Lock-Free Readers/Writers

Anupriya Chakraborty[1], Sourav Saha[2], Ryan Saptarshi Ray[3] and Utpal Kumar Ray[4]

**[1] Department of Information Technology, Jadavpur University Salt Lake Campus
Kolkata, West Bengal 700098, India**

**[2] Department of Information Technology, Jadavpur University Salt Lake Campus
Kolkata, West Bengal 700098, India**

**[3] Department of Information Technology, Jadavpur University Salt Lake Campus
Kolkata, West Bengal 700098, India**

**[4]Department of Information Technology, Jadavpur University Salt Lake Campus
Kolkata, West Bengal 700098, India**

Abstract

The past few years have marked the start of a historic transition from sequential to parallel computation.The necessity to write parallel programs is increasing as systems are getting more complex while processor speed increases are slowing down. Current parallel programming uses low-level programming constructs like threads and explicit synchronization using locks to coordinate thread execution. Parallel programs written with these constructs are difficult to design, program and debug. Also locks have many drawbacks which make them a suboptimal solution.

Software Transactional Memory (STM) is a promising new approach to programming shared-memory parallel processors. It is a concurrency control mechanism that is widely considered to be easier to use by programmers than locking. It allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, concurrently executing parts of the program.

This paper shows the concept of writing code using Software Transactional Memory (STM) and the performance comparison of codes using locks with those using STM.

Keywords: *Parallel Programming; Multiprocessing; Locks; Transactions; Software Transactional Memory*

1. Introduction

Generally one has the idea that a program will run faster if one buys a next-generation processor. But currently that is not the case. While the next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If one wants programs to run faster, one must learn to write parallel programs as currently multi-core processors are becoming more and more popular. The past few years have marked the start of a historic transition from sequential to parallel computation. The necessity to write parallel programs is increasing as systems are getting more complex while processor speed increases are slowing down. Parallel Programming means using multiple computing resources like processors for programming so that the time required to perform computations is reduced [**1**].

2. The Readers/Writers Problem

The Readers/Writers Problem of Synchronization can be described as follows:

An object is shared among many threads, each belonging to one of two classes:
– Readers: read data, never modify it
– Writers: read data and modify it

Using a single lock on the data object is overly restrictive:
--There are many readers reading the object at once
– Allow only one writer at any point
–We must control access to the object to permit this protocol.

Correctness criteria:

– Each read or write of the shared data must happen within a critical section.
– Guarantee mutual exclusion for writers.
– Allow multiple readers to execute in the critical section at once.

3. The Readers/Writers Problem using Locks

3.1 Description

The hardest problem that should be overcome when writing parallel programs is that of synchronization. Multiple threads may need to access the same locations in memory and if careful measures are not taken the result can be disastrous. If two threads try to modify the same variable at the same time, the data can become corrupt. Currently locks are used to solve this problem. Locks ensure that a critical section, which is a block of code that contains variables that may be accessed by multiple threads, can only be accessed by one thread at a time. When a thread tries to enter a critical section, it must first acquire that section's lock. If another thread is already holding the lock, the former thread must wait until the lock-holding thread releases the lock, which it does when it leaves the critical section [**2**].
In the Readers/Writers problem there are multiple readers and writers accessing the elements in the same buffer at the same time. The buffer is of fixed size. In the examples below we have taken the buffer size as 100000000. The problem is to synchronize these accesses properly so that when a write operation is occurring it should not be affected by any other read or write operation.

