Pulsating STM – The in-memory Optimistic Concurrency Control Technique for Multi Core Systems

Sana Jafar, Ranjana Rajnish, Pankaj Kumar

Abstract: ***In the world of ever increasing parallelism, the problem of deadlock-free concurrency control is inevitable. As the number of processing cores is increasing, the number of processing threads is also increasing, and with this increase in the number of processing threads, there is a good chance of problems arising due to lack of proper concurrency control. The application areas under the domain of advanced graphics, cryptography, deep learning, embedded system programming, artificial intelligence and networking are prone to the problems of heavy uncontrolled concurrency of threads. This paper presents a novel Software Transactional Memory (STM) based optimistic concurrency control technique that is deadlock free for threads accessing the in-memory data structure for the purpose of reading as well as writing. The technique is lock free and is based upon timestamping. Threads involved in the proposed approach possess the transactional properties of atomicity, concurrency and isolation. Durability is not expected as the threads are working on an in-memory data source. The approach involves lazy conflict detection that ensures minimum aborts and restarts as well as maximum concurrency among transactions. Being lock free, the algorithm is better than the existing lock-based techniques. The technique is tested on Sniper-6.1 multi core simulator simulating 64 CPU cores and running 16, 32, 40 and 50 threads in our case. The results show significant improvement in throughput with the increasing number of threads over the existing lock-based techniques as well as other STM techniques based on optimistic concurrency control.***

Keywords **: *Concurrency Control, Optimistic Concurrency control, multi core, Software Transactional Memory, parallel programming, in-memory data structure, Sniper multi core simulator, Cycles Per Instructions.***

I. INTRODUCTION

The era of multi core processing is advancing at a lightning speed towards an era where the number of cores will be extraordinarily higher. That demands for a most efficient and highly optimized parallel algorithm. Parallel programming is generally not as simple as serial programming but it is much more efficient than the latter when the workload is too large. Concurrency in parallel programming is the main issue which is both desirous as well as which leads to concurrency control problems if left unhandled. If concurrency is left uncontrolled, it will lead to problems like, dirty read, race condition, unrepeatable reads, lost update problem, lack of inter thread communication and deadlocks to name a few. If concurrency is restricted by over control, then the parallel program will lose its essence. Therefore, designing an efficient concurrency control algorithm is essentially important which could secure maximum parallelism as well as maximally optimized to guard the algorithm from the above problems.

Parallel programming works both for in-memory data structures as well as for OLTP Transactions. This paper focuses on algorithms that work on in-memory data structures. One such data structure that is used in this work is a one dimensional array.

Several techniques are there in literature that control concurrency. Some of them are lock based techniques, TM approach, timestamp based techniques, optimistic concurrency control, multi version concurrency control. Lock based techniques have the biggest con of limiting the concurrency by allowing only one thread to enter the critical section at a time. Also, they are most probable of causing a deadlock. So they are the least popular techniques. Transactional Memory(TM) is an alternative approach for lock free concurrency control[1]. This approach works for shared in-memory objects. The threads of execution inhibit the ACI properties of transaction, viz. Atomicity, Consistency, Isolation and access the read/write set of the shared in memory data objects. Two transactions are said to conflict if they access the same object and one of the access is a write access. Transactional memory is said to provide higher level of programming abstraction. There are three approaches supporting Transactional Memory: (i)SoftwareTransactionalMemory(STM), (ii)approach providing hardware support to accelerate the STM, (iii) Hardware Transactional Memory (HTM)[2]. Rest of the three concurrency control techniques are based on Transactional Memory.

Timestamp based technique is better than the lock based techniques as there are no locks so no fear of deadlocks and works on TM. The concurrency is controlled via schemes that allot and compare the timestamps of the elements in the read and write set of the systems with the timestamp of the current transaction. Accordingly it is decided whether the transaction has to proceed reading or writing

Revised Manuscript Received on October 15, 2019
***** Correspondence Author
Sana Jafar*, Amity Institute of Information Technology, Amity University Uttar Pradesh, Lucknow Campus, Lucknow(India), India. Email: india.sana@gmail.com
Ranjana Rajnish, Amity Institute of Information Technology, Amity University Uttar Pradesh, Lucknow Campus, Lucknow(India), India. Email: rrajnish@lko.amity.edu
Pankaj Kumar, Department of Computer Science and Engineering, Sri Ram Swaroop College of Engineering and Management, Lucknow, India. Email: pk79jan@gmail.com

the elements in the read/ write set respectively or it has to abort and restart. Aborting and restarting is a costlier affair. Also having a centralized timestamp manager may become the reason of bottlenecks in a highly loaded system. Timestamping alone, is therefore not a very good solution always. Optimistic concurrency control is as the name suggests, optimistic in its approach towards accessing and writing the elements in a shared datastructure. This technique assumes that initially all the transactions are allowed to access and update the data elements in the shared datastructure and then the validity of the transaction is decided later before the final commit. Being optimistic, this technique ensures maximum threads or transactions participating in the system thus increasing the concurrency. However, in a highly contended workload there will still be a number of aborts and restarts. The challenge of a good concurrency control algorithm is to minimize these aborts and restarts and to ensure maximum concurrency among transactions or threads. Multi version concurrency control is an Optimistic concurrency control technique. In this, the in memory data objects are assumed to have multiple versions each. This is done to ensure maximum concurrency. The transactions have freedom to update as many versions as they want concurrently if they happen to pass the validity test.

II. RELATED WORK

Lot of work has been done on Transactional Memory and much is still going on. Some of the relevant study made is mentioned below:

In[3], Nir Shavit and Dan Touitou invented a Software Transactional Memory as a novel method towards translating sequential implementations of objects into highly concurrent non-blocking ones using k-word compare&swap STM-transaction. The work was based on multi-processors.

Mohammed El-Shambakey and Binoy Ravindran have studied the SoftwareTransactional Memory approach in real time embedded system in [1]. They have analytically established the upper bounds on the transactional retry and response time.

Bratin Saha et.al. have presented a novel high performance software transactional memory system for a multi-core runtime in [4]. This paper has done detailed study of the various STM tradeoffs like optimistic concurrency control versus pessimistic concurrency control; undo logging versus write buffering and object based versus cache line based conflict detection. Also the authors have developed the novel STM designs that works in cooperation with other components of McRT system to prevent blocking of active transactions through inactive transactions. The McRT STM is read versioning and undo-logging system and implements both object-based conflict detection and cache-line based conflict detection. The scheme is based on locking and versioning of the locks.

Yunlong Xu et.al. have developed a STM based technique for GPU based systems in [5]. The authors claim that their technique is free from livelocks and is scalable. The technique involves three characteristics. The first characteristic is Hierarchical validation that implements the conflict detection. Hierarchical validation is said to be the combination of value based validation and timestamp based validation. It involves the use of global version locks. The second characteristic is encounter time lock sorting. This feature is introduced in order to avoid livelocks. The third characteristic is coalesced read/write set organization in which the read/write set of all transactions within a warp are merged for reducing the overhead of transaction bookkeeping.

In [6], the authors have studied that prior ways to amortize the commit latencies in GPU SIMT applications, for example reducing the transactional warps to very few per SIMT core, aborting and restating the transactions; have resulted in poor performance and not actual reduction in commit latencies. The authors have thus, proposed a new GPU Hardware TM called GETM (GPU TM) based on eager conflict detection and lazy version management. The solution is based on timestamping and lock mechanism.

In [7], the authors have presented APUTM, a transactional memory approach for Accelerated Processing Units (APUs). Here they have deployed the concept of minimizing the access to the shared memory for reducing the conflicts among transactions. They have adopted a lazy conflict detection and lazy version management approach. One implementation is based on global sequence lock for reducing the commit latency of the transactions and the other implementation checks the transactional conflicts by using a private read set.

In [8], the authors have reviewed the currently existing concurrency control techniques for in-memory databases. Three such techniques have been discussed with their pros and cons. These are Cicada[9], MOCC[10], TicToc[11]. MOCC is based on optimistic concurrency control with a slight variation by using the concept of temperature to acquire selective read locks and minimize aborts. TicToc is a timestamp ordering scheme and is optimistic in nature. The commit timestamp for a transaction is calculated dynamically and is allotted just before the commit point. Cicada is optimistic, multi version and multi clock concurrency control scheme.

In [12], the authors of NEMO, a NUMA-aware TM algorithm have proposed a well optimized solution for providing scalability to applications running in NUMA architectures. NEMO is tested using well-known and synthetic OLTP transactional workloads. The authors have performed two tests whose results form the basis of the design of NEMO. In the first test various STM algorithms TL2[13], SwissTM[14], TinySTM[15], RingSTM[16], and NOrec[17], implementing a version of the Bank benchmark, partition the accounts across different NUMA zones and threads operate only on accounts stored in their local NUMA zones. The test shows that on incrementing the number of threads beyond 16, the algorithms cease to scale and the cost of updating the global metadata at this point also goes up significantly. In the second test, the authors have calculated the latency required for incrementing the logically shared timestamp through Compare-and-Swap. They have deployed two configurations: one in which there is a single timestamp in one NUMA zone incremented by multiple threads; the second configuration in which there are 8 timestamps located in 8 different NUMA zones and incremented by 8

threads in their local NUMA zones. The result shows that the former configuration provides almost no scalability due to heavy traffic in case of high number of threads. The latter configuration, on the other hand provides better scalability even when CAS primitive is used.

III. PULSATINGSTM

In an effort for developing a more efficient concurrency control scheme for in-memory data structures, the authors have developed PulsatingSTM. This STM approach is lock as well as deadlock free scheme. It is primarily inspired from TicToc[11], the timestamp ordering scheme for in-memory databases. But it is better than TicToc as there are no locks in it. Also TicToc is for in-memory databases, whereas PulsatingSTM is for in-memory data structures. PulsatingSTM is primarily based upon the optimistic concurrency control scheme and is free from the overheads of centralized timestamp manager. Here the commit timestamp of a transaction is computed not early than the commit phase. The authors have adopted lazy conflict detection[6]. The three phases in this scheme are: (i) Read phase, (ii) Validation phase and (iii) Write phase.

A. Read Phase

In this phase the threads as transactions are allowed to read the elements from the shared memory into their private read/ write set based upon the purpose of access. If the access is read access, the thread reads that element in its private read set. It notes down the read and write timestamp of that element in the set, displays the element and sets the pointer to the element in the shared array. If the access is the write access, the thread reads the element in its private write set, notes down read and write timestamps of the element, updates the element in the write set, and sets the pointer to the element in the shared array.

B. Validation Phase

In this phase the commit timestamp of the transaction is calculated based upon the read and write timestamps of the elements in its read/write set. It has following three major steps:

a) Firstly, the transaction's current timestamp is checked against the read and write timestamps of the element in the read set of that transaction. The transaction's timestamp must be greater than write timestamp and less than read timestamp. If not then it is adjusted to some value abiding this constraint.

b) Secondly, the validation is done against the read set. If the transaction's timestamp is in between write timestamp and read timestamp for every element in the read set, then it is assumed that the transaction has read a valid version, else it is assumed that the version read by the transaction is invalid. In that case the changes made by the transaction in its write set are rolled back.

c) Thirdly, if the transaction has read a valid version, then its final commit timestamp is calculated which should be greater than the read timestamp of all the elements in the write set.

C. Write Phase

After successful validations, each transaction does the following:

a) sets the read and write timestamps of its write set to its own commit timestamp.

b) Copies the write set to the shared memory.

IV. DESIGN OF PULSATINGSTM

PulsatingSTM is a timestamp based optimistic concurrency control system. The main features of this design are:

1. Avoids deadlock as there are no locks.
2. Being optimistic, it allows all the threads to access and update the copy of the element in the datastructure without worrying about any conflicts, thus increases the concurrency among threads.
3. Each element maintains a metadata. This metadata stores the read and write timestamps of the element, data held in the element and a pointer in the original data structure. A thread (transaction) accessing this element will use its metadata.
4. Every transaction has certain timestamp associated with itself which the unique value allotted to it when it enters the system.
5. Every transaction has a read and write set associated with itself. The read set contains all the copies of elements that the transaction has read and write set contains all the copies of elements that it has updated along with their updated values.
6. The commit timestamp of the transaction is computed late in the execution just before commit from the read and write timestamps of elements in its read/ write set.

Due to the property of Isolation of Transaction, they do not interfere with each other's read/write sets. Since the write operation is atomic in nature, the transaction will roll back on reading an invalid version or incorrect data. Consistency is maintained as the transactions are serializable.

Each node of the in-memory datastructure has some metadata associated with it which gives information about the read/write timestamp, a pointer to the node in original data structure and the data value of the node, as mentioned above. The read/ write timestamp associated with the elements of the datastructure is the timestamp value of the last committed transaction that read or wrote that element. These metadata are tabulated in table1.

Table- I: Metadata of a node and of a transaction in PulsatingSTM

rtime	Read timestamp
wtime	Write timestamp
point	Pointer to the node in the original data structure
data	Data value of the node
tranread [ ]	An array maintaining read set of the transaction
tranwrite [ ]	An array maintaining write set of the transaction
timestamp	Timestamp associated with the transaction when it enters the system

Algorithm 1 shows the BeginTX, ReadTX, ValidateTX, and WriteTX procedures of pulsatingSTM.

BeginTX begins by allotting

each thread or transaction a unique value that is calculated by the autoinc procedure (Algorithm 4).

Algorithm 1 PulsatingSTM algorithm

```
1: procedure BeginTX
2:   timestamp⟵autoinc()
3: end procedure
4 procedure ReadTX
5:    p⟵ write(tranwrite, (arr+i),i,p,timestamp)
6:    s⟵read(tranread,(arr+i),i,s,timestamp)
7: end procedure
8: procedure ValidateTX
9:   k⟵0
10:  if tranread[k].rtime!=tranread[k].wtime do
11:   while k<s do
12:     while timestamp<=tranread[k].wtimeOR
       timestamp>=tranread[k].rtime do
13:       if timestamp<=tranread[k].wtimedo
       timestamp++;
14:     elseif  timestamp>=tranread[k].rtime do
       timestamp--;
15:      end if
16:    end while
17:    k⟵k+1
18:   end while
19:   end if
20:  k⟵0
21: while k<s do
22:  if  timestamp >tranread[k].wtime  AND
       timestamp<tranread[k].rtime  do
23:   flag⟵1;
24:   else
25:    flag⟵0; break;
26:   end if
27:  k⟵ k+1
28: end while
29: if flag == 0 do
30:   Transaction has read invalid version and has to
      roll back
31: for  j⟵ 0, j< p do
32   if tranread[k].point==tranwrite[j].point do
33:    for l⟵j, l<p-1 do
34:     tranwrite[j]⟵ tranwrite[j++];
35:     l⟵ l + 1
36:    end for
37:  p⟵p -1; break
38:  end if
39: end for
40: else do
41: Transaction has read a valid version
42:  k⟵0
43: while k < p do
44: if  timestamp<tranwrite[k].rtime do
45: timestamp⟵tranwrite[k].rtime
46: end if
47: k⟵ k+1
48: end while
49:  if  k == p do
50:    timestamp⟵ timestamp +1
51:  end if
52: end if
53:  commit timestamp is timestamp
54: end procedure
55: procedure WriteTX
56:  for  j=0 , j<p do
57:   tranwrite[j].wtime⟵timestamp
58:   tranwrite[j].rtime⟵ timestamp
59:  *(tranwrite[j].point)⟵tranwrite[j]
60:   j⟵j+1
61:  end for
62:  end procedure
```

Here, **arr** is a globally shared array which the transactions access for the purpose of reading and writing. It is an array of structure wherein each element holds the read and write timestamp of the integer element, its data value and a pointer pointing to itself. **tranwrite** and **tranread** are the write and read sets respectively of every transaction which are implemented as dynamic arrays. Variable **i** is holding the index for **arr**. Variables **p** and **s** are the size of the write and read set respectively for each transaction. Variable **timestamp** is the unique timestamp allotted by procedure BeginTX (line numbers 1 to 3) to each transaction entering into the system.

The Read Phase begins by every transaction calling the functions **write( )** and **read ( )** in parallel (line numbers 4 to 7).

Algorithms 2, 3 and 4 show the functions associated with read operation, write operation and auto increment respectively as used in Algorithm 1. The **read( )** and **write( )** functions are as explained under the sub-section A of section III above. Here the variables **TR, TW v, i, size and timestamp** are the formal arguments of the functions read( ) and write( ). **TR** and **TW** are pointing to the **tranread** and **tranwrite** sets as used in Algorithm 1.

The validation phase in Algorithm 1 is shown under procedure ValidateTX that starts from line number 8 and extends till 54. The procedure begins by ensuring that the read and write timestamps of each and every element in the transaction's read set is different. Once this is ensured, the transaction's timestamp is checked whether it is lying between the read and write time stamps of the elements in the read set. If not then its adjusted to abide this constraint (line numbers 11 to 18). After this, the validation is done against the read set as explained in the sub-section B of section III above (line numbers 20 to 41). Here, the variable **point** is a pointer to the element in the shared data structure. The third part of validation phase where the transaction has read a valid version and its final commit time stamp is calculated is from line number 42 to 51.

The write phase in Algorithm 1 is shown through procedure WriteTX that starts from line number 55 and extends till 62. Line numbers 57 to 58, assign the valid transaction's timestamp to the read and write timestamp of every element in its write set. Finally line number 59, copies this element to the shared memory as explained under the write phase of sub-section C of section III.

Algorithm 2 Read operation algorithm

```
1: procedure read (struct element *TR, struct element
*v,int i,int size, int timestamp)
2: TR[i]←*v;
3: size ← size+1;
4: TR[i].rtime ← v->rtime;
5: TR[i].wtime ← v->wtime;
6: TR[i].point ← v;
7: return size;
8: end procedure
```

Algorithm 3 Write operation algorithm

```
1: procedure write (struct element *TW, struct
element *v,int i,int size, int timestamp)
2: size ← size+1;
3: TW[i] ← *v;
4: update (TW[i].data);
5: TW[i].rtime ← v->rtime;
6: TW[i].wtime ← v->wtime;
7: TW[i].point ←v;
8: return size;
9: end procedure
```

Algorithm 4 autoinc algorithm

```
1: procedure autoinc
2: static int c ←1;
3: c ←c +1;
4: return c;
5: end procedure
```

V. EXPERIMENTAL TEST BED

The above experiments are performed on multi core simulator the Sniper-6.1[18] simulating 64 CPU cores using the gainestown configuration file for Xeon X5550. The configuration has following settings:

- Core frequency—2.66 GHz
- Number of cores sharing L3 cache— 4
- Data access time by L3 cache – 30 cycles
- Network memory model --- bus
- Bus bandwidth – 25.6 GB/s (12.8 GB/s per direction and per connected chip pair)
- Local traffic has been ignored because the memory controllers are on chip.

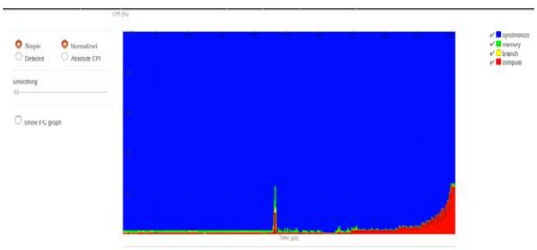

Fig. 1. Average CPI graph for the algorithm running 16 threads on 64 simulated cores

In Fig. 1-4, the average Cycles per Instructions (CPI) graph is plotted for time in microseconds on the X-axis versus percentage CPI on Y-axis.

Following observations can be deduced from the Fig. 1. The total time consumed by the algorithm is 509 microseconds. Most of the CPI percentage per time is synchronization. Only towards the last fourth part of execution time, the CPI is spent on computation. A spike in between the graph shows the read phase. Towards the end is the validation and write phase that covers around 25% of the CPI. The spike also covers around 25% of the CPI.

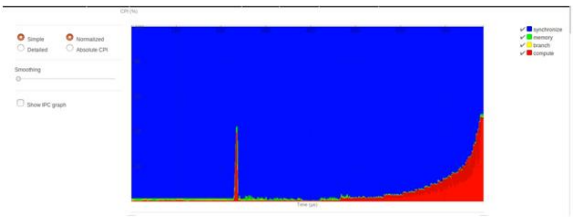

Fig. 2. Average CPI graph for the algorithm running 32 threads on 64 simulated cores

In Fig. 2, the algorithm is run using 32 threads. The total time consumed by the algorithm is 786.7 microseconds. The spike in the graph is a little more than that seen in the graph for 16 threads. It is now covering around 45% of CPI. Also, as the number of threads has doubled, more computation is done and that is shown towards the end of the graph, with the validation and write phase that covers around 50% of CPI now.

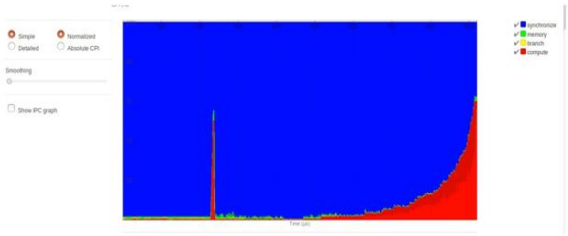

Fig. 3. Average CPI graph for the algorithm running 40 threads on 64 simulated cores

Fig. 3 shows the result of running the algorithm using 40 threads on 64 cores. With 40 threads, the read phase spikes up to around 55% and the validation and write phase covers around 52% of CPI. This is due to more number of threads and hence more works done in validation and write phase. The total time taken by the algorithm with 40 threads is 922.8 microseconds.

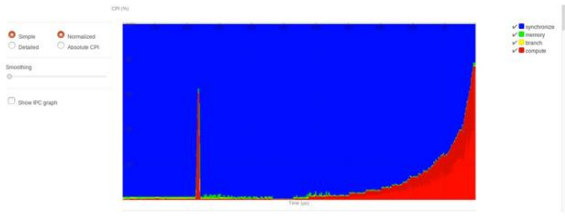

Fig. 4. Average CPI graph for the algorithm running 50 threads on 64 simulated cores

Fig. 4 shows the result of running the algorithm using 50 threads on 64 cores. With 50 threads, the read phase spikes up to around 62% and the validation

and write phase covers around 78% of CPI. The total time taken by this algorithm with 50 threads is 1.095 milliseconds.

VI. EVALUATION

We have closely observed the results obtained on Sniper by comparing the values obtained by running PulsatingSTM with 16, 32, 40 and 50 threads on 64 cores employing the gainestown configuration. The results suggest that on increasing the number of threads, the throughput is increasing. This is attributable to the fact that as the thread count increases, the branch misses, L1, L2 cache misses and DRAM access reduces, thus, giving a better performance. The results obtained on running the PulsatingSTM on sniper are tabulated in Table 2.

Table-II: Parametric values from sniper for running PulsatingSTM employing different number of threads on 64 cores

	Threads			
	16	**32**	**40**	**50**
Instructions	5.580 m	19.35 m	29.25 m	44.25 m
IPC	0.064	0.145	0.186	0.237
Cycles	1.352 m	2.093 m	2.455 m	2.913 m
Time	508.3 µs	786.7 µs	922.8 µs	1.095 ms
Branch MPKI	1.997	0.855	0.655	0.508
L1-I MPKI	1.090	0.577	0.469	0.302
L1- D MPKI	1.291	0.596	0.470	0.371
L2 MPKI	2.202	1.115	0.899	0.725
DRAM APKI	0.912	0.402	0.312	0.249

IPC: Instructions Per Cycle, MPKI: Misses Per Kilo Instructions, L1-I: Instruction level L1 Cache, L1-D: Data level L1 Cache, L2: L2 cache, DRAM: Dynamic Random Access Memory, APKI: Access Per Kilo Instructions

Fig. 5 shows the increase in throughput by running PulsatingSTM on higher number of threads.

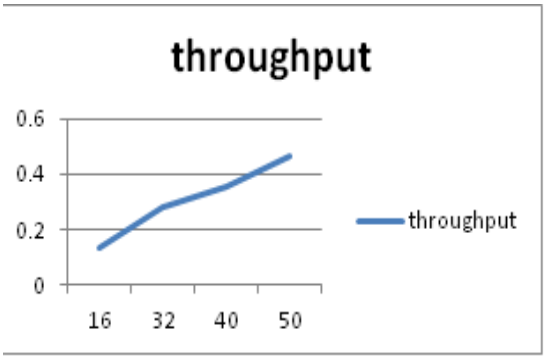

Fig. 5. Throughput Versus Number of Threads

VII. CONCLUSION

PulsatingSTM is a novel in-memory optimistic concurrency control technique based on timestamping. It employs lazy conflict detection among the threads. The algorithm is better than lock based protocols and other STM protocols as it is free from locks. The algorithm is built on multiple threads doing the same job of reading and writing. It is optimistic as all the threads are allowed to read a copy of data from the shared memory in their private read/write sets and perform write operation in their private set. Only when the writing is complete, the validation phase begins. In the validation phase, transaction's timestamp is validated and its commit timestamp is calculated just before the write phase.

The algorithm is run on 64 cores using 16, 32, 40 and 50 threads on sniper, the multi-core simulator. The results obtained show that the throughput obtained on running this algorithm increases with increase in the number of threads.

VIII. FUTURE WORK

The authors next will implement the algorithm with different number threads doing different jobs of reading and writing. The authors have proposed a multi-version flavor of this algorithm in the upcoming work wherein each element of the shared data structure will have multiple versions and threads writing to it will be writing new versions on the data structure. Also, the authors propose employing this algorithm to techniques like parallel sorting of enormous arrays and come up with the results.

REFERENCES

1. El-Shambakey, Mohammed and Binoy Ravindran, "STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs." In Proceedings of SAC (2012), Riva del Garda, Italy, March 25-29, 2012, pp. 1602-1609.
2. Yan Solihin, Fundamentals of Parallel Multi core systems, Broken Sound Parkway NW: CRC Press, Taylor and Francis Group, 2016.
3. Nir Shavit and Dan Touitou, "Software Transactional memory." In Proceedings of the 14th Annual ACM Symposium of PODC 95, Ottawa Ontario CA, August 20-23, 1995, pp. 204-213.
4. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, Benjamin Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime.", In Proceedings of 11th ACM SIGPLAN symposium on PPoPP, New York, NY, USA., '06 March 29-31, 2006, pp. 187-197.
5. Yunlong Xu, Rui Wangy, Nilanjan Goswamiz, Tao Liz, Lan Gaoy, Depei Qian, "Software Transactional Memory for GPU Architectures", In Proceedings of IEEE/ACM International Symposium on CGO '14, Orlando, FL, USA, February 15 - 19 2014, pp. 1
6. Xiaowei Ren and Mieszko Lis, "High-performance GPU Transactional Memory via Eager Conflict Detection", In Proceedings of 2018 International Symposium on High Performance Computer Architecture, Vienna, Austria, Feb 24-28, 2018, pp. 235-246
7. Alejandro Villegas , Angeles Navarro, Rafael Asenjo, Oscar Plata, "Toward a software transactional memory for heterogeneous CPU–GPU processors" The Journal of Supercomputing, https://doi.org/10.1007/s11227-018-2347-0, pp. 1-16
8. Sana Jafar, Pankaj Kumar, Ranjana Rajnish, "Reviewing the Current Concurrency Control Techniques in Multi and Many core systems", In Proceedings of the 12th INDIACom; INDIACom-2018 5th 2018 International Conference on "Computing for Sustainable Global Development", Bharati Vidyapeeth's Institute of Computer Applications and Management (BVICAM), New Delhi (INDIA), March 14th – 16th, 2018, pp. 525-530.
9. H. Lim, M. Kaminsky, and D.G. Andersen, "Cicada: Dependably Fast Multi-core In-Memory Transactions", In Proceedings of

the 2017 ACM International Conference on Management of Data SIGMOD, Chicago, Illinois, USA, May 14 - 19, 2017, pp. 21 – 35
10. T. Wang, and H. Kimura, "Mostly-Optimistic Concurrency Control for Highly contended dynamic workloads on a thousand cores", In proceedings of VLDB Endowment, vol. 10. No. 2., 2016, pp. 49-60.
11. X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time travelling Optimistic Concurrency Control", In Proceedings of the 2016 International Conference on Management of Data SIGMOD, San Francisco, California, USA, June 26 - July 01, 2016, pp. 1629-1642.
12. Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, Roberto Palmieri " Nemo: NUMA-aware Concurrency Control for Scalable Transactional Memory", In Proceedings of 47th International Conference on Parallel Processing, Eugene, OR, USA, August 13–16, 2018, Article No. 38.
13. Dave Dice, Ori Shalev, and Nir Shavit, "Transactional Locking II.", In Proceedings of the 20th international conference on Distributed Computing, Stockholm, Sweden, September 18 - 20, 2006 , pp. 194–208.
14. Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka, " Stretching Transactional Memory", In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, June 15 - 21, 2009, pp. 155-165.
15. Pascal Felber, Christof Fetzer, and Torvald Riegel, "Dynamic Performance Tuning of Word-based Software Transactional Memory", In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA, February 20 - 23, 2008, pp. 237–246.
16. Michael F. Spear, Maged M. Michael, and Christoph von Praun, "RingSTM: Scalable Transactions with a Single Atomic Instruction", In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, Munich, Germany, June 14 - 16, 2008, pp. 275–284.
17. Luke Dalessandro, Michael F. Spear, and Michael L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records". In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Bangalore, India, January 09 - 14, 2010, pp. 67–78.
18. [18] T. E. Carlson, W. Heirman, and L. Eeckhout., "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations", In Proceedings of International Conference on High Performance Analysis, Networking, Storage and Analysis, Seatle, WA, USA, Nov. 12-18, 2011, pp. 1-12.

AUTHORS PROFILE

Sana Jafar is currently working as an IT consultant with Argus Technology LLC. She is a research scholar in the faculty of Information Technology from Amity University Uttar Pradesh Lucknow Campus, enrolled since January 2015. She has worked as an Assistant Professor (Computer Science & IT) in the Department of Amity School of Engineering and Technology, Amity University Uttar Pradesh Lucknow Campus from 2009 till 2018. She completed her MCA with silver medal and received her degree with honors in 2009. Her area of research is Parallel Computing and High Performance Computing. She is a student member of IEEE. She has 4 papers published and presented in IEEE sponsored International and National conferences and one book chapter published in Scopus Indexed Ebook series titled "Advances in Parallel Computing", IOS Press, Netherlands. Sana Jafar has Participated in the Short Term Course (under QIP IIT Delhi) on many core parallel Programming at IIT Delhi from 4th June -15th June 2018., learning hands on Nvidia CUDA: API for parallel programming in GPU based architecture and accessed the HPC clusters at IIT Delhi (PADUM). She has also worked as an intern under Prof Subodh Kumar (Dept. CSE at IIT Delhi) under the Summer Faculty Research Fellow Program from 4th June -13th July 2018 at IIT Delhi. She has published a useful workbook on Object oriented programming using C++ as main author(publishers: Alok Prakashan) for the B.Tech students of Amity University and is in the process of generalizing it for the B.Tech pursuing students of all the engineering colleges in Uttar Pradesh. She has successfully attended various faculty development programs and workshops in Amity University Lucknow campus and outside. As well has played an important part in conducting such programs within the Amity University Lucknow campus. She has attended the five days military training camp organized by Amity University Manesar in 2016 as faculty guide with post graduate students. She has also secured a second position in women badminton in the annual sports meet of Amity University Lucknow (Sangathan) in 2015.
Sana Jafar is diligently working towards inventing innovative and efficient ways for improving concurrency control methods in multi and many core systems using STM and optimistic methods.

Dr. Ranjana Rajnish is an Assistant Professor at Amity Institute of Information Technology at Amity University, Lucknow. Dr. Ranjana possesses approximately 25 years of experience in academics/research. She has been engaged with institutions like U.P. Technical University and Amity University in roles ranging from a faculty in computer science to Academic Head. Her area of interest includes Software Engineering, Opinion Mining/Sentiment Analysis and Healthcare.
She has several publications in national and international journals and conference proceedings of National and International Conferences of repute. She is also member of various professional bodies like Computer Society of India (CSI), Association of Computing Machinery(ACM), International Association of Engineers (IAENG), Internet Society and Computer Science Teaching Association (CSTA).
Along with being a committed teacher and a passionate researcher, Dr. Ranjana is reviewer for various International Journal and member of editorial board for different International Journals. She is also reviewer, member of technical programme committee in various conferences of repute in and outside India. She has many Ph.D. scholars pursuing Ph.D. under her.

Dr. Pankaj Kumar is currently working as Assistant Professor (Reader) in Department of Computer Science & Engineering in Sri Ramswaroop Group of Professional College, Lucknow. He has more than 18 years of teaching experiences. He received his MCA degree in 2001, M.Tech in 2010 and PhD degree in Computer Application in 2011. His Area of Expertise is Parallel Computing/ Mining/Security. More than 50 research papers of Dr. Pankaj Kumar have been published in various national/international journals and IEEE proceeding publication. He is Senior Member of IEEE, Professional Member of ACM and Life member of CSI, IETE, ISTE, IAENG, ISOC and IACSIT. He is member of Management Committee of CSI and IETE Lucknow Chapter. He is reviewer for various International Journal and member of editorial board for different International Journals. He also participated in various conferences as reviewer, member technical committee, and co-chair. One PhD thesis is awarded and eight students are enrolled as PhD scholar under his guidance. More than 10 students are guided by him in M.Tech Thesis.