2015

Speculative Parallelism and Transactional Memory Algorithms in TBB and LIBITM

Stephen Robert Louie
Lehigh University

Speculative Parallelism and Transactional Memory Algorithms in LIBITM and TBB

By

Stephen Louie

A Thesis

Presented to the Graduate and Research Committee

Of Lehigh University

In Candidacy for the Degree of

Master of Engineering

In

Computer Science

Lehigh University

May 2015

Table of Contents

Abstract

This paper implemented four transactional memory algorithms one hardware transaction algorithm and three software transaction algorithms. The goal of this research was to investigate the cost of determinism in a parallel world. The current standard for top performing parallelism is working on jobs that are independent such as a Map Reduce, because it isolates each job. This paper attempts to investigate the cost of obtaining deterministic output on independent and non-independent tasks while being parallel. It is obvious that the cost of determinism will be steep, and the results presented will not exceed the independent parallel cases. But how damaging is determinism, can it exceed serial executions, if so when is it appropriate to run a deterministic parallel execution over a serial execution. The findings concluded that ordered parallelism performed much worse than serial executions. To perform parallel operations in a deterministic way efficiently would require a high level of knowledge about the specific hardware and benchmark.

Introduction

For over 20 years, Transactional Memory (TM) [3] has been viewed as the most promising proposal for simplifying the creation of correct, scalable concurrent programs. The concept behind TM is tantalizingly simple: programