### ECEN 4313 Concurrent Programming

### Final Project Write Up

### Elena Murray

# Project Overview

## Code Organization

The code is organized in to four parts: main, bank\_tester, bank, and ticket\_lock. Bank contains the concurrent bank class that implements bank transactions using one of the following methods: single-global lock, two phase locking, software transactional memory, hardware transactional memory, and an optimistic concurrency control. The bank class has a field that stores the transaction implementation to be used and for each call to deposits, withdraws, and transfers a switch statement based off of this field was used to execute the selected transaction method. The bank\_tester simulates the banking system as a whole and assigns threads to execute a set of banking transactions (deposits, withdraws, transfers). It also computes the transactional throughput. Ticket\_lock is code from lab 2 and is used for locks. Main is responsible for reading in user input and using said input to configure the bank and bank\_tester.

## SGL Approach

For the single global lock deposits, withdraws, and transfers I simply protected accesses with a single global lock.

## Two Phase Locking Approach

For two phase locking I created an array of locks where each lock is responsible for protecting the account with the same id number. Deposits and withdraws, I protected my changes with the lock associated with the account number. Transfers were much harder. Initially, I acquired locks for the account to transfer money from and the account to transfer money to. I then made my changes and released both locks. However, this resulted in cases of deadlock. I tried resolving this by using a trylock scheme and a while loop to acquire both locks simultaneously. If only one lock was acquired, it would be released, and the thread would try to acquire the lock again. This also was unable to prevent deadlock. I ultimately executed withdraw first protected by the lock associated with the account to transfer money from. If this withdraw was successful, a deposit would be executed, protected by the lock associated with the account to transfer money to. This resolved the issue with deadlock.

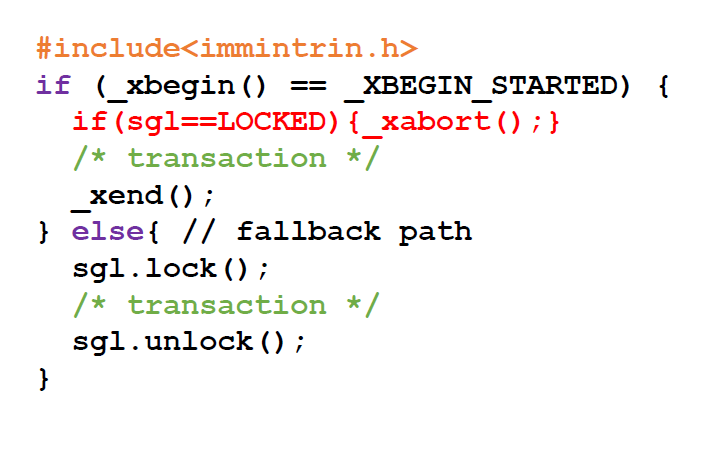
## STM Approach

For implementing software transactional memory, I used C++’s experimental transactions. I created transaction\_safe functions (account\_withdraw, account\_deposit) for withdrawing and depositing money into a given account. The account\_withdraw function ensured that the account balance was greater than the withdraw amount before withdrawing money. If this was not the case the function returned -1 and did not make any changes to the account balance. These functions were then called by the appropriate bank transaction. For example, deposit called the account\_deposit function. Transfer first called account\_withdraw and if this call was successful, it then called the account\_deposit function.

## HTM Approach

For implementing hardware transactional memory, I used Intel’s TSX library. I structure the TSX transaction to repeat for a fixed number of times before using the single global lock. The transaction also checked to see if the lock was held as was shown in the example in class. This example, shown below, served as a model for my approach.

Figure 1: TSX with SGL example from class



Because I needed to check the lock status, I could not use the pthreads mutex. As a result, I used one of my custom locks from lab 2. I decided to use my ticket lock as it would be fairer to threads. I did modify it slightly to have a trylock function and a function that checks to see if the lock is held.

## Optimistic Concurrency Control Approach

I based my approach for an optimistic concurrency control on the TL2 algorithm covered in class. My read set included the account balances I was planning on editing and the total. These were stored in thread\_local variables. My write set included the new total and account that used the read set. These too were stored in thread\_local variables. I then used a global lock and then checked the read set. If the read set was unchanged then I set the actual account variables and total to the local variables in the write set. If the read set was changed then the transaction was unsuccessful. I looped until the transaction was successful.