3.2 Reader and Writer Threads Code using Locks

The following code shows the reader and writer threads using threads and locks which solves the Readers/Writers problem:

```
void *reader(void *num_ptr1)
{
    unsigned char num1,*number_ptr1,
                byte_under_stm1;
                unsigned long k;
                int i;
                structtimevalini_tv;
                number_ptr1=num_ptr1;
                num1=*number_ptr1;

  for((k=(((num1*ARRAY_SIZE)/(NUM_THREADS))));
  k<(((num1+1)*ARRAY_SIZE)/(NUM_THREADS))/2;
  k++)
  {
    pthread_mutex_lock(&mutex);
    printf("The data read is %d\n",arr[rcount[num1]]);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}


void *writer(void *num_ptr)
{
  unsigned long byte_under_stm1,k,ki=0;
  unsigned char num, *number_ptr;
  structtimevalini_tv;
  number_ptr=num_ptr;
  num=*number_ptr;

  for(k=((((num)*ARRAY_SIZE)/(NUM_THREADS)));
  k<((num+1)*ARRAY_SIZE)/(NUM_THREADS)/2;
  k++,ki++)
  {
    pthread_mutex_lock(&mutex);
    arr[rcount[num]]=1;
    printf("The data written is %d\n",
    arr[rcount[num]]);
    rcount[num]++;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

3.3 Code Explanation

The above program was tested using NUM_THREADS ranging from 1 to 6 threads (in effect 12 threads are created, one for reading and one for writing, making up a total of 6 reader/writer pairs), created to access the values in array arr. There are two processes, reader and writer whose functions are respectively, as the names suggest. The array is divided into several parts depending on the

value of NUM_THREADS and the reader/writer pair accesses the corresponding part of the array.
The reader thread is invoked using the thread ID which is passed to it as the parameter *num_ptr*. Based on this parameter, each thread accesses the corresponding part of the array. Each element is first locked, thereby entering the critical section, where no other thread may access the data being read. Once the data is read, the lock is released. The writer process works in a similar manner.
In the above program "reader" and "writer" are the two thread processes for reading and writing elements from the buffer respectively. Here the array *arr* is the buffer. The global array *rcount* keeps track of the position of elements in the buffer.

In the thread "reader", elements are read from the buffer by the following statements.

```
for((k=(((num1*ARRAY_SIZE)/(NUM_THREADS))));
k<(((num1+1)*ARRAY_SIZE)/(NUM_THREADS))/2;
k++)
{
   pthread_mutex_lock(&mutex);
   printf("The data read is %d\n",
   arr[rcount[num1]]);

   pthread_mutex_unlock(&mutex);
}
```

In the thread "writer" elements are written into the buffer by the following statements.

