Concurrency Control in Transactional Systems: Autumn 2019

Programming Assignment 1: Implementing BOCC and FOCC algorithms

Submission Date: 16th October 2019, 9:00 pm

Goal: The goal of this assignment is to implement BOCC and FOCC algorithms studied in the class. Implement both these locking algorithms in C++.

Details. As shown in the book, you have to implement both the optimistic concurrency algorithms **in C++**: BOCC & FOCC. Since, you are using optimistic concurrency control approach, all writes become visible only after commit. Thus on abort of a transaction, no rollback is necessary as none of the writes of the transactions will ever be visible.

You have to implement the following methods for both the algorithms:

- $begin\_trans()$: It begins a transactions and returns a unique transaction id, say i
- $read(i, x, l)$: Transaction t_i reads data-item x into the local value l.
- $write(i, x, l)$: Transaction t_i writes to data-item x with local value of l.
- $tryC(i)$: Transaction t_i wants to commit. The return of this function is either a for abort or c for commit.

For each algorithm, please implement the two variants: (1) the validating transaction gets aborted on detection of a conflict. Let us call this as *current-transaction-abort* or *CTA*. (2) the transactions conflicting with validating transaction gets aborted. We call this *other-transaction-abort* or *OTA*. Hence, there are four variants: (1) BOCC-CTA (2) BOCC-OTA (3) FOCC-CTA (4) FOCC-OTA

To test the performance of both the algorithms, develop an application, opt-test is as follows. Once, the program starts, it creates n threads and an array of m shared variable. Each of these threads, will update the shared array randomly. Since the threads could simultaneously update the shared variables of the array, the access to shared variables have to be synchronized. This synchornization is performed using the above mentioned methods of BOCC, FOCC and the variants.

To pseudocode opt-test given is as follows explains the idea better:

Listing 1: main thread

```
1 void main()
2 {
```

```
    ...
    ...
    // create a shared array of size m
    shared[] = new SharedArray[m];
    ...
    ...
    create n updtMem threads;
}
```

Listing 2: updtMem thread

```
void updtMem()
{
    int status = abort;        // declare status variable
    int abortCnt = 0;          // keeps track of abort count

    long critStartTime, critEndTime;

    // Each thread invokes numTrans transactions
    for (int curTrans=0; curTrans<numTrans; curTrans++)
    {
        abortCnt = 0;                       // Reset the abort count
        critStartTime = getSysTime();       // keep track of critical section start time

        // getRand(k) function used in this loop generates a random number in the range 0 .. k
        do
        {
            id = begin_trans(); // begins a new transaction id
            randIters = getRand(m); // gets the number of iterations to be updated

            int locVal;
            for (int i=0; i<randIters; i++)
            {
                // gets the next random index to be updated
                randInd = getRand(m);

                // gets a random value using the constant constVal
                randVal = getRand(constVal);

                // reads the shared value at index randInd into locVal
                read(id, shared[randInd], locVal);

                logFile << "Thread id " << pthread_self() << "Transaction " << id <<
                " reads from" << randInd << " a value " << locVal << " at time " <<
                getSysTime;

                // update the value
                locVal += randVal;

                // request to write back to the shared memory
                write(id, shared[randInd], locVal);

                logFile << "Thread id " << pthread_self() << "Transaction " << id <<
                " writes to " << randInd << " a value " << locVal << " at time " <<
                getSysTime;
```

```
46
47                  // sleep for a random amount of time which simulates some complex computation
48                  randTime = getExpRand(λ);
49                  sleep(randTime);
50              }
51
52              status = tryCommit(id);       // try to commit the transaction
53              logFile ≪ "Transaction " ≪ id ≪ " tryCommits with result "
54              ≪ status ≪ " at time " << getSysTime;
55              abortCnt++;                   // Increment the abort count
56          }
57          while (status! = commit);
58
59          critEndTime = getSysTime(); // keep track of critical section end time
60          record commitDelay & abortCnt;        // Record these values for collecting statistics
61      }   // End numTrans for
62  }
```

Here $randTime$ is an exponentially distributed with an average of λ mill-seconds. The objective of having this time delays is to simulate that these threads are performing some complicated time consuming tasks. It can be seen that the time taken by a transaction to commit, *commitDelay* is defined as $critEndTime - critStartTime$.

Input: The input to the program will be a file, named inp-params.txt, consisting of all the parameters described above:$n, m, numTrans, constVal, \lambda$. A sample input file is: 10 10 50 100 20.

Output: Your program should output two files in the format given in the pseudocode for each algorithm: (1) BOCC-CTA-log.txt (2) BOCC-OTA-log.txt (3) FOCC-CTA-log.txt (4) FOCC-OTA-log.txt. A sample output is as follows:

The Output of your program (for any of the variants mentioned):
Thread 1 Transaction 1 reads 5 a value 0 at time 10:00
Thread 2 Transaction 2 reads 7 a value 0 at time 10:02
Thread 1 Transaction 1 writes 5 a value 15 at time 10:05
Thread 2 Transaction 2 tryCommits with result abort at time 10:10

.
.
.

The output is essentially a history. By inspecting the output one should be able to verify the serializability of your implementations.

Report: You have to submit a report for this assignment. This report should contain a comparison of the performance of the various variants of BOCC & FOCC. The comparison must consist of two graphs:

- A graph comparing the average time taken by a transaction to successfully commit, i.e. commitDelay in the variants of BOCC & FOCC. The x-axis should be the number of transactions varying from 500 to 1000 in the increments of 100; the y-axis is the average commitDelay of all the transactions for each of the algorithms. This graph will have four curves, one for each of the algorithms: (1) BOCC-CTA (2) BOCC-OTA (3) FOCC-CTA (4) FOCC-OTA.

- A graph comparing the average abort count, i.e. the number of times a transaction abort before it can successfully commit . The x-axis should be the number of transactions varying from 500 to 1000 in the increments of 100; while the y-axis should be abort count. Thus, this graph similar to the previous graph will have two curves: one representing BOCC and the other representing FOCC.

You must run both these algorithms multiple times to obtain performances values. You can vary have the number of threads fixed in all these experiments. Please have m to be fixed at 10 in all these experiments. Finally in your report, you must also give an analysis of the results while explaining any anomalies observed.

Deliverables: You have to submit the following:

- The source files of BOCC & FOCC. Name them as: BOCC-CTA-<rollno>.cpp, BOCC-OTA-<rollno>.cpp, FOCC-CTA-<rollno>.cpp, FOCC-OTA-<rollno>.cpp.
- A readme.txt that explains how to execute the program.
- The report as explained above

Zip all the three files and name it as ProgAssn1-<rollno>.zip. Then upload it on the google classroom page of this course. Submit it by the deadline mentioned above.