# File Structure

* main:
  + Reads in user input to create a bank and initiates the bank system. Uses an initialization file to configure the starting balances of the bank. Uses a transaction file that determines what type of banking transactions (i.e deposits, withdraws, and transfers) to run on specified accounts and known amounts.
* bank:
  + Contains my implementation of a concurrent bank as a class containing a fixed sized array of structs. Supports deposits, withdraws, and transfers using the selected transaction method (SGL, STM, HTM, etc.).
* bank\_tester:
  + Contains a parallelized banking system where each thread processes a selection of transactions.
* ticket\_lock:
  + Contains my implementation of the ticket lock which is used by the bank for lock protection.
* test\_files/
  + high\_contention/
    - init\_data.txt:
      * Contains the starting balances to initialize the bank to for this test.
    - txn\_data.txt:
      * Contains the transactions to run for this test. Configure for high contention between threads.
    - ledger.txt
      * File generated by running bank with the init\_data.txt and txn\_data.txt files in this folder. Contains the final output of the bank.
    - ledger\_soln.txt
      * File used to validate ledger.txt and the test.
  + low\_contention/
    - init\_data.txt:
      * Contains the starting balances to initialize the bank to for this test.
    - txn\_data.txt:
      * Contains the transactions to run for this test. Configure for minimal contention between threads.
    - ledger.txt
      * File generated by running bank with the init\_data.txt and txn\_data.txt files in this folder. Contains the final output of the bank.
    - ledger\_soln.txt
      * File used to validate ledger.txt and the test.
* Makefile:
  + The Makefile used for compiling and creating the executable.
* bank\_script:
  + Used to test functionality all transaction methods for a given test (high\_contention or low\_contention) and thread count. Serves as a means of unit testing. Tests high contention by running bank for all transaction methods with the input files in /high\_contention. Tests low contention by running bank for all transaction methods with the input files in /low\_contention. Verifies that the ledger produced from the bank call matches the specified solution ledger.
* /docs:
  + Contains documentation on the project and the data collected from the experiment.

# Transaction Throughput

## Experimentation Methodology:

The throughput that I choose to measure was the transaction throughput. To me this made sense over measuring cycles and other throughput as the goal of this project was to implement transactions. All my calculations were done internal to the program as perf would be able to identify my custom transactions. As a result, I computed throughput on a per thread basis by dividing the number of transactions a given thread processed by the time for the thread to finish all transactions. From these per thread throughput calculations, I then calculated the average throughput. This value is displayed in bank\_tester at the end of the bank\_tester function. I ran all experiments on the Jupyter server.

To test how my transaction implementations handles contention, I created test files that either purposely maximized or minimized contention, hence the low\_contention and high\_contention files.

## Test Results:

I collected data for both high and low contention across all implementations (SGL, Two Phase, STM, HTM, and Optimistic CC) with varying thread counts that ranged from high to low values. The data from each implementation, test, and thread count are shown in the tables below:

Table : SGL Throughput Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SGL Transaction Implementation | | | | | | | | | | | | | | | | |
| Contention Status | Thread Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15 | 20 | 25 | 30 | 35 |
| High | Average Throughput (txns/ns) | 0.00008 | 0.000035 | 0.000031 | 0.000028 | 0.000026 | 0.000014 | 0.000018 | 0.000016 | 0.000009 | 0.000009 | 0.000007 | 0.000008 | 0.000005 | 0.000005 | 0.000005 |
| Average Throughput (txns/s) | 79594.44531 | 34865.58203 | 30755.40625 | 27590.17969 | 26434.69727 | 13758.17676 | 17654.95117 | 15739.02734 | 9344.768555 | 9136.186523 | 6605.494141 | 8277.607422 | 4858.189941 | 5080.721191 | 5096.743652 |
| Low | Average Throughput (txns/ns) | 0.000185 | 0.000171 | 0.000031 | 0.0001 | 0.000021 | 0.000074 | 0.000013 | 0.000007 | 0.000008 | 0.000082 | 0.000009 | 0.000004 | 0.000004 | 0.000003 | 0.000017 |
| Average Throughput (txns/s) | 185157.9531 | 170694.25 | 31052.54102 | 100394.25 | 20829.27344 | 74250.84375 | 12589.49316 | 7006.71875 | 7903.827148 | 82390.54688 | 8591.917969 | 4100.473633 | 4055.96582 | 2878.787354 | 16545.90625 |

