, by adding a new version to each data item we accessed.

k-MVTO:

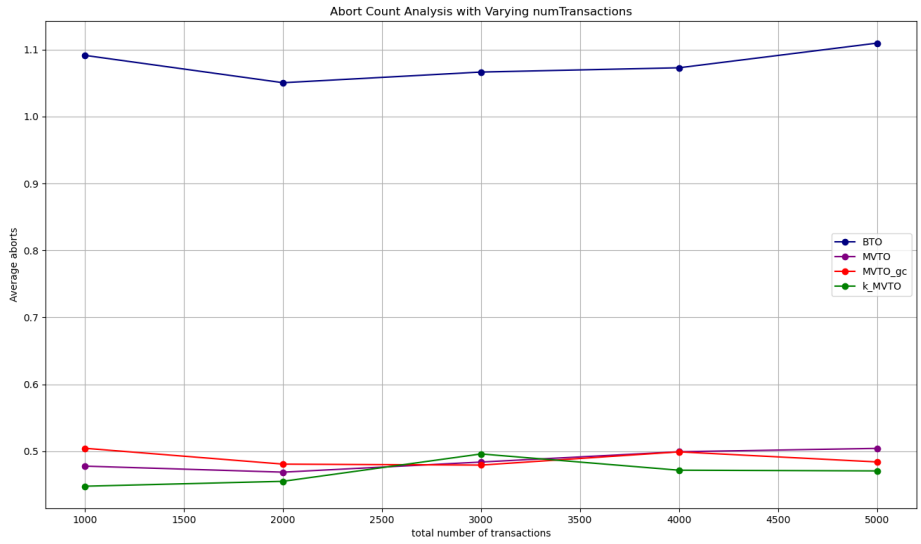
- When we read, we see if it is local shared memory, else we read from the smallest version less than ourselves. We abort the transaction if we cannot find a version less than ourselves.
- When we write, we see if the write is valid, and if not, we abort. If it is valid, we write to the local memory.
- When the transaction is trying to commit, we check if the transaction is ongoing. If it is not, we abort. If it is valid, we write to the shared memory, by adding a new version to each data item we accessed. If the number of versions exceeds k, we remove the oldest version.

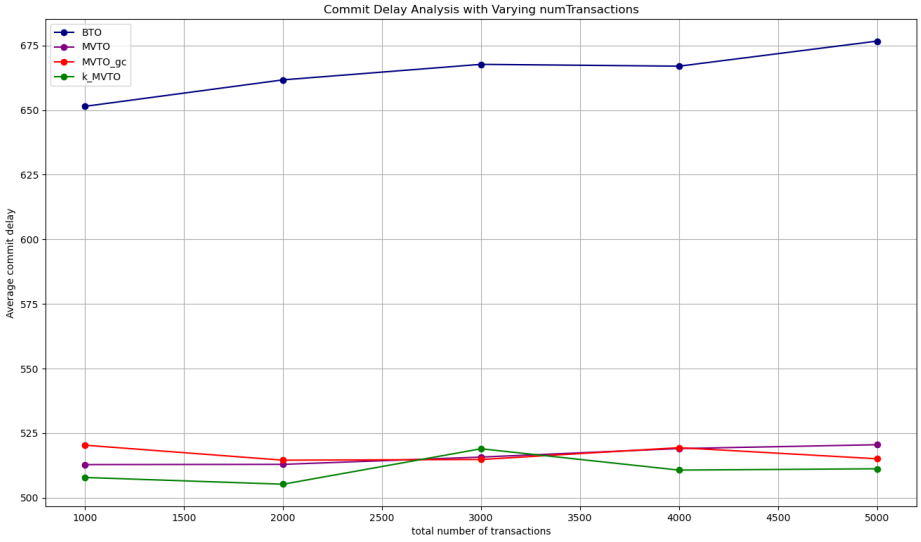
MVTO-gc:

- When we read, we see if it is local shared memory, else we read from the smallest version less than ourselves.
- When we write, we see if the write is valid, and if not, we abort. If it is valid, we write to the local memory.
- When the transaction is trying to commit, we check if the transaction is ongoing. If it is not, we abort. If it is valid, we write to the shared memory, by adding a new version to each data item we accessed.
- Whenever a transaction commits, it checks the smallest live transaction from the live set. We also keep a variable smallest live transaction, initialised with 0. Whenver a transaction commits, it checks if the smallest live transaction in the live set is same as the one that was seen by a previously committed transaction. If it changed, **only then** the transaction performs garbage collection. And whenever a transaction observes the smallest live transaction changed, it updates the variable.
- If we donot keep an extra variable, we will access every data item and find that there is no version less than the smallest live transaction and return without performing any. Even though this is O(1) time, this time will be multiplied with number of shared variables, hence saves time.

4 Experimental Results

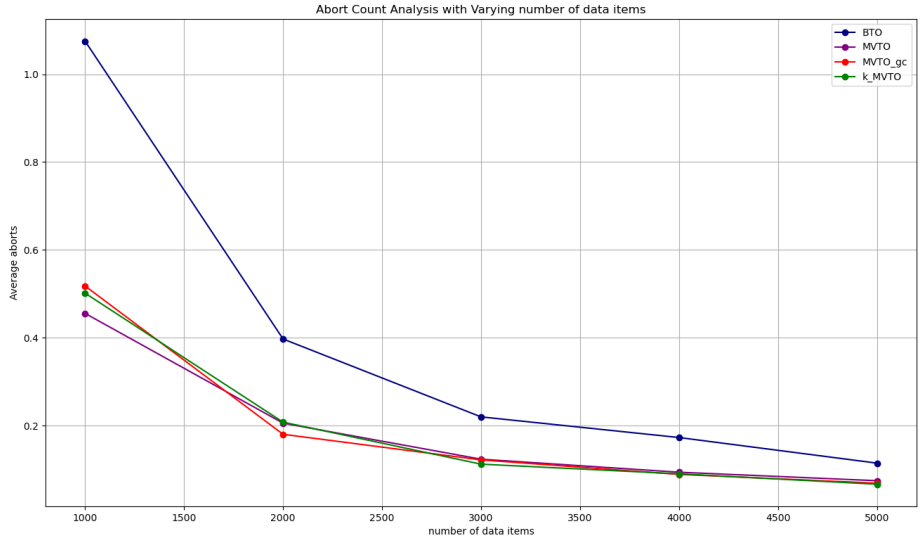
4.1 Experiment 1: Varying number of transactions —— 16 threads

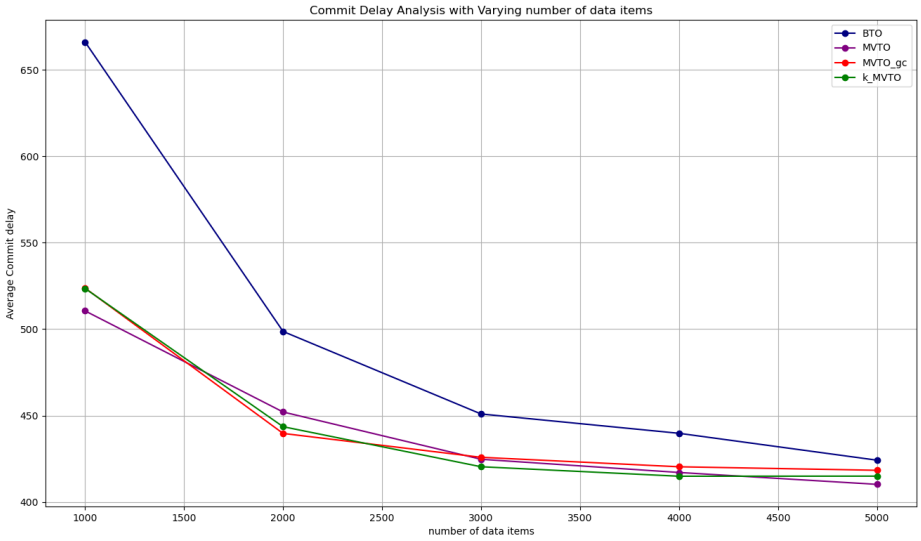




Explanation We can see that BTO performs the worst, because of more number of aborts the algoroithm causes. The performance of all three variants of MVTO are comparable, with k-MVTO outperforming others most of the times! This is a bit surprising because k-mvto is the one which has fixed number of versions and also causes aborts even in read-only transactions. However, this isn't observed here. This can be attributed to the randomness caused by readRatio.

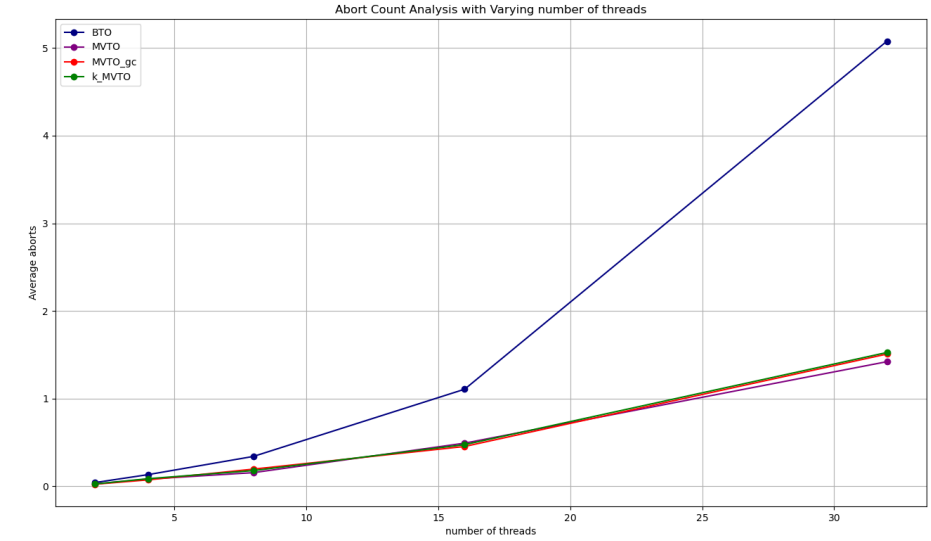
4.2 Experiment 2: Varying number of shared variables

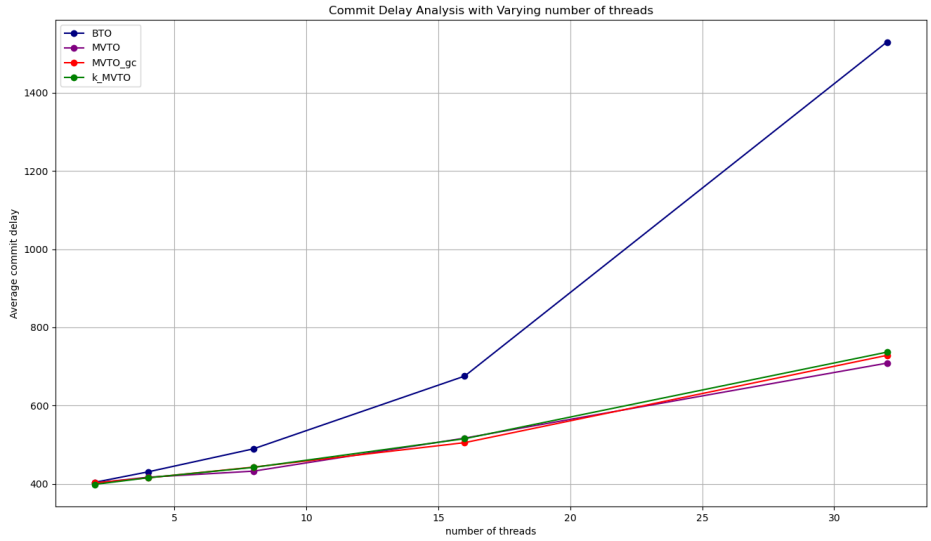




Explanation By increasing the number of data items, we can see that the abort count and commit delays are rapidly decreasing. It is because when there are more data items, and when we are sampling the data items, we are decreasing the likelihood of same data item being picked by concurrent transactions, which reduce conflicts and thus reduce the abort counts and commit delays. We also see that the performance of BTO is still the worst, but the difference between the three MVTO variants is not that much.

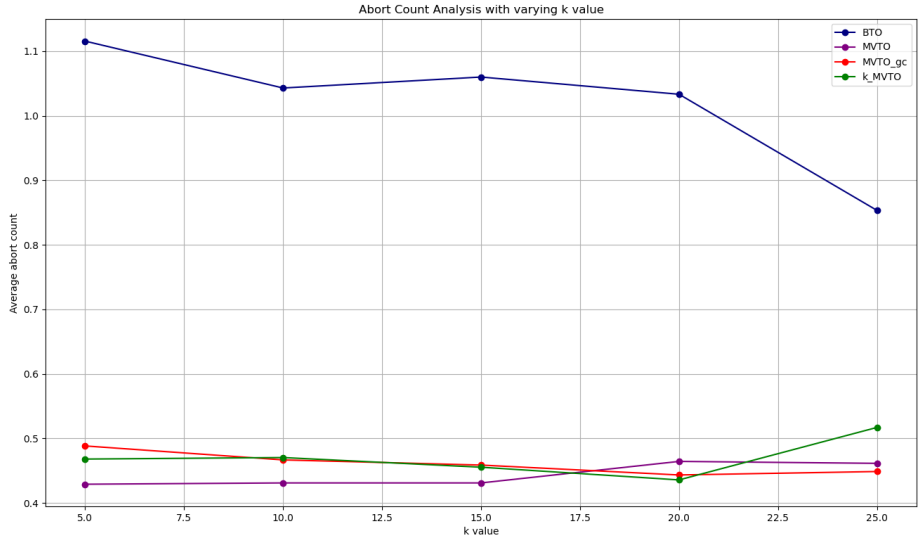
4.3 Experiment 3: Varying number of threads

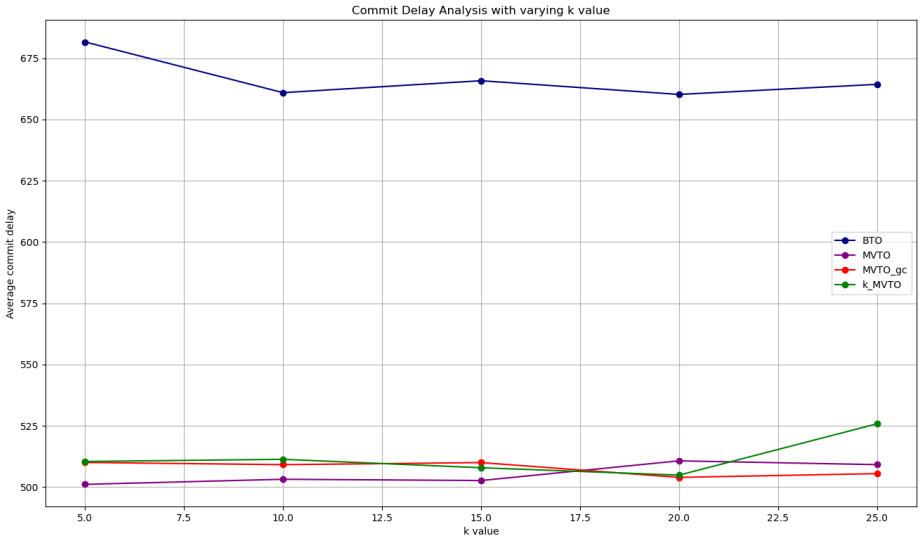




Explanation Usually we see that increasing threads improves performance because of a greater degree of concurrency. But in this case, by increasing the threads, we are increasing the number of concurrent transactions, not dividing work but we are creating more competition. When we have two threads, affectively only 2 transactions are running paralelly - chances they conflict are low, number of items to check are low and so on. Hence the result. If we were to measure the overall time taken for the programs to execute instead of abort count or commit delays, the result would've been different. Also, we can see that BTO doesnot scale well with the increasing number of threads, which is expected.

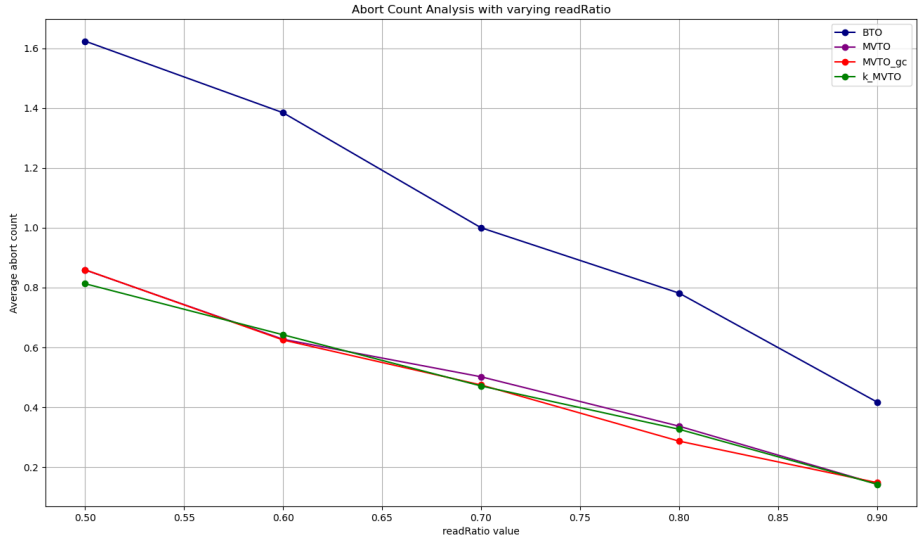
4.4 Experiment 4: Varying value of k

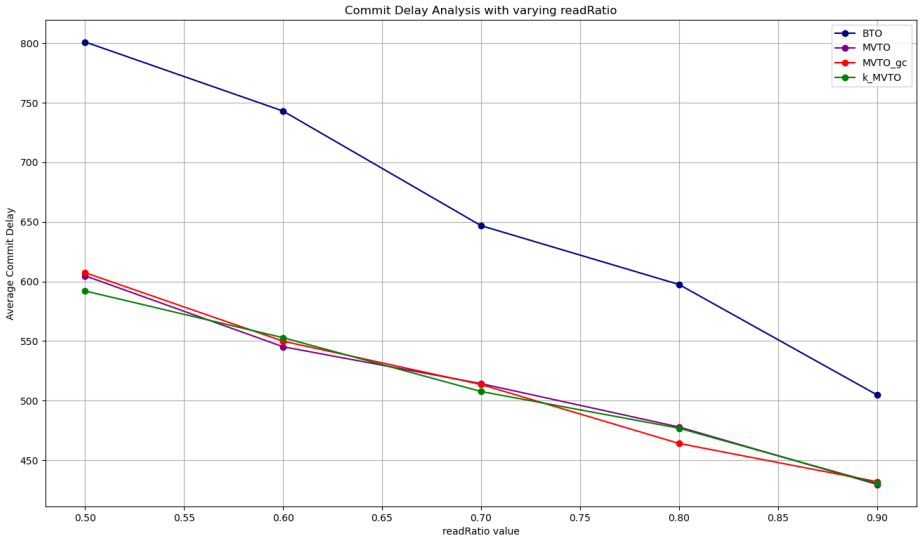




Explanation Specific to k-mvto, we see that it shows improvement as k increases, and we also see that it starts performing equal and even better than the mvto and mvto-gc at some values of k. And as usual, BTO is performing nowhere near, for any value of k of k-mvto.

4.5 Experiment 5: Varying value of readRatio





Explanation We see that with increased read ratio, the abort count and commit delay of all the algorithms are decreasing. This is expected because this significantly decreases the number of conflicts by avoiding read-write conflicts and write-write conflicts. We also see that mvto-gc is being able to take the best advantage of the increased read-ratio, which is expected because it avoids the iteration time for searching the right version, as well as avoids the aborts caused by not having the version needed.

5 Future Work

We can further improve the algorithm by using **binary search for finding the right version**, by keeping the sorted versions. This would exponentially decrease search time, and we can expect better results.