```
for(k=((((num)*ARRAY_SIZE)/(NUM_THREADS)));
k<((num+1)*ARRAY_SIZE)/(NUM_THREADS)/2;
k++,ki++)
{
   pthread_mutex_lock(&mutex);
   arr[rcount[num]]=1;
   printf("The data written is %d\n",
   arr[rcount[num]]);
   rcount[num]++;
   pthread_mutex_unlock(&mutex);
}
```

The following statement is used to record the time before the threads are created:
gettimeofday(&ini_tv,NULL);
The same call is also used to record the time when all threads have just finished their executions.

The total time taken is then calculated and printed using the following statement:
printf("Total Time Taken = %ld\n", final_tv.tv_sec - ini_tv.tv_sec);

As it can be seen from the above code snippet, 3 calls related with the mutex are being used. They are as follows:

- **pthread_mutex_init(&mutex,NULL)** is used for lock initialization.
- **pthread_mutex_lock(&mutex)** means that any thread must acquire the lock on mutex to execute the critical section following this function.
- **pthread_mutex_unlock(&mutex)** is used for unlocking.

In this program, the regions where more than one thread may access the global array, *rcount[]* at the same time are the critical sections. Thus these regions are enclosed within locks. Hence when a write operation is occurring in this program it is not being affected by any other read or write operation.

3.4 Experimental Results

We have taken all the experimental data for the outputs shown in this paper by running the codes on a machine which has 6 cores with hyper-threading. Thus, a maximum of 12 threads can run in parallel.

The experimental results for The Readers/Writers Problem using locks are presented below:

Number of Reader/Writer Pairs	Time Taken (Locks)
1	99
2	52
3	33
4	25
5	19
6	16

Table 1: Experimental Results for the Readers/Writers Problem using Threads and Locks

The above experimental data has been represented graphically in Figure 1 and Figure 2 which show the variation of Time Taken for execution of the code, and Speedup respectively, with increase in the number of threads for the code of the Readers/Writers Problem using threads with Locks.

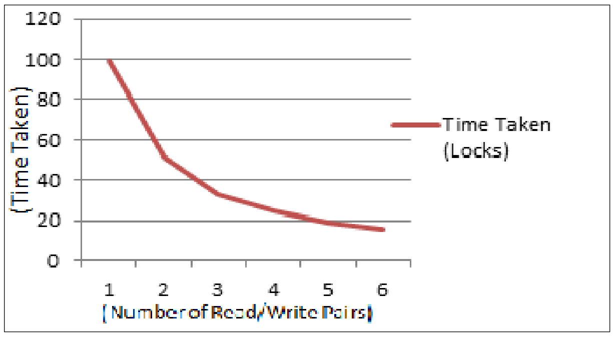

Fig. 1 : Graph showing the Time Taken vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using Threads with Locks

It can be seen from the above graph that as the number of reader/writer pairs increases, the time taken for execution decreases.

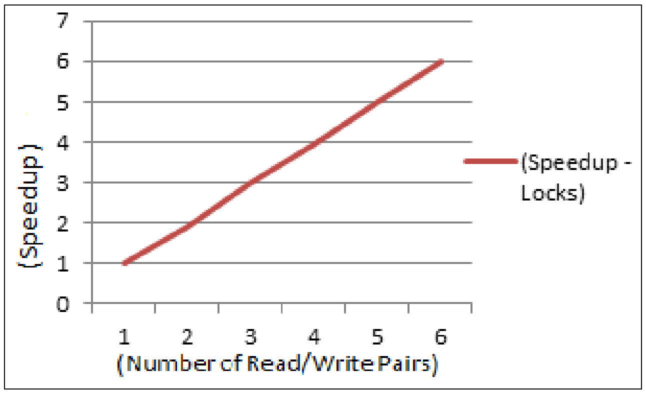

Fig. 2 : Graph showing the Speedup vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using Threads with Locks

We can see that the speedup increases linearly with the number of reader/writer pairs.

4. The Readers/Writers Problem using STM

4.1 Description

The synchronization problem can also be solved using STM. If STM is used in a program then we do not have to use locks in the program. Thus the problems which occur due to the presence of locks in a program do not occur in this type of code. The critical section of the program has to be enclosed within a transaction. Then STM by its internal constructs ensures synchronization in the program.

There are 8 categories of calls associated with STM which have been used in this program. They are as follows:

- **stm_init**is used to initialize the TinySTM library at the outset. It is called from the main thread before accessing any other functions of the TinySTM library.
- **stm_init_thread**is used to initialize each thread that will perform transactions. It is called once from each thread that performs transactional operations before the thread calls any other functions of the TinySTMlibrary. In this program it is called from the threads **reader** and **writer**.
- **stm_exit**is the corresponding shutdown function for stm_init. It cleans up the TinySTM library. It is called once from the main thread after all transactional threads have completed execution.
- **stm_exit_thread**is the corresponding shutdown function for stm_init_thread. It cleans up the transactional thread. It is called once from each thread that performs transactional operations upon exit. In this program it cleans up the threads **reader** and **writer**.
- **START(0,RW)** is used to start a transaction. In this program it is used in the threads **reader** and **writer**.
- **COMMIT** is used to close the transaction. In this program it is used in the threads **reader** and **writer**.
- **byte_under_stm1=(unsigned char)LOAD(&rcount)** stores the value of rcount in byte_under_stm1. In this program it is used in the threads **reader** and **writer**.
- **STORE(&rcount, byte_under_stm1)** stores the value of byte_under_stm1 in rcount. In this program it is used in the threads **reader** and **writer**.

4.2 Reader and Writer Threads Code using STM

The following code shows the reader and writer threads using threads and STM which solves the Readers/WritersProblem:

```
void *reader(void *num_ptr1)
{
  unsigned char num1,*number_ptr1;
  unsigned long k;
  unsigned long byte_under_stm1;
  structtimevalini_tv;
  number_ptr1=num_ptr1;
  num1=*number_ptr1;

  stm_init_thread();

  for((k=(((num1*ARRAY_SIZE)/(NUM_THREADS))));
  k<(((num1+1)*ARRAY_SIZE)/(NUM_THREADS))/2;
  k++)
  {
    START(0,RW);
    byte_under_stm1=(unsigned char)
    LOAD(&rcount);
    printf("The data read is %d\n", arr[rcount[num1]]);
    STORE(&rcount,byte_under_stm1);
    COMMIT;
  }
  stm_exit_thread();
  pthread_exit(0);
}


void *writer(void *num_ptr)
{
  unsigned long byte_under_stm1,k,ki=0;
  unsigned char num, *number_ptr;
  number_ptr=num_ptr;
  num=*number_ptr;
  stm_init_thread();

  for(k=((((num)*ARRAY_SIZE)/(NUM_THREADS)));
  k<((num+1)*ARRAY_SIZE)/(NUM_THREADS)/2;
  k++,ki++)
  {
    START(0,RW);
    byte_under_stm1=(unsigned char)
    LOAD(&rcount);


     arr[rcount[num]]=1;
     printf("The data written is %d\n", arr[rcount[num]]);
    rcount[num]++;
    STORE(&rcount,byte_under_stm1);
    COMMIT;
  }
  stm_exit_thread();
  pthread_exit(0);
}
```

4.3 Code Explanation

The program structure of the above code is same as that of the program for readers-writers problem using threads and locks. The only difference is that STM is being used in this code.
In this program, the regions where more than one thread may access the global array *rcount[]*at the same time are the critical sections. Thus these regions are enclosed within transactions using STM. Hence when a write operation is occurring in this program it is not being affected by any other read or write operation.

4.4 Experimental Results

The experimental results for The Readers/Writers Problem using STM are presented below:

No. of Read/Write Pairs	Time Taken (STM)
1	116
2	59
3	36
4	25
5	23
6	18

Table 2 : Experimental Results for the Readers/Writers Problem using Threads and STM

The above data has been represented graphically in Figure 3 and Figure 4 which show the variation of Time Taken for execution of the code, and Speedup respectively, with increase in the number of threads for the code of the Readers/Writers Problem using threads with STM.

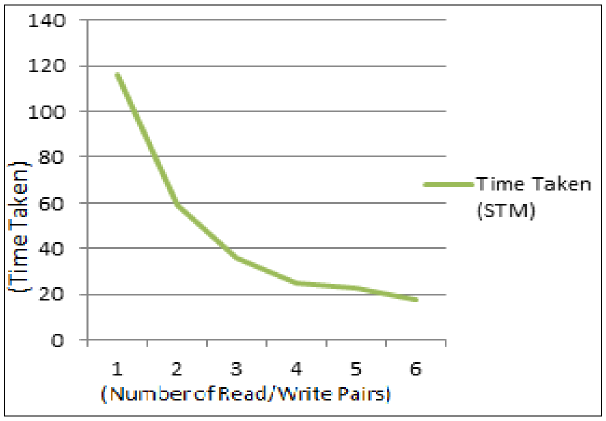

Fig. 3 : Graph showing the Time Taken vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using Threads with STM

It can be seen from the above graph that as the number of reader/writer pairs increases, the time taken for execution decreases.

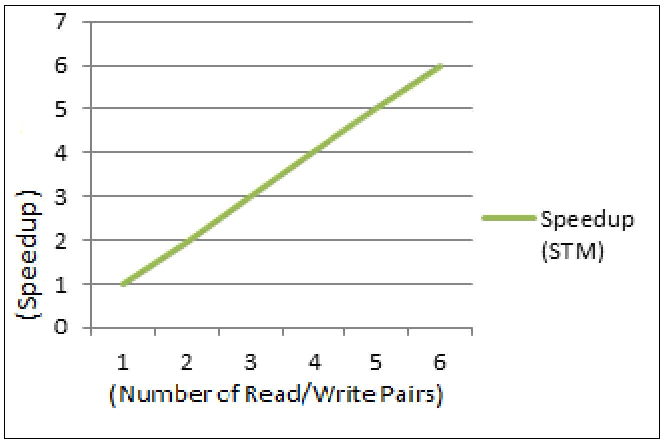

Fig. 4 : Graph showing the Speedup vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using Threads with STM

We can see that the speedup increases linearly with the number of reader/writer pairs.

5. Conclusions

Figure 5 is a combination of Figures 1 and 3 as shown in previous sections.

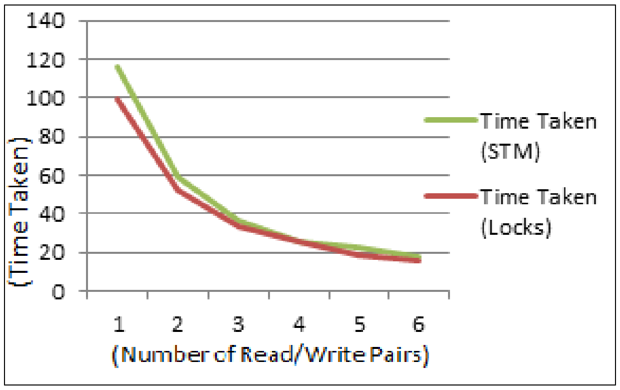

Fig. 5 : Graph showing the Time Taken vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using both Threads with Locks and Threads with STM

We can see from the above graph that the time taken for executing the code using threads and STM is almost equal to the time taken for executing the same code using threads and locks. Thus, further research is being undertaken to improve the execution speed of STM.

Similarly, Figure 6 is a combination of Figures 2 and 4 as shown in previous sections.

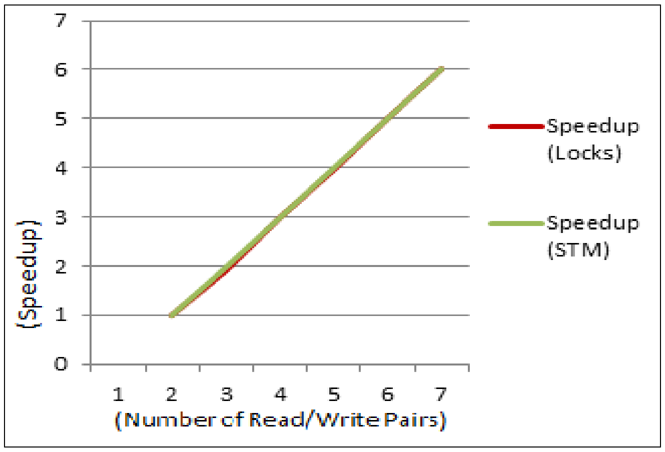

Fig. 6 : Graph showing the Speedup vs. Number of Reader/Writer Pairs for the Readers/Writers Problem using both Threads with Locks and Threads with STM

From the above graph we can see that the speedup for the codes using both locks and STM for this problem, are nearly the same. This is because the times taken for executing both the codes are also nearly equal.

STM has been shown in many ways to be a good alternative to using locks for writing parallel programs. STM provides a time-tested model for isolating concurrent computations from each other. This model raises the level of abstraction for reasoning about concurrent tasks and helps avoid many parallel programming errors.

This paper has discussed how STM can be used to solve the problem of synchronization in parallel programs, and in particular, the Readers/Writers Problem of Synchronization has been solved with Lock-Free Code using STM. STM has ensured that lock-free parallel programs can be written. This ensures that the problems which occur due to the presence of locks in a program do not occur in this type of code.

Many aspects of the semantics and implementation of STM are still the subject of active research. While it may still take some time to overcome the various drawbacks, the necessity for better parallel programming solutions will drive the eventual adoption of STM. Once the adoption of STM begins it will have the potential to pick up momentum and make a very large impact on software development in the long run. In the near future STM will become a central pillar of parallel programming.

References

[1] [1] Simon Peyton Jones, “Beautiful concurrency”.

[2] Elan Dubrofsky, “A Survey Paper on Transactional Memory”.

[3] Pascal Felber, Christof Fetzer, Torvald Riegel, “Dynamic Performance Tuning of Word-Based Software Transactional Memory”.

[4] http://en.wikipedia.org/wiki/Transactional_memory

[5] James Larus and Christos Kozyrakis. “Transactional Memory”

[6] Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel, “Time-Based Software Transactional Memory”

[7] Tim Harris, James Larus, Ravi Rajwar, “Transactional Memory”

[8] Mathias Payer, Thomas R. Gross, “Performance Evaluation of Adaptivity in Software Transactional Memory”

[9] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, David A. Wood., “LogTM: Log-based Transactional Memory”

[10] Dave Dice , Ori Shalev , Nir Shavit., “Transactional Locking II”

[11] http://tmware.org

[12] Maurice Herlihy, J. Eliot B. Moss, “Transactional Memory:Architectural Support for Lock-Free Data Structures”.

[13] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, Luca Benini, **“**Towards Transactional MemorySupport for GCC”.

[14] Virendra J. Marathe, Michael F. Spear, Christopher Heriot,Athul Acharya, David Eisenstat, William N. Scherer III, Michael L. Scott, “Lowering the Overhead ofNonblocking Software Transactional Memory”.

[15] Utku Aydonat, Tarek S. Abdelrahman,Edward S. Rogers Sr., “Serializability of Transactions inSoftware Transactional Memory”.

[16] Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”.

[17] Brendan Linn, Chanseok Oh, “G22.2631 project report: software transactional memory”.

[18] Ryan Saptarshi Ray , “WritingLock-FreeCode using Software Transactional Memory”.

[19] http://en.wikipedia.org/wiki/Software_transactional_memory

[20] http://research.microsoft.com/~simonpj/papers/stm/

[21]http://www.haskell.org/haskellwiki/Software_transactional_memory.

First Author is currently a 3rd year student of B.E. (I.T.) in Jadavpur University, Kolkata, West Bengal, India.

Second Author is also a 3rd year student of B.E. (I.T.) in Jadavpur University, Kolkata, West Bengal, India.

Third Author is a research scholar in the Department of Information Technology, Jadavpur University, Kolkata, West Bengal, India.

Fourth Author is an Associate Professor in the Department of Information Technology, Jadavpur University, Kolkata, West Bengal, India.