Table : Two Phase Locking Throughput Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Two Phase Locking Transaction Implementation | | | | | | | | | | | | | | | | |
| Contention Status | Thread Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15 | 20 | 25 | 30 | 35 |
| High | Average Throughput (txns/ns) | 0.000096 | 0.000033 | 0.000033 | 0.000085 | 0.000027 | 0.000015 | 0.000019 | 0.00001 | 0.000017 | 0.000009 | 0.000005 | 0.000004 | 0.000005 | 0.000004 | 0.000051 |
| Average Throughput (txns/s) | 96057.36719 | 32828.38281 | 33001.85547 | 84951.14063 | 26931.30664 | 14545.60254 | 18648.00195 | 10396.5 | 16671.10742 | 9282.835938 | 4932.439941 | 4250.602051 | 4667.65918 | 4027.512939 | 50539.23047 |
| Low | Average Throughput (txns/ns) | 0.000092 | 0.00003 | 0.000029 | 0.000014 | 0.000019 | 0.000011 | 0.000014 | 0.000014 | 0.000008 | 0.000008 | 0.000005 | 0.000007 | 0.000031 | 0.000005 | 0.000002 |
| Average Throughput (txns/s) | 92155.00781 | 29594.30664 | 28503.28906 | 14372.10059 | 18660.47656 | 10724.84473 | 13604.72949 | 14285.92969 | 7839.189941 | 7618.383789 | 5262.038574 | 6644.799805 | 31235.38477 | 5494.387695 | 2309.02124 |

Table : STM Throughput Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| STM Transaction Implementation | | | | | | | | | | | | | | | | |
| Contention Status | Thread Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15 | 20 | 25 | 30 | 35 |
| High | Average Throughput (txns/ns) | 0.000083 | 0.000163 | 0.000032 | 0.000023 | 0.000016 | 0.000014 | 0.000012 | 0.000011 | 0.00001 | 0.000008 | 0.000009 | 0.000041 | 0.000005 | 0.000006 | 0.000003 |
| Average Throughput (txns/s) | 82589.46094 | 162715.5625 | 31718.53711 | 23429.85938 | 15881.53418 | 13922.54297 | 12395.32129 | 10570.80566 | 10209.48242 | 7963.34375 | 9036.780273 | 40762.06641 | 4538.916504 | 6200.39502 | 3255.375 |
| Low | Average Throughput (txns/ns) | 0.000404 | 0.000041 | 0.000031 | 0.000015 | 0.000021 | 0.000021 | 0.000012 | 0.000063 | 0.000008 | 0.000037 | 0.000007 | 0.000005 | 0.000005 | 0.000005 | 0.000003 |
| Average Throughput (txns/s) | 404110.5313 | 40682.14844 | 30734.95898 | 14633.53516 | 20917.02734 | 21354.83398 | 11916.2793 | 62878.13672 | 8202.833984 | 37453.30078 | 6892.921875 | 5353.479004 | 4792.48584 | 4883.834473 | 3010.306152 |

Table : HTM Throughput Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| HTM Transaction Implementation | | | | | | | | | | | | | | | | |
| Contention Status | Thread Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15 | 20 | 25 | 30 | 35 |
| High | Average Throughput (txns/ns) | 0.0001 | 0.000037 | 0.000036 | 0.000015 | 0.000023 | 0.000017 | 0.000095 | 0.000077 | 0.000044 | 0.000006 | 0.000006 | 0.00005 | 0.000005 | 0.000004 | 0.000004 |
| Average Throughput (txns/s) | 99861.30469 | 36945.83594 | 35793.98047 | 15461.85156 | 23469.51367 | 17474.90039 | 95491.60156 | 77012.57031 | 43830.08203 | 5925.741211 | 5573.581543 | 50419.39453 | 4894.004883 | 3709.30127 | 4095.800537 |
| Low | Average Throughput (txns/ns) | 0.000097 | 0.000042 | 0.000028 | 0.000017 | 0.000013 | 0.000018 | 0.000013 | 0.000009 | 0.000013 | 0.000013 | 0.000009 | 0.000006 | 0.000004 | 0.000005 | 0.000004 |
| Average Throughput (txns/s) | 97130.53906 | 42330.72266 | 28239.34961 | 17432.41406 | 12931.29102 | 17652.97461 | 13001.25293 | 9205 | 13142.87207 | 13329.20801 | 9356.826172 | 6480.947754 | 4201.607422 | 5184.01123 | 3831.920898 |

Table : Optimistic Concurrency Control Throughput Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Optimistic Transaction Implementation | | | | | | | | | | | | | | | | |
| Contention Status | Thread Count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15 | 20 | 25 | 30 | 35 |
| High | Average Throughput (txns/ns) | 0.000429 | 0.000184 | 0.000163 | 0.00008 | 0.000101 | 0.000055 | 0.000057 | 0.00001 | 0.00001 | 0.000016 | 0.000007 | 0.000004 | 0.000007 | 0.000005 | 0.000004 |
| Average Throughput (txns/s) | 429174.625 | 183565.25 | 163429.4219 | 79858.49219 | 101227.5547 | 54827.80078 | 56618.44141 | 9846.922852 | 10437.4375 | 15696.69043 | 6921.17334 | 4140.543945 | 7233.138184 | 4925.152832 | 3858.750977 |
| Low | Average Throughput (txns/ns) | 0.00008 | 0.000238 | 0.00003 | 0.000022 | 0.000018 | 0.000016 | 0.000013 | 0.000013 | 0.000104 | 0.000016 | 0.000008 | 0.000007 | 0.000006 | 0.000004 | 0.000004 |
| Average Throughput (txns/s) | 79633.125 | 238315.1875 | 29551.98438 | 22037.90625 | 17756.78125 | 15847.7334 | 13227.04981 | 12815.45898 | 103537.4375 | 15740.18945 | 7797.942871 | 7455.091309 | 5751.645508 | 4115.157227 | 4428.605957 |

### Analysis:

Across all methods and for both high and low contention cases, the data generally shows that as thread count increases average throughput decreases. This visually can be seen in the following plot:

Plot : Throughput versus Thread Count

It is hard to give a concrete explanation on to why this is the case. I believe there are several factors that encourage this trend. Firstly, there is a finite limit to the number of hardware threads available. The throughput overall reduces after 10 – 15 threads, which is about the limit of the JupyterLab server. Having more threads than available hardware threads means that threads will be de-scheduled which reduces throughput. Also, how the transactions get distributed to each thread varies for each thread count as the transaction array is divided based off of the number of threads. Thus, there are different combinations of thread transaction subsets. This may increase or decrease contention for different thread counts and could explain why there are peaks at various threads counts as these may have reduced contention. However, further testing is required to analyze determine the actual cause of this behavior. Perhaps running multiple iterations of the same test and averaging the results would provide a better understanding of the relationship between throughput and thread count.

# Compilation Instructions

1. Run the Makefile using: make
   1. This will generate all object files and the executable.
2. Clean the project using: make clean

# Execution Instructions

* SGL
  1. ./bank –init init\_file.txt –txn txn\_file.txt -o ledger.txt -t NUM\_THREADS –alg=sgl
* Two Phase Locking
  1. ./bank –init init\_file.txt –txn txn\_file.txt -o ledger.txt -t NUM\_THREADS –alg=2pl
* STM
  1. ./bank –init init\_file.txt –txn txn\_file.txt -o ledger.txt -t NUM\_THREADS –alg=stm
* HTM
  1. ./bank –init init\_file.txt –txn txn\_file.txt -o ledger.txt -t NUM\_THREADS –alg=htm
* Optimistic
  1. ./bank –init init\_file.txt –txn txn\_file.txt -o ledger.txt -t NUM\_THREADS –alg=opt
* bank\_script.sh:
  1. Low contention test: ./bank\_script -t NUM\_THREADS -c low\_contention
  2. Low contention test: ./bank\_script -t NUM\_THREADS -c high\_contention

# Known Assumptions

1. All files must be text files.
2. The number of iterations must be greater than or equal to 1 for counter.
3. If the number of threads exceeds the number of transactions, the number of threads will be set to the number of transactions. A warning will appear, but bank will still run.
4. The bank’s total must be the same as the sum of all the account balances.

# Known Bugs

1. Occasionally, two phase locking will not work.
2. Additionally, the print statements that print out the current transaction are needed for two phase locking to work.