Language Abstractions for Concurrent and Parallel Programming Uppsala University – Autumn 2017 Report for Assignment 1

Nam Nguyen, Nevine Gouda

27th November 2017

1 Exercise 1: Warm-Up

1. Execute `transactions.cpp`

Compile: `g++ -fgnu-tm -std=c++11 transactions.cpp -o transaction`

Execution: `./transaction`

Output: `diff: 0`

As can be seen, the reported value of `diff` is 0. Using atomic transactions successfully prevents data race from happening as one transaction cannot interrupt the other.

2. Strong/weak isolation

The definition of weak and strong isolation are: Weak Isolation - "Transactions are serializable only against other transactions" Strong Isolation - "Transactions are serializable against all memory accesses (Non-transactional LD/ST are 1-instruction TXs)" [Transactional Memory - Implementation Lecture 1, Arun Raman, Princeton University]

According to this, in a data race free programs, atomic transactions provide strong isolation because the all memory accesses are not conflicting/creating data races. In fact, in a data race free program, using atomic transaction is not even necessary.

3. Non-transactional statements

The modified version is called `transactions.cpp` and is included in the submission. The reported value of `diff` changes with multiple runs, where it changes from 1231114, 2028766, 1670923, etc.

Where it is obvious that since the shared variables are not properly protected/isolated, the changes to x and y in the toggle function, are affecting their reads in the check function, which in turn affects the `diff` variable.

4. Data-race and/or race condition

A *race condition* is caused by timing sensitivity, which occurs when the timing or ordering of events affects a program's outcome. Typically, this is led to by either external timing or ordering non-determinism. A data race may cause the race condition, but race condition does not necessarily be caused by a data race.

A *data race* occurs when two memory accesses other than reads that are performed concurrently by two threads target the same location without any protection/synchronization.

Thus, due to lack of protection or locks on the critical sections and shared variables, and since there is both reads and writes, both a data race and a race condition exist. This implies that the program is nondeterministic and the outputted result is unreliable.

2 Exercise 2: Double-Ended Queue

Implementation of `deque.cpp`

Compile: `g++ -fgnu-tm -std=c++11 deque.cpp -o deque`

Execution: `./deque`

Notice: the `main()` function is commented out because otherwise, the `work-steaking.cpp` program will complain about re-declaring `main()`. Please remove the comment on `main()` function in `deque.cpp` to run it.

The two ends of the queue are managed by `leftSentinel`, `rightSentinel` such that values are popped/pushed from/to at one of two queue ends. A new node is added to the queue as both the left and the right sentinels if the queue is empty. In the opposite, if the queue has only one element, removing that from the queue would set both `leftSentinel` and `rightSentinel` to `nullptr`. Synchronization is done such that the whole operation of pushing and locking is done within `__transaction_atomic`. This makes the code clear to read/understand but serializes the concurrent execution of the program.

3 Exercise 3: Performance Measurements

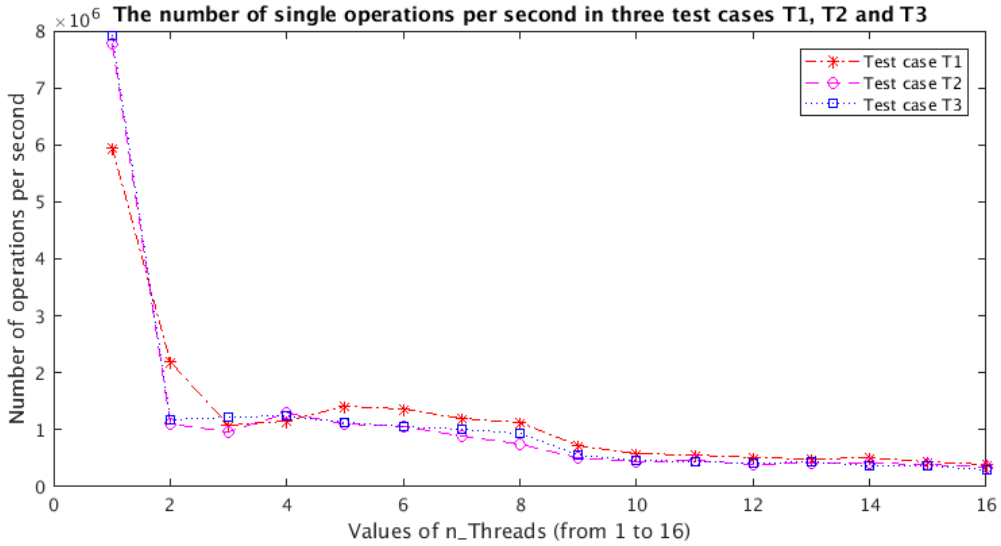
A driver program that exercises our `DQueue` class to determine its performance under different workloads was implemented as `Ex3.cpp`. A `counter` is added to the `DQueue` structure so as to count the number of operations. This counter is increased by 1 every time one operation is completed and this increment is done inside `__transaction_atomic` to make sure no data races happen. Three test cases were experimented: only `PushLeft()`, `PushLeft()` followed by a `PopLeft()` and `PushLeft()` followed by a `PopRight()` - accordingly notated $T1$, $T2$ and $T3$.

