International Journal of Computer and Communication Technology

Volume 8 | Issue 2 Article 13

April 2017

Cigarette-Smokers' Problem with STM

Rup Kamal
Department of Information Technology, Jadavpur University Kolkata, India, rupkamal@gmail.com

Ryan Saptarshi Ray
Department of Information Technology Jadavpur University, Kolkata, India, ryan.ray@rediffmail.com

Utpal Kumar Ray
Department of Information Technology, Jadavpur University Kolkata, India, utpalkumarray@gmail.com

Parama Bhaumik
Department of Information Technology, Jadavpur University Kolkata, India, paramabhaumik@gmail.com

Cigarette-Smokers' Problem with STM

Rup Kamal, Ryan Saptarshi Ray, Utpal Kumar Ray & Parama Bhaumik

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. One such drawback is that locks should be only used to enclose the critical section of the parallel-processing code. If locks are used to enclose the entire code then the performance of the code drastically decreases.

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. If STM is used to enclose the entire code then the performance of the code is the same as that of the code in which STM is used to enclose the critical section only and is far better than code in which locks have been used to enclose the entire code. So STM is easier to use than locks as critical section does not need to be identified in case of STM.

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 why the use of STM in parallel-processing code is better than the use of locks.

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. CIGARETTE-SMOKERS' PROBLEM

In the cigarette-smokers' problem there are three smokers and one agent. There are three resources-tobacco, paper and matches. Each smoker has only one resource available at a time. The agent collects the three resources from the smokers and makes a cigarette and informs the smokers that a cigarette is ready. Then any one of the smokers smokes the cigarette and after finishing informs the agent. This process should continue without any synchronization problems.

III. CIGARETTE SMOKERS' 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 parallel program using threads and locks which solves the cigarette-smokers' problem there are four thread functions- one agent-"agent()" and three smokers-"smoke1()","smoke2()" and "smoke3()". Each resource is represented by a variable- paper(a), tobacco(b) and matches(c).

The following code snippet shows the **agent** thread:

```
void *agent(int *num_ptr)
{
unsigned long j;
int num,*number_ptr;
number_ptr=num_ptr;
num=*number_ptr;
pthread_mutex_lock(&mutex1);
a++;b++;c++;
pthread_mutex_unlock(&mutex1);
for((j=(((num*n)/(NUM_THREAD))));j<(((num+1)*n)/(NUM_THREAD));j++)
{
arr[j]=d+3;
}
pthread_exit(0);
}
```

In the thread "agent" when the agent accesses a resource then the corresponding variable is incremented.

```
pthread_mutex_lock(&mutex1);
a++; b++; c++;
pthread_mutex_unlock(&mutex1);
```

The following code snippet shows the **smoke1** thread:

```
void *smoke1(int *num_ptr)
{
if(a>0&&b>0&&c>0&&(a==b)&&(b==c)&&(a==c))
{ pthread_mutex_lock(&mutex2);
s++;
pthread_mutex_unlock(&mutex2);
}
pthread_exit(0);
}
```

The thread functions **smoke2** and **smoke3** are similar in structure to smoke1. In the threads "smoke1()","smoke2()" and "smoke3()" when any smoker smokes the global variable s is incremented.

The following statement is used to record the time before the threads are created:

```
gettimeofday(&ini_tv,NULL);
```

The following statement is used to record the time when all threads have just finished their executions:

```
gettimeofday(&final_tv,NULL);
```

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);
```

12 lock calls are being used in the program.

pthread_mutex_init(&mutex1,NULL), pthread_mutex_init(&mutex2,NULL), pthread_mutex_init(&mutex3,NULL) and **pthread_mutex_init(&mutex4,NULL)** are used for lock initialization.

pthread_mutex_lock(&mutex1), pthread_mutex_lock(&mutex2), pthread_mutex_lock(&mutex3) **and pthread_mutex_lock(&mutex4)** are used for locking.

pthread_mutex_unlock(&mutex1), pthread_mutex_unlock(&mutex2), pthread_mutex_unlock(&mutex3) **and pthread_mutex_unlock(&mutex4)** are used for unlocking.

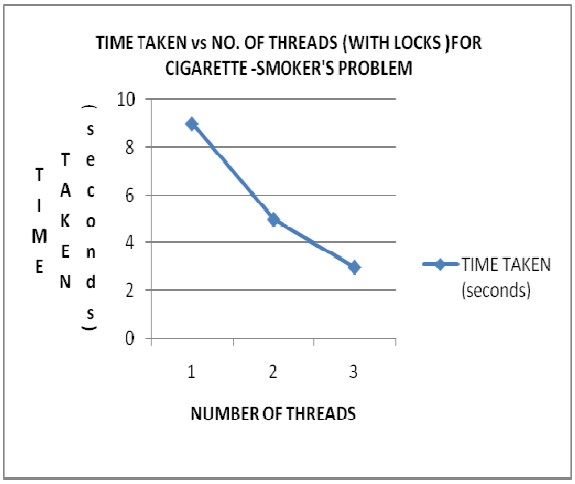
In the program the regions where more than one thread may access the global variables a,b,c and s at the same time are the critical sections. Thus these regions are enclosed within locks. Hence there is no synchronization problem in the above code.

IV. EXPERIMENTAL RESULTS FOR CIGARETTE-SMOKERS' PROBLEM USING LOCKS

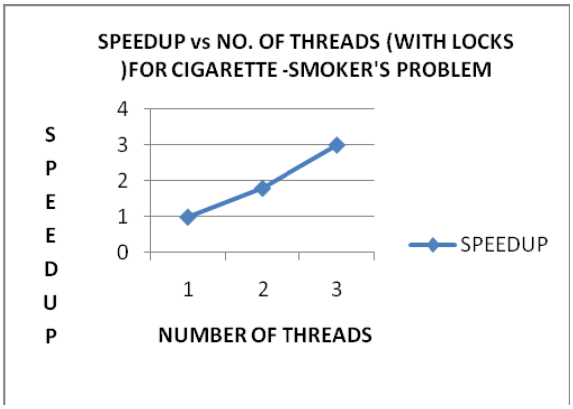
The following table shows the experimental results for cigarette-smokers' problem using locks:

NUMBER OF THREADS	TIME TAKEN(seconds)	SPEEDUP	EFFICIENCY
1	9	1	1
2	5	1.8	0.9
3	3	3	1

The corresponding graphs for the above experimental results are shown below:



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



From the above graph we can see that as the number of threads increases the speedup also steadily increases.

V. Cigarette-Smokers' Problem Using STM

Use of locks in parallel processing code leads to some drawbacks. The problems of priority inversion, deadlocks and convoying occur while performing synchronization using locks. [**4**], [**10**]. 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 structure of the program using threads and STM which solves the cigarette-smokers' problem is same as that of the program using threads and locks. The only difference is that STM is being used in this program.

The following code snippet shows the **agent** thread:

```
void *agent(int *num_ptr)
{
unsigned long j;
unsigned char byte_under_stm;
unsigned char byte_under_stm1;
unsigned char byte_under_stm2;

 int num,*number_ptr;
 number_ptr=num_ptr;
 num=*number_ptr;
stm_init_thread();

 START(0,RW);
    byte_under_stm=(unsigned char) LOAD(&a);
    byte_under_stm1=(unsigned char) LOAD(&b);
    byte_under_stm2=(unsigned char) LOAD(&c);
byte_under_stm++;
byte_under_stm1++;
byte_under_stm2++;
  STORE(&a,byte_under_stm);
STORE(&b,byte_under_stm1);
STORE(&c,byte_under_stm2);

 COMMIT;

for((j=(((num*n)/(NUM_THREAD))));j<(((num+1)*n)/
(NUM_THREAD));j++)
    {
arr[j]=d+3;
} stm_exit_thread();
    pthread_exit(0);
}
```

The following code snippet shows the **smoke1** thread:

```
 void *smoke1(int *num_ptr)
{
stm_init_thread();
unsigned char byte_under_stm;
if(a>0&&b>0&&c>0&&(a==b)&&(b==c)&&(a==c))
{ START(0,RW);
byte_under_stm=(unsigned char) LOAD(&s);
byte_under_stm++;
   STORE(&s,byte_under_stm);
   COMMIT;
}
stm_exit_thread();

