IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-5, May 2019 E-ISSN: 2347-2693

A Study and Analysis of Lock and STM Overheads

Ryan Saptarshi Ray[1*], Parama Bhaumik[2], Utpal Kumar Ray[3]

[1,2,3]Dept. of Information Technology, Jadavpur University, Kolkata, India

[*]*Corresponding Author: ryan.ray@rediffmail.com, Tel.: 9831520613*

DOI: https://doi.org/10.26438/ijcse/v7i5.548556 | Available online at: www.ijcseonline.org

Accepted: 18/May/2019, Published: 31/May/2019

***Abstract*—** In this paper we make a comparative study of the overheads of locks and STM by taking different practical synchronization problems as examples to understand why the performance of STM is worse than that of locks. Overhead is the combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task. While executing parallel programs whenever any lock or STM function is called it takes some time and also occupies some space. The total time taken by all the lock or STM calls of the program is the total lock or STM time overhead of that program. The total space occupied by all the lock or STM calls of the program is the total lock or STM space overhead of that program. The flexible approach is an approach of programming with STM by which STM has been made more user-friendly and by which execution time of STM has been reduced. We make a study of the overheads of the flexible approach also. We found that the time and space overheads of STM are higher than that of locks. The time and space overheads of the Flexible Approach were less than those of STM but higher than those of locks.

***Keywords*—** Multiprocessing, Parallel Processing, Locks, Software Transactional Memory, Overheads

I. INTRODUCTION

Overhead is the combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task **[1]**.

Software Transactional Memory (STM) is a new approach for solving synchronization problems in parallel programs that does not suffer from the drawbacks of locks. However performance of STM is either equal to or worse than that of locks. In this paper we make a comparative study of the overheads of locks and STM to understand why this happens.

While executing parallel programs whenever any lock or STM function is called it takes some time and also occupies some space. The total time taken by all the lock or STM calls of the program is the total lock or STM time overhead of that program. The total space occupied by all the lock or STM calls of the program is the total lock or STM space overhead of that program.

The flexible approach is an approach of programming with STM by which STM has been made more user-friendly and by which execution time of STM has been reduced. We make a study of the overheads of the flexible approach also.

We found that the time and space overheads of STM are higher than that of locks. The time and space overheads for the Flexible Approach were less than those of STM but higher than those of locks.

Section II discusses about different approaches which have been proposed to improve the performance of STM. Section III shows the time overhead for locks and STM for different practical synchronization problems. Section IV shows the space overhead for locks and STM for different practical synchronization problems. Section V shows the time overhead for the Flexible Approach for different practical synchronization problems. Section VI shows the space overhead for the Flexible Approach for different practical synchronization problems. Section VII makes a comparison of the overheads for locks, STM and the Flexible Approach. Section VIII shows the specifications of the system in which the programs were compiled and executed. Section IX concludes the paper.

II. RELATED WORK

Different approaches have been proposed to improve the performance of STM. These are discussed below.

In 2007 Yang Ni, Vijay Menon, Richard L. Hudson, Ali-Reza Adl-Tabatabai, J. Eliot, B. Moss, Bratin Saha, Antony L. Hosking, Tatiana Shpeisman published a paper entitled “Open Nesting in Software Transactional Memory” **[2]**. This paper described new language constructs to support open nesting in Java and also discussed new abstract locking mechanisms that a programmer could use to prevent logical conflicts. In 2009 Zhengyu He and Bo Hong published a paper entitled “Impact of Early Abort Mechanisms on Lock-Based Software Transactional Memory” **[3]**. This paper adopted Queuing theory to model the behaviors of transactional execution. Also in 2009 Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum and Marek Olszewski published a paper entitled “Anatomy of a Scalable Software Transactional Memory” **[4]**. This paper described novel techniques to eliminate bottlenecks from existing STM mechanisms and presented SkySTM. In 2010 Justin E. Gottschlich, Manish Vachharajani, Jeremy G. Siek published a paper entitled “An Efficient Software Transactional Memory Using Commit-Time Invalidation”. This paper presented an efficient implementation of committime invalidation, a strategy where transactions resolved their conflicts with in-flight (uncommitted) transactions before they commited **[5]**. In 2011 Sandhya S.Mannarswamy and Ramaswamy Govindarajan published a paper entitled “Variable Granularity Access Tracking Scheme for Improving the Performance of Software Transactional Memory” **[6]**. In order to mitigate the disadvantages associated with Uniform Granularity Access Tracking (UGAT) scheme, this paper proposed a Variable Granularity Access Tracking (VGAT) scheme.

In our work we have made a comparative study of the overheads of locks, STM and the flexible approach. This is because by reducing the overheads of STM its performance can be improved.

III. TIME OVERHEAD FOR LOCKS AND STM

While executing parallel programs whenever any lock or STM function is called it takes some time. Thus the sum of the time taken by all the lock or STM calls of a program is the total lock or STM time overhead of that program. Some lock calls are:-

i) pthread_mutex_lock(&mutex1)- Any thread must acquire the lock on the variable **mutex1** to execute the critical section following this function.
ii)pthread_mutex_unlock(&mutex1)- This function is used for unlocking.

The execution time of both these functions is 2 microseconds.

Some STM Calls are:-

i)byte_under_stm=(unsigned char)LOAD(&global_min)- It stores the value of **global_min** in **byte_under_stm**.
ii)STORE(&global_min, byte_under_stm)- It stores the value of **byte_under_stm** in **global_min.**

The execution time of both these functions is 2 microseconds.

Start of transaction in case of STM takes 4 microseconds and commit of transactions takes 2 microseconds.

The time overheads for lock and STM for some different practical synchronization problems are shown now.

Finding minimum element in an array

In the program for finding minimum element in an array using locks in each thread there is one set of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls. So for each thread the time overhead is 4 microseconds**[7]**.

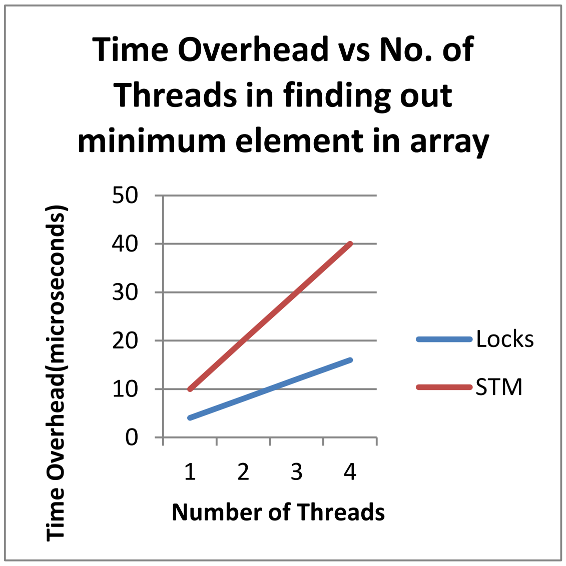
In the program for finding minimum element in an array using STM for each thread there is one transaction and one LOAD and STORE call. Thus for each thread the time overhead is 10(4+2+2+2) microseconds.

The table below shows the time taken for finding minimum element in an array in case of different numbers of threads for both locks and STM.

Table.1 Time Overhead for finding minimum element in an array

No. of Threads	Time Overhead(microseconds)(Locks)	Time Overhead(microseconds)(STM)
1	4	10
2	8	20
3	12	30
4	16	40

The graph below shows the comparison of time overheads for locks and STM for finding minimum element in an array.



Readers-Writers Problem

In the program for solving Readers-Writers Problem using locks there are two sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every Reader-Writer pair. So for each Reader-Writer pair the time overhead is 8 microseconds **[8]**.

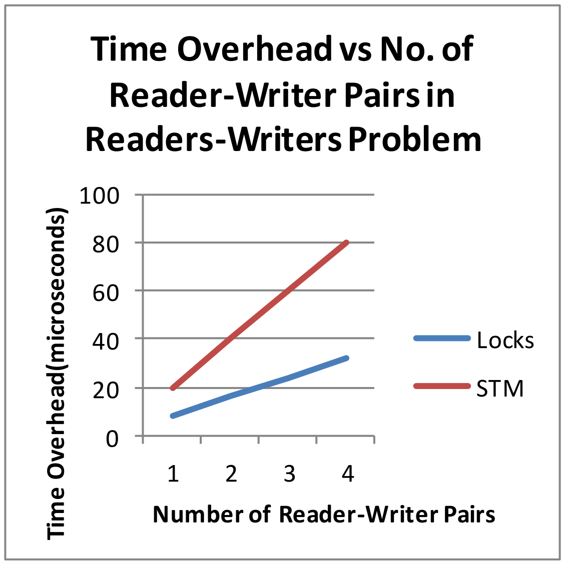
In the program for solving Readers-Writers Problem using STM for each Reader-Writer pair there are two transactions and two LOAD and STORE calls. Thus for each Reader-Writer pair the time overhead is 2x10(4+2+2+2)=20 microseconds.

The table below shows the time taken for solving Readers-Writers Problem in case of different numbers of Reader-Writer Pairs for both locks and STM.

Table.2 Time Overhead for Readers-Writers Problem

No. of Reader-Writer Pairs	Time Overhead(microseconds)(Locks)	Time Overhead(microseconds)(STM)
1	8	20
2	16	40
3	24	60
4	32	80

The graph below shows the comparison of time overheads for locks and STM in Readers-Writers Problem.



Dining Philosophers' Problem

In the program for solving Dining Philosophers' Problem using locks there are two sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every thread. So for each thread the time overhead is 8 microseconds.

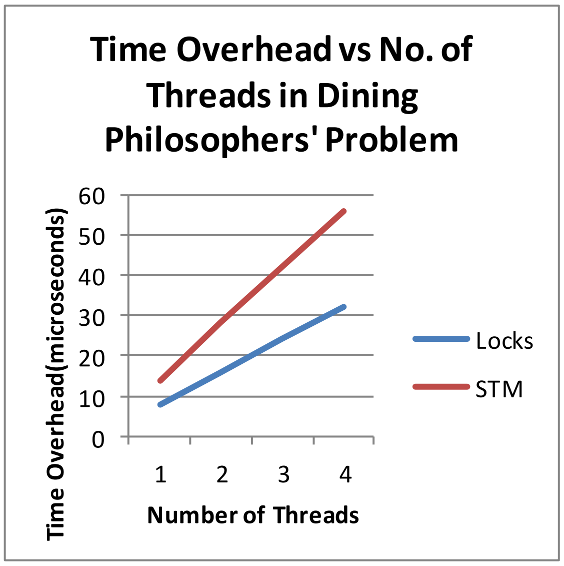
In the program for solving Dining Philosophers' Problem using STM for each thread there is one transaction, two LOAD and two STORE calls. Thus for each thread the time overhead is 4+2+4+4=14 microseconds **[9]**.

The table below shows the time taken for solving Diming Philosophers' Problem in case of different numbers of threads for both locks and STM.

Table.3 Time Overhead for Dining Philosophers' Problem

No. of Threads	Time Overhead(microseconds)(Locks)	Time Overhead(microseconds)(STM)
1	8	14
2	16	28
3	24	42
4	32	56

The graph below shows the comparison of time overheads for locks and STM in Dining Philosophers' Problem.



Cigarette Smokers' Problem

In the program for solving Cigarette Smokers' Problem using locks there are four sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every Agent-Smoker set. So for each Agent-Smoker set the time overhead is 16 microseconds.

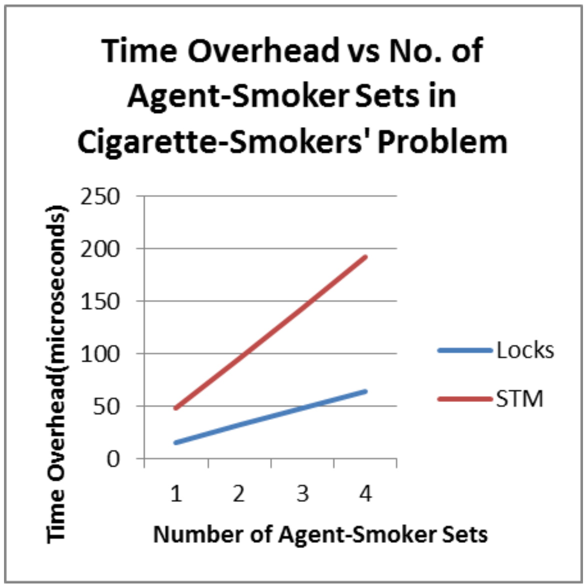
In the program for solving Cigarette Smokers' Problem using STM for each Agent-Smoker Set there are four transactions, six LOAD and six STORE calls. Thus for each Agent-Smoker Set the time overhead is 16+8+12+12=48 microseconds **[10]**.

The table below shows the time taken for solving Cigarette-Smokers' Problem in case of different numbers of Agent-Smoker sets for both locks and STM.

Table.4 Time Overhead for Cigarette-Smokers' Problem

No. of Agent-Smoker Sets	Time Overhead(microseconds)(Locks)	Time Overhead(microseconds)(STM)
1	16	48
2	32	96
3	48	144
4	64	192

The graph below shows the comparison of time overheads for locks and STM in Cigarette-Smokers' Problem.



So from the above observations we can say that the time overhead of STM in all the applications is higher than that of locks.

In case of locks every critical section has to be enclosed within one set of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls irrespective of the number of global variables being accessed in the critical section. In case of STM each critical section has to be enclosed within a transaction. Start and commit of transactions take time. Then for each global variable being accessed within a transaction there has to be one set of LOAD and STORE calls.

So even though the time overhead of individual lock and STM function calls are almost same the time overhead of STM is higher as more number of function calls have to be used in case of STM than locks and also as there is additional time overhead for start and commit of transactions.

IV. SPACE OVERHEAD FOR LOCKS AND STM

While executing parallel programs whenever any lock or STM function is called it occupies some space. Thus the sum of the space occupied by all the lock or STM calls of a program is the total lock or STM space overhead of that program.

The space occupied by **pthread_mutex_lock(&mutex1) and pthread_mutex_unlock(&mutex1)** are 4 bytes each.

The space occupied by **stm_load() and stm_store()** are 11 bytes and 88 bytes respectively.

The space overheads for lock and STM for some different practical synchronization problems are shown now.

Finding minimum element in an array

In the program for finding minimum element in an array using locks for each thread there is one set of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls. So for each thread the space overhead is 8 bytes.

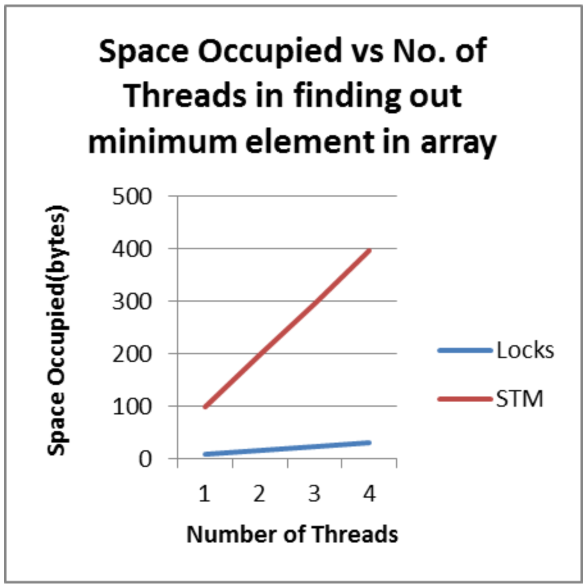
In the program for finding minimum element in an array using STM for each thread there is one LOAD and STORE call. Thus for each thread the space overhead is 99 bytes.

The table below shows the space occupied for finding minimum element in an array in case of different numbers of threads for both locks and STM.

Table.5 Space Overhead for finding minimum element in an array

No. of Threads	Space Overhead(bytes)(Locks)	Space Overhead(bytes)(STM)
1	8	99
2	16	198
3	24	297
4	32	396

The graph below shows the comparison of space overheads for locks and STM for finding minimum element in an array.



Readers-Writers Problem

In the program for solving Readers-Writers Problem using locks there are two sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every Reader-Writer pair (one set for reader( ) and one set for writer( )). So for each Reader-Writer pair the space overhead is 16 bytes.

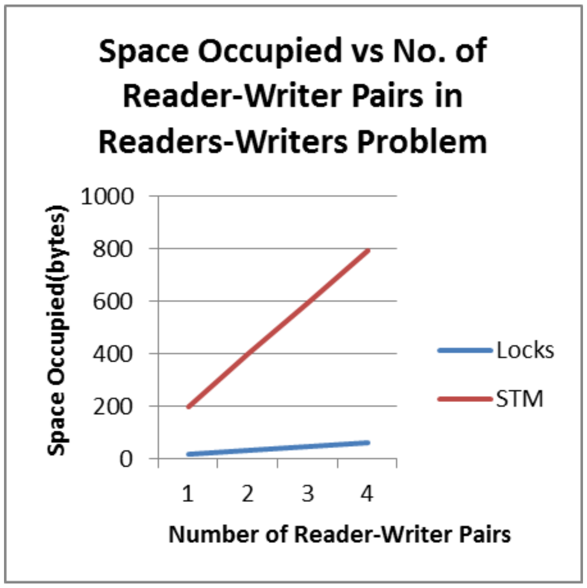
In the program for solving Readers-Writers Problem using STM for each Reader-Writer pair there are two LOAD and STORE calls. Thus for each Reader-Writer pair the space overhead is 198 bytes.

The table below shows the space occupied for solving Readers-Writers Problem in case of different numbers of Reader-Writer Pairs for both locks and STM.

Table.6 Space Overhead for Readers-Writers Problem

No. of Reader-Writer Pairs	Space Overhead(bytes)(Locks)	Space Overhead(bytes)(STM)
1	16	198
2	32	396
3	48	594
4	64	792

The graph below shows the comparison of space occupied for locks and STM in Readers-Writers Problem.



Dining Philosophers' Problem

In the program for solving Dining Philosophers' Problem using locks there are two sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every thread. So for each thread the space overhead is 16 bytes.

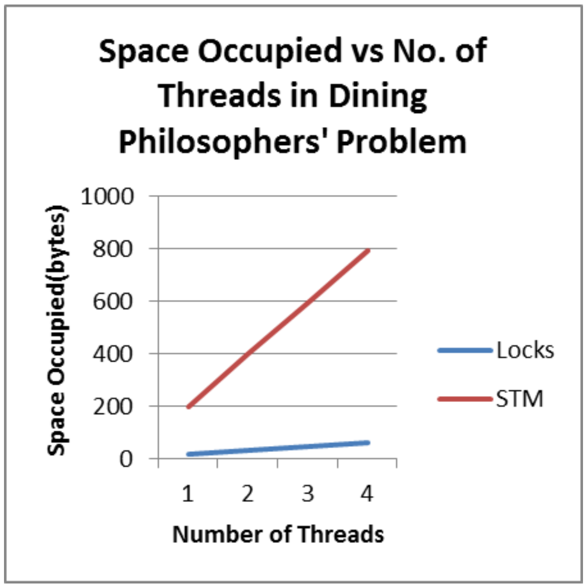
In the program for solving Dining Philosophers' Problem using STM for each thread there are two LOAD and STORE calls. Thus for each thread the space overhead is 198 bytes.

The table below shows the space occupied for solving Dining Philosophers' Problem in case of different numbers of threads for both locks and STM.

Table.7 Space Overhead for Dining Philosophers' Problem

No. of Threads	Space Overhead(bytes)(Locks)	Space Overhead(bytes)(STM)
1	16	198
2	32	396
3	48	594
4	64	792

The graph below shows the comparison of space overheads for locks and STM in Dining Philosophers' Problem.



Cigarette Smokers' Problem

In the program for solving Cigarette Smokers' Problem using locks there are four sets of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls for every Agent-Smoker set. So for each Agent-Smoker set the space overhead is 32 bytes.

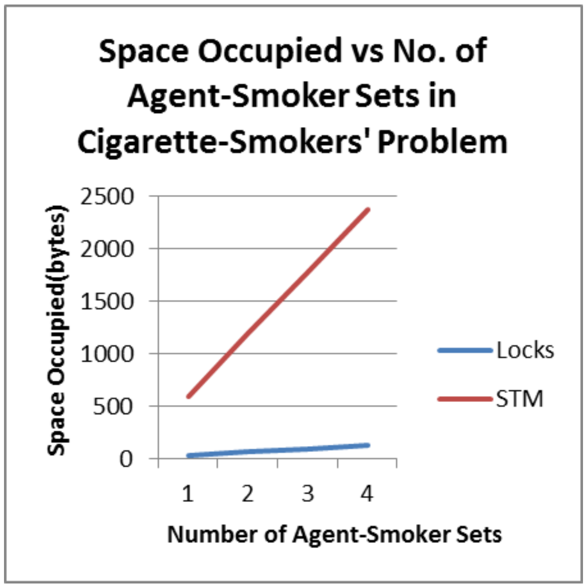
In the program for solving Cigarette Smokers' Problem using STM for each Agent-Smoker set there are six LOAD and STORE calls. Thus for each thread the space overhead is 594 bytes.

The table below shows the space occupied for solving Cigarette-Smokers' Problem in case of different numbers of Agent-Smoker sets for both locks and STM.

Table.8 Space Overhead for Cigarette-Smokers' Problem

No. of Agent-Smoker Sets	Space Overhead(bytes)(Locks)	Space Overhead(bytes)(STM)
1	32	594
2	64	1188
3	96	1782
4	128	2376

The graph below shows the comparison of space overheads for locks and STM in Cigarette-Smokers' Problem.



So from the above observations we can say that the space overhead of STM in all the applications is higher than that of locks.

This is because the space overhead of individual STM function calls are higher than that of locks and also as more number of function calls have to be used in case of STM than locks.

V. TIME OVERHEAD FOR FLEXIBLE APPROACH

The STM calls used in case of Flexible Approach are:-

i) stm_unit_load( )- It reads the specified memory location outside of the context of any transaction and returns its value. The operation behaves as if executed in the context of a dedicated transaction (i.e., it executes atomically and in isolation) that never aborts, but may get delayed.

ii) stm_unit_store( )- It writes a value to the specified memory location outside of the context of any transaction. It also behaves as if executed in the context of a dedicated transaction (i.e., it executes atomically and in isolation) that never aborts, but may get delayed.

Both these function calls take 2 microseconds to execute.

The time overheads for Flexible Approach for the different programs are shown now.

Finding minimum element in an array

In this program for each thread there is one stm_unit_load( ) and stm_unit_store( )call. Thus for each thread the time overhead is 4 microseconds.

The table below shows the time taken for finding minimum element in an array using flexible approach in case of different numbers of threads.

Table.9 Time Overhead for finding minimum element in an array using Flexible Approach

No. of Threads	Time Overhead(microseconds)
1	4
2	8
3	12
4	16

Readers-Writers Problem

In this program for each Reader-Writer pair there are two stm_unit_load( ) and stm_unit_store( )calls. Thus for each Reader-Writer pair the time overhead is 8 microseconds.

The table below shows the time taken for solving Readers-Writers Problem using Flexible Approach in case of different numbers of Reader-Writer Pairs.

Table.10 Time Overhead for Readers-Writers Problem using Flexible Approach

No. of Reader-Writer Pairs	Time Overhead(microseconds)
1	8
2	16
3	24
4	32

Dining Philosophers' Problem

In this program for each thread there are two stm_unit_load( ) and stm_unit_store( )calls. Thus for thread the time overhead is 8 microseconds.

The table below shows the time taken for solving Dining Philosophers' Problem using flexible approach in case of different numbers of threads.

Table.11 Time Overhead for Dining Philosophers' Problem using Flexible Approach

No. of Threads	Time Overhead(microseconds)

1	8
2	16
3	24
4	32

Cigarette Smokers' Problem

In this program for each Agent-Smoker Set there are six stm_unit_load( ) and stm_unit_store( )calls. Thus for each Agent-Smoker Set the time overhead is 24 microseconds.

The table below shows the time taken for solving Cigarette-Smokers' Problem using Flexible Approach in case of different numbers of Agent-Smoker sets.

Table.12 Time Overhead for Cigarette-Smokers' Problem using Flexible Approach

No. of Agent-Smoker Sets	Time Overhead(microseconds)
1	24
2	48
3	72
4	96

From the above observations we can say that the time overhead for the Flexible Approach is less than the time overhead for STM and in some cases same as the time overhead for locks. This is because in case of the Flexible Approach there is no need to enclose the critical sections within transactions.

VI. SPACE OVERHEAD FOR FLEXIBLE APPROACH

The space occupied by stm_unit_load( ) and stm_unit_store( ) are 11 bytes and 44 bytes respectively.

The space overheads for Flexible Approach for the different programs are shown now.

Finding minimum element in an array

In this program for each thread there is one stm_unit_load( ) and stm_unit_store( )call. Thus for each thread the space occupied is 55 bytes.

The table below shows the space occupied for finding minimum element in an array using flexible approach in case of different numbers of threads.

Table.13 Space Overhead for finding minimum element in an array using Flexible Approach

No. of Threads	Space Occupied(bytes)
1	55
2	110
3	165
4	220

Readers-Writers Problem

In this program for each Reader-Writer pair there are two stm_unit_load( ) and stm_unit_store( )calls. Thus for each Reader-Writer pair the space occupied is 110 bytes.

The table below shows the space occupied for solving Readers-Writers Problem using Flexible Approach in case of different numbers of Reader-Writer Pairs.

Table.14 Space Overhead for Readers-Writers Problem using Flexible Approach

No. of Reader-Writer Pairs	Space Occupied(bytes)
1	110
2	220
3	330
4	440

Dining Philosophers' Problem

In this program for each thread there are two stm_unit_load( ) and stm_unit_store( )calls. Thus for each thread the space overhead is 110 bytes.

The table below shows the space occupied for solving Dining Philosophers' Problem using flexible approach in case of different numbers of threads.

Table.15 Space Overhead for Dining Philosophers' Problem using Flexible Approach

No. of Threads	Space Occupied(bytes)
1	110
2	220
3	330
4	440

Cigarette Smokers' Problem

In this program for each Agent-Smoker Set there are six stm_unit_load( ) and stm_unit_store( )calls. Thus for each Agent-Smoker Set the space occupied is 330 bytes.