Compile: `g++ -fgnu-tm -std=c++11 Ex3.cpp -o ex3`

Execution: `./ex3 [number_of_threads]`

This driver program was tested on a Linux lab machine Intel(R) Xeon(R) CPU E5520 processor with 16 (hyper-threading) cores. For each of the three measurements, different numbers of concurrent threads from 1 to 16 was used, as 16 is the maximum speedup that a concurrent program could achieve in the aforementioned machine. The experiment was done out of lab hours and the machine was lightly loaded. The benchmark performance of our `DQueue` class is shown in figure 1.

Figure 1: Number of single operations per second in three test cases $T1$, $T2$ and $T3$ with respect to different number of threads (1-16)



As you can see in the figure, a decreasing trend is seen in all three test cases. The number of single operations per second dropped drastically when using 2 threads compare to sequential execution on one thread. If the number of threads keeps increasing, the performance stayed about the same at 10^6 until it halved to 5×10^5 when using more than 8 threads. This behavior could be explained as due to an increase in the amount of conflicts between transactions when using more threads. When more than two threads access the same element of the queue to write (push) - in test case $T1$ - the leftmost node of the queue, only one got to commit the change to the value. In the worst case, one transaction could repeat itself many times but not commit. This effect is intensified when more threads are in use, leading to a steady decrease when using more threads.

4 Exercise 4: Semantics: Exceptions

1. Implementation

A program throwing exception in atomic transactions is implemented in the file `Ex4.cpp` submitted with this report. The program calculates the ticket price for a person based on his/her age, which is computed from the birth year `birth_year` inputted.

Compile: `g++ -fgnu-tm -std=c++11 Ex4.cpp -o Ex4`

Execution: `./Ex4`

2. Execution

Output: If input `birth_year` ≤ 2017, the transaction commits, and prints a ticket price based on the calculated age. Otherwise, the transaction aborts with the following message `"Birth year entered is bigger than 2017"` and the ticket price is -100, which is set inside the transaction before the exception was thrown. This is not good because if the transaction is failed to commit, we want to restore the initial values of all variables.

3. Cancel-and-throw semantics

If we, instead of using `throw exception`, use `__transaction_cancel`, the transaction will be canceled and all the variables whose value has been changed inside the transaction will be rolled back. However, it does not throw an exception because the cancel takes immediate effect. Putting a `throw exception` before that will lead to the previous case, while throwing an exception after `__transaction_cancel` has no effect according to the previous discussion. This is not good because if the transaction is failed to commit, we want to be notified with the encountered error.

If `__transaction_cancel throw throw-expression` is reached in a program, the immediate enclosing transaction is canceled and all the side effects done by the transaction is rolled back. At the same time, an expression of value `throw-expression` is thrown to notify what the error was. If the `cancel-throw` statement was located inside nested transactions, it only cancels the inner transaction and propagates the thrown exception to the outer transaction to catch it.

5 Exercise 5: Semantics: Nesting

1. Implementation

A program throwing exception in atomic transactions is implemented in the file `Ex5.cpp` submitted with this report. The program contains two nested transactions. The inner transaction modifies the value of y from 0 to 1 while the outer one modifies x from 0 to 1. Two variables A_commit and B_commit are used to accordingly mark if the outer and the inner transaction commits. $A\_commit = 0$ means the outer transaction is not committed - canceled; meanwhile, $A\_commit = 1$ means the outer transaction is successful - committed. The same principle applies for B_commit.

Compile: `g++ -fgnu-tm -std=c++11 Ex5.cpp -o Ex5`

Execution: `./Ex5`

2.Nesting Semantics in gcc.

Table 1: Flattened Nesting

A_commit	B_commit	Output A	Output B
0	0	Abort	Abort
0	1	Abort	Abort
1	0	Abort	Abort
1	1	Commit	Commit

Table 2: Closed Nesting

A_commit	B_commit	Output A	Output B
0	0	Abort	Abort
0	1	Abort	Abort
1	0	Commit	Abort
1	1	Commit	Commit

Three nesting types called flattened, closed and opened were implemented in `Ex5.cpp` given two nested transaction. For each nesting type, we constructed the semantics in a truth-table-like fashion. Experiments with the implemented program returned correct results as seen in the following tables. To be specific, table 1 represents the semantics of flattened nesting. It shows that canceling the inner transaction causes the both transactions to abort, while committing the inner transaction has no effect except until the outer transaction commits. In other words if either of the transactions abort, then both will abort.

