RESEARCH ARTICLE OPEN ACCESS

Multiple Printer Problem with STM

Sayantan Sen, Ryan Saptarshi Ray, Hridam Basu, Utpal Kumar Ray, Parama Bhaumik

B. E (IT) 3rd Year Student Department of Information Technology, Jadavpur University Kolkata, India
Junior Research Fellow Department of Information Technology, Jadavpur University Kolkata, India
B. E (IT) 3rd Year Student Department of Information Technology, Jadavpur University Kolkata, India
Associate Professor Department of Information Technology, Jadavpur University Kolkata, India
Assistant Professor Department of Information Technology, Jadavpur University Kolkata, 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. It also shows that STM is easier to use than locks. This is because critical sections need not to be identified in case of STM. If we enclose the entire code with STM, then the performance does not decrease. But in case of lock, if we enclose the whole code within lock, performance drastically decreases.
Keywords- Parallel Programming; Multiprocessing; Locks; Transactions; Software Transactional Memory

I. 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**].

II. MULTIPLE PRINTER PROBLEM

There is a fixed amount of work. Initially, only one printer is used to perform this work. Now if the same work is performed using multiple printers, then time taken should proportionately decrease. But in case of multiple printers, as more than one printer may try to perform the same work at the same time, synchronization problems may occur. This paper shows how these synchronization problems can be overcome.

III. MULTIPLE PRINTER PROBLEM USING LOCKS

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 multiple printer problem,the printer thread is
"print_time()". The work which is being performed by the printer threads is showing the number the number of elements greater than 100 in the array " arr[] " .The following code snippet shows the printer thread :

```
void  *print_time(int * num_ptr)
{

        unsigned long j,a;

        int num,*number_ptr;

        number_ptr=num_ptr;

         num=*number_ptr;

        local_min_array[num]=255;


for((j=(((num*n)/NUM_THREAD)));j<(((num+1)*n)
/NUM_THREAD);j++)
 {
        if(arr[j]>100)
       {

pthread_mutex_lock(&mutex1);

                printf("%u",arr[j]);

  pthread_mutex_unlock(&mutex1);
         }

 }
}
```

The following code snippet shows how in the printer thread the elements greater than 100 in the array are being displayed:

```
  if(arr[j]>100)
  {

        pthread_mutex_lock(&mutex1);

        printf("%u",arr[j]);

        pthread_mutex_unlock(&mutex1);

   }
```

In case of multiple threads, the same printer thread is being called multiple times.
Three lock calls have been used in the code. These are shown below:

pthread_mutex_init(&mutex1,NULL) is used for lock initialization.

pthread_mutex_lock(&mutex1) is used for locking. This means that any thread which tries to access the critical section has to first acquire the lock on mutex1.

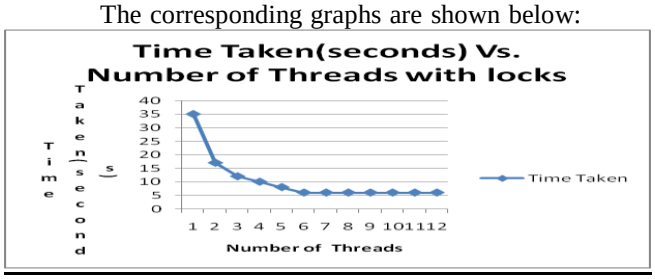
pthread_mutex_unlock(&mutex1) is used for unlocking.

In this program the region where more than one thread may access the global array arr[] at the same time is the critical section. Thus this region is enclosed within locks. Hence there is no synchronization problem in the code.

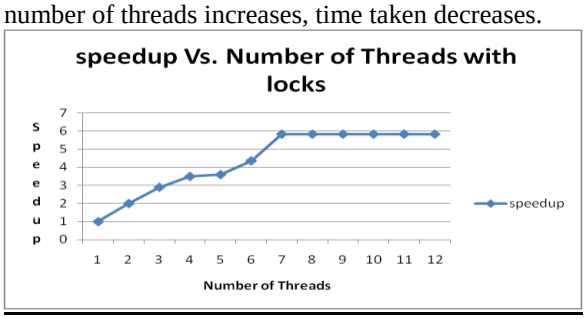
IV. EXPERIMENTAL RESULTS FOR MULTIPLE PRINTER PROBLEM USING LOCKS

No. of Threads	Time taken	Speedup	Efficiency
1	35	1	1
2	17	2	1
3	12	2.9	0.97
4	10	3.5	0.88
5	8	3.6	0.87
6	6	4.36	0.97
7	6	5.83	0.83
8	6	5.83	0.73
9	6	5.83	0.65
10	6	5.83	0.58
11	6	5.83	0.53
12	6	5.83	0.49

The corresponding graphs are shown below:



From the above graph we can see that as the number of threads increases, time taken decreases.



From the above graph we can see that as the number of threads increases, speedup also increases. But after a certain point, speedup becomes constant.

V. MULTIPLE PRINTER PROBLEM USING STM

In case of parallel programs using locks, certain problems like priority inversion, convoying and deadlocks occur. 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.

The following code snippet shows the printer thread :

```
void *print_time(int* num_ptr)

{
      unsigned long j;

      unsigned char byte_under_stm;

      int num,*number_ptr;

      number_ptr=num_ptr;

      num=*number_ptr;

      stm_init_thread( );

for((j=(((num*n)/NUM_THREAD)));j<(((num+1)*n)
/NUM_THREAD);j++)

      {

            if(arr[j]>100)
            {

                  START(0,RW);

                  byte_under_stm=(unsigned
char)LOAD(&arr[j]);

                printf("%u",byte_under_stm);

                  STORE(&arr[j], byte_under_stm);

                  COMMIT;
            }

      }
      stm_exit_thread( );
      pthread_exit(0);

}
```

The program structure is same as that of the program for multiple printer problem using threads and locks. The only difference is that STM is being used in this program.

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 TinySTM library. In this program it is called from the thread **print_time**.

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 thread **print_time**.

START(0,RW) is used to start a transaction. In this program it is used in the thread **print_time**.

COMMIT is used to close the transaction. In this program it is used in the thread **print_time**.

byte_under_stm=(unsigned char)LOAD(&arr[j]); stores the value of arr[j] in byte_under_stm. In this program it is used in the thread **print_time**.

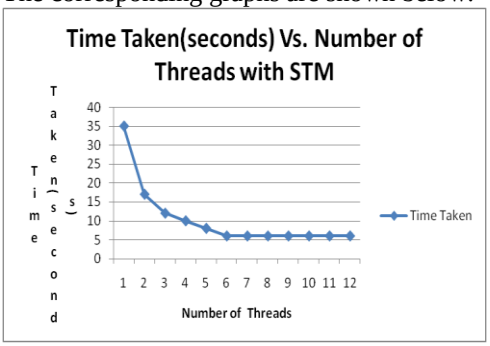
STORE(&arr[j], byte_under_stm) stores the value of byte_under_stm in arr[j]. In this program it is used in the thread **print_time**.

In this program the region where more than one thread may access the global array arr[] at the same time is the critical section. Thus this region is enclosed within transactions using TinySTM which is a type of STM. Hence there is no synchronization problem in the code.

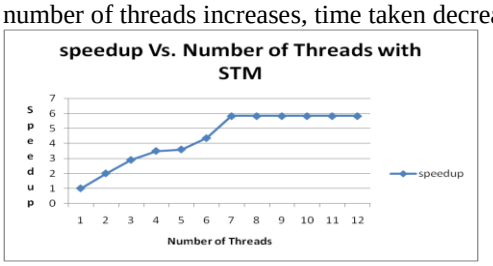
VI. EXPERIMENTAL RESULTS FOR MULTIPLE PRINTER PROBLEM USING STM

NO. of Threads	Time taken	speedup	Efficiency
1	35	1	1
2	17	2	1
3	12	2.9	0.97
4	10	3.5	0.88
5	8	3.6	0.87
6	6	4.36	0.97
7	6	5.83	0.83
8	6	5.83	0.73
9	6	5.83	0.65
10	6	5.83	0.58
11	6	5.83	0.53
12	6	5.83	0.49

The corresponding graphs are shown below:



From the above graph we can see that as the number of threads increases, time taken decreases.



From the above graph we can see that as the number of threads increases, speedup also increases. But after a certain point, speedup becomes constant.

VII. PERFORMANCE COMPARISON OF LOCKS AND STM

From the above experimental results, we can see that the performances of locks and STM are similar. We had enclosed only the critical sections of the codes within locks and STM. When we enclosed the entire code with STM, then the performance did not decrease. But in case of locks, when we enclosed the whole code within locks, performance drastically decreased**.** Hence we can say that performance of STM is better than that of locks.

VIII. CONCLUSION

STM has been shown in many ways to be a good alternative to using locks for writing parallel programs. STM provides a timetested 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. 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. This paper has also shown why STM is easier to use than locks and has also shown that the performance of STM is better than that of locks.

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] 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 Memory Support 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 of Nonblocking Software Transactional Memory".
[15] Utku Aydonat, Tarek S. Abdelrahman, Edward S. Rogers Sr., "Serializability of Transactions in Software 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] http://en.wikipedia.org/wiki/Software_transactional_memory
[19] http://research.microsoft.com/~simonpj/papers/stm/
[20] http://www.haskell.org/haskellwiki/Software_transactional_memory.
[21] Ryan Sapatarshi Ray,"Writing Lock Free Code Using Software Transactional Memory"