The table below shows the space occupied for solving Cigarette-Smokers' Problem using Flexible Approach in case of different numbers of Agent-Smoker sets.

Table.16 Space Overhead for Cigarette-Smokers' Problem using Flexible Approach

No. of Agent-Smoker Sets	Space Occupied(bytes)
1	330
2	660
3	990
4	1320

From the above observations we can say that the space occupied for the Flexible Approach is less than the space occupied for STM but higher than the space occupied for locks. This is because even though in case of the Flexible Approach there is no need to enclose the critical sections within transactions the space occupied by the individual function calls are higher than that of locks.

VII. OVERHEAD COMPARISON OF LOCKS, STM AND FLEXIBLE APPROACH

The time overhead of STM in all the applications is higher than that of locks. In case of locks every critical section has to be enclosed within one set of pthread_mutex_lock( ) and pthread_mutex_unlock( ) calls irrespective of the number of global variables being accessed in the critical section. In case of STM each critical section has to be enclosed within a transaction. Start and commit of transactions take time. Then for each global variable being accessed within a transaction there has to be one set of LOAD and STORE calls. So even though the time overhead of individual lock and STM function calls are almost same the time overhead of STM is higher as more number of function calls have to be used in case of STM than locks and also as there is additional time overhead for start and commit of transactions.

The space overhead of STM in all the applications is also higher than that of locks. This is because the space overhead of individual STM function calls are higher than that of locks and also as more number of function calls have to be used in case of STM than locks.

The time overhead for the Flexible Approach is less than the time overhead for STM and in some cases same as the time overhead for locks. This is because in case of the Flexible Approach there is no need to enclose the critical sections within transactions.

The space occupied for the Flexible Approach is less than the space occupied for STM but higher than the space occupied for locks. This is because even though in case of the Flexible Approach there is no need to enclose the critical sections within transactions the space occupied by the individual function calls are higher than that of locks.

In earlier works we had seen that the execution time of codes with STM were either equal to or worse than that of locks. We had also seen that in case of the flexible approach using STM the time taken was somewhat less **[7], [8], [9], [10], [11]**. The comparison of the overheads have confirmed these observations.

VIII. SYSTEM SPECIFICATIONS

The specifications of the system in which we compiled and executed the codes are given below:

SYSTEM DESCRIPTION

1. Hardware Configuration

Model Name: Intel® Xeon ® CPU E5645 2.40 GHz
Number of CPU cores: 6
Total Memory Space: 4.008 GB
Cache: 12288KB

2. Operating System

Fedora 11

3. Software Configuration

1) The language used in the programs is C.
2) gcc compiler version 4.4.0.

IX. CONCLUSION

The time overhead of STM in all the applications is higher than that of locks. The space overhead of STM in all the applications is also higher than that of locks. The time overhead for the Flexible Approach is less than the time overhead for STM and in some cases same as the time overhead for locks. The space occupied for the Flexible Approach is less than the space occupied for STM but higher than the space occupied for locks. In earlier works we had seen that the execution time of codes with STM were either equal to or worse than that of locks. We had also seen that in case of the flexible approach using STM the time taken was somewhat less. The comparison of the overheads have confirmed these observations.

REFERENCES

[1] Cornelia Cecilia Eglantine,"*Overhead(Computing)*", published by TypPress, **2012**

[2] Yang Ni, Vijay Menon, Richard L. Hudson, Ali-Reza Adl-Tabatabai, J. Eliot, B. Moss, Bratin Saha, Antony L. Hosking, Tatiana Shpeisman, "*Open Nesting in Software Transactional Memory*", In the Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. **68-78**, **2007**

[3] Zhengyu He, Bo Hong, "*Impact of Early Abort Mechanisms on Lock-Based Software Transactional Memory*", In the Proceedings of the International Conference on High Performance Computing (HiPC), **2009**

[4] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, Marek Olszewski, "*Anatomy of a Scalable Software Transactional Memory*", In the Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing , **2009**

[5] Justin E. Gottschlich, Manish Vachharajani, Jeremy G. Siek, "*An Efficient Software Transactional Memory Using Commit-Time Invalidation*", In the Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization , pp. **101-110, 2010**

[6] Sandhya S.Mannarswamy, Ramaswamy Govindarajan, "*Variable Granularity Access Tracking Scheme for Improving the Performance of Software Transactional Memory*", In the Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. **232-242**, **2011**

[7] Ryan Saptarshi Ray, "*STM:Lock-Free Synchronization*", Special Issue of IJCCT, ISSN (*ONLINE*): 2231 – 0371, ISSN (*PRINT*): 0975 – 7449, Volume- **3**, Issue-**2**, pp. **19-25**, **2012**

[8] Anupriya Chakraborty, Sourav Saha, Ryan Saptarshi Ray, Utpal Kumar Ray," *Lock-Free Readers/Writers*", International Journal of Computer Science Issues (IJCSI), ISSN (*PRINT*): 1694 – 0814, ISSN (*ONLINE*): 1694 – 0784, Volume- **10**, Issue-**4**, No-**2,** pp. **180-186, 2013**

[9] Venkatakash Raj Rao Jillelamudi, Sourav Mukherjee, Ryan Saptarshi Ray, Utpal Kumar Ray," *Lock-Free Dining Philosopher*", International Journal of Computer and Communication Technology (IJCCT), ISSN (*PRINT*): 0975 – 7449, Volume- **4**, Issue-**3**, pp. **54-58, 2013**

[10] Rup Kamal, Ryan Saptarshi Ray, Utpal Kumar Ray, Parama Bhaumik, "*Cigarette Smokers' Problem with STM*", International Journal of Computer and Communication Technology (IJCCT), ISSN (*PRINT*): 0975 – 7449, Volume- **4**, Issue-**3**, pp **63-68, 2013**

[11] Ryan Saptarshi Ray, Parama Bhaumik, Utpal Kumar Ray, "*Flexible Programming Approach using STM*", International Journal of Computer Sciences and Engineering (IJCSE) E-ISSN:2347-2693 Volume- **6**, Issue-**7**, pp. **349-353**, **2018**

[12] Nilam Choudhary, Shikhar Agarwal, Geerija Lavania, "*Smart Voting System through Facial Recognition*", International Journal of Scientific Research in Computer Sciences and Engineering (ISSN: 2320-7639) Volume- **7**, Issue-**2**, pp. **7-10**, **2019**

[13] S. JabeenBegum, B. Swaathi, "*A Survey for identifying Parkinson's disease by Binary Bat Algorithm*", International Journal of Scientific Research in Computer Sciences and Engineering (ISSN: 2320-7639) Volume- **7**, Issue-**2**, pp. **17-23**, **2019**

Authors Profile

Ryan Saptarshi Ray received the degree of B.E. in I.T. from School of Information Technology, West Bengal University of Technology, India in 2007.

He received the degree of M.E. in Software Engineering from Jadavpur University, India in 2012. Currently he is PhD Scholar in the Department of Information Technology, Jadavpur University, India.

He was employed as Programmer Analyst from 2007 to 2009 in Cognizant Technology Solutions. He has published 3 papers in International Conferences, 13 papers in International Journals and also a book titled "Software Transactional Memory: An Alternative to Locks" by LAP LAMBERT ACADEMIC PUBLISHING, GERMANY in 2012 co-authored with Utpal Kumar Ray.

Parama Bhaumik received B.Sc Phy(Hons.), B.Tech and M.Tech in Computer Science & Engineering from Calcutta University, India in 1996,1999 and 2002 respectively. She has done her Ph.D in Engineering from Jadavpur University, India in 2009. Currently she is working as Associate Professor in the Department of Information Technology, Jadavpur University, India. She has more than 32 research publications in Journals of repute, Book chapters and International Conferences.

Utpal Kumar Ray received the degree of B.E. in Electronics and Telecommunication Engineering in 1984 from Jadavpur University, India and the degree of M.Tech in Elecrical Engineering from Indian Institute of Technology, Kanpur in 1986.

He was employed in different capacities in WIPRO INFOTECH LTD., Bangalore, India; WIPRO INFOTECH LTD., Bangalore, India, Client: TANDEM COMPUTERS, Austin, Texas, USA; HCL America, Sunnyvale, California, USA, Clent: HEWLETT PACKARD, Cupertino, California, USA; HCL Consulting, Gurgaon, India; HCL America, Sunnyvale, California, USA; RAVEL SOFTWARE INC., San Jose, California, USA; STRATUS COMPUTERS, San Jose, California, USA; AUSPEX SYSTEMS, Santa Clara, California, USA and Sun Micro System, Menlo Park, California, USA for varying periods of duration from 1986 to 2002. From 2003 he is working as Assistant Professor in the Department of Information Technology, Jadavpur University, India. He has published 23 papers in different conferences and journals. He has also published a book titled "Software Transactional Memory: An Alternative to Locks" by LAP LAMBERT ACADEMIC PUBLISHING, GERMANY in 2012 co-authored with Ryan Saptarshi Ray.