In the semantics for the closed nesting found in table 2, canceling the inner transaction transfers the control to the outer transaction, while committing the inner transaction has no effect until the outer transaction actually commits.

Finally the semantics for the opened nesting can be found in Table 3. This shows that canceling the inner transaction transfers the control to the outer transaction, however committing the inner transaction takes effect immediately regardless of the outer transaction.

6 Exercise 6: Relaxed Transactions

1. Execute `relaxed.cpp`

Output: An error is outputted as shown in figure 2. The reason is that `cout` is a buffered stream, which means that data written to the buffer will be printed out when the `endl` marker or a `cin` statement is reached, or a buffer flush `cout.flush()` is executed. This makes `std::cout` a transaction-unsafe function and thus, cannot be called inside an atomic transaction.

Table 3: Opened Nesting

A_commit	B_commit	Output A	Output B
0	0	Abort	Abort
0	1	Abort	Commit
1	0	Commit	Abort
1	1	Commit	Commit

Figure 2: Error returned when using `std::cout` inside a relaxed transaction

```
                ^~~~~~~~
relaxed.cpp:9:29: error: unsafe function call 'std::basic_ostream<_CharT, _Traits>::__ostream_type& std::basic_ostream<_C
harT, _Traits>::operator<<(std::basic_ostream<_CharT, _Traits>::__ostream_type& (*)(std::basic_ostream<_CharT, _Traits>::
__ostream_type&)) [with _CharT = char; _Traits = std::char_traits<char>]' within atomic transaction
       std::cout << "world!" << std::flush;
       ~~~~~~~~~~~~~~~~~~~~~~^~~~~~
```

2. Change `__transaction_atomic` to `__transaction_relaxed`

The program `Ex6.cpp` is modified from `transactions.cpp` by changing `__transaction_atomic` to `__transaction_relaxed` and is submitted with this report.

Compile: `g++ -fgnu-tm -std=c++11 Ex6.cpp -o Ex6`

Execution: `./Ex6`

Output: "brave new Hello world!" or "Hello world! brave new" or "Hello brave new world!".

3. Elaboration

"Another output" can be "Hello brave new world!", though we did actually get that output after tens of executions.

SLA stands for single lock atomic while TSC stands for transaction sequential consistency. A relaxed transaction does not provide sequential consistency and is not singly lock atomic as illustrated by the output of `Ex6.cpp` where a relaxed transaction can be interleaved by a non-transactional operation.

7 Exercise 7: Work Stealing

1. Implementation of work stealing

The program `work-stealing.cpp` is extended by completing the `processor()` function and is submitted with this report. The idea behind this is simple. A processor first checks its job queue and pops the leftmost job to execute if the job queue is not empty. Otherwise, this processor will check the job queue of other processors, steal and execute the rightmost job with non-zero duration. If the rightmost job is of duration 0, it will be push back to the right side of the original queue, and this is done atomically in a transaction which makes it appear transparent to other processors.

Compile: `g++ -fgnu-tm -std=c++11 work-stealing.cpp -o work-stealing`

Execution: `./work-stealing`

2. Testing

Figure 3: Output for `work-stealing.cpp`

```
38: scheduling a job on processor 1 (duration: 4 s).
Processor 1: executing job (duration: 4 s).
39: scheduling a job on processor 3 (duration: 3 s).
Processor 0: stealing job (duration: 3 s) from processor 3.
40: scheduling a job on processor 2 (duration: 0 s).
Processor 2: executing job (duration: 0 s).
41: scheduling a job on processor 0 (duration: 3 s).
Processor 2: stealing job (duration: 3 s) from processor 0.
42: scheduling a job on processor 3 (duration: 4 s).
Processor 1: stealing job (duration: 4 s) from processor 3.
43: scheduling a job on processor 1 (duration: 0 s).
44: scheduling a job on processor 1 (duration: 0 s).
45: scheduling a job on processor 1 (duration: 1 s).
Processor 3: stealing job (duration: 1 s) from processor 1.
Processor 1: executing job (duration: 0 s).
Processor 1: executing job (duration: 0 s).
46: scheduling a job on processor 2 (duration: 2 s).
Processor 2: executing job (duration: 2 s).
47: scheduling a job on processor 1 (duration: 1 s).
Processor 0: stealing job (duration: 1 s) from processor 1.
48: scheduling a job on processor 0 (duration: 2 s).
Processor 2: stealing job (duration: 2 s) from processor 0.
49: scheduling a job on processor 1 (duration: 3 s).
Processor 0: stealing job (duration: 3 s) from processor 1.
```

As can be seen in figure 3, at the second 42, processor 1 has an empty job queue and stole a job of duration 4s from processor 3. In the next two consecutive clock second, two jobs of duration 0s are added to the job queue of processor 1 but no other processors can steal it - only processor 1 after finishing the current job at the second 45 can execute the two duration 0 jobs.