 pthread_exit(0);
}
```

The thread functions **smoke2** and **smoke3** are similar in structure to **smoke1.**

The STM functions and calls which have been used in the code are explained below:

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 threads **agent, smoke1, smoke2** and **smoke3**.

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 **agent, smoke1, smoke2** and **smoke3**.

START(0,RW) is used to start a transaction. In this program it is used in the threads **agent, smoke1, smoke2** and **smoke3**.

COMMIT is used to close the transaction. In this program it is used in the threads **agent, smoke1, smoke2** and **smoke3**.

byte_under_stm=(unsigned char) LOAD(&a) stores the value of a in byte_under_stm. In this program it is used in the thread **agent**.

byte_under_stm1=(unsigned char) LOAD(&b)

stores the value of b in byte_under_stm1. In this program it is used in the thread **agent**.

byte_under_stm2=(unsigned char) LOAD(&c) stores the value of c in byte_under_stm2. In this program it is used in the thread **agent**.

byte_under_stm=(unsigned char) LOAD(&s)

stores the value of s in byte_under_stm. In this program it is used in the threads **smoke1, smoke2 and**.**smoke3.**

STORE(&a,byte_under_stm)

stores the value of byte_under_stm in a. In this program it is used in the thread **agent**.

STORE(&b,byte_under_stm1)

stores the value of byte_under_stm1 in b. In this program it is used in the thread **agent**.

STORE(&c,byte_under_stm2)

stores the value of byte_under_stm2 in c. In this program it is used in the thread **agent**.

STORE(&s,byte_under_stm)

stores the value of byte_under_stm in s. In this program it is used in the threads **smoke1, smoke2** and **smoke3**.

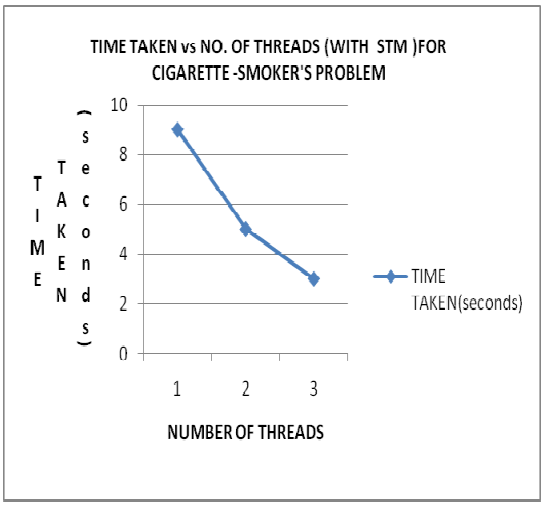
In this program the regions where more than one thread may access the global variables count and a,b,c and s at the same time are the critical sections. Thus these regions are enclosed within transactions using TinySTM which is a type of STM. Hence there is no synchronization problem in the above code.

VI. Experimental Results For Cigarette-Smokers' Problem Using STM

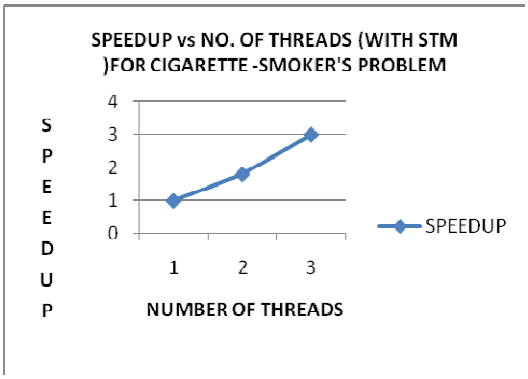
The following table shows the experimental results for cigarette-smokers' problem using STM :

NUMBER OF THREADS	TIME TAKEN(seconds)	SPEEDUP	EFFICIENCY
1	9	1	1
2	5	1.8	0.9
3	3	3	1

The corresponding graphs for the above experimental results are shown below:



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



From the above graph we can see that as the number of threads increases the speedup also steadily increases.

VII. Performance Comparision of Locks and STM

From the above experimental results we see that performance of locks and STM are similar.

In the code with locks we have enclosed only the critical section with locks. When we enclosed the entire code with locks then the performance drastically decreased. In the code with STM also we have enclosed only the critical section with STM. When we enclosed the entire code with STM then also the performance remained same. So it can be said that performance of STM is better than that of locks. Also we can say that STM is easier to use than locks as critical section need not be identified in case of STM.

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.It has also been shown that STM is easier to use than locks as critical section need not be identified explicitly in case of STM. In case of STM if the entire code is enclosed within STM the performance of the code is same as that of the code in which only the critical section is enclosed within STM.

But in case of locks if the entire code is enclosed within locks then the performance sharply decreases. So it has been shown that performance of STM is much 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.

IX. 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 Saptarshi Ray, "Writing Lock-Free Code using Software Transactional Memory ".

[22] Ryan Saptarshi Ray,Utpal Kumar Ray "Different Approaches for improving performance of Software Transactional Memory ".

[23] John H.Reynolds, "Solving The Cigarette Smokers Problem Using Uniprocessor Concurrency And True Parallelism' ".

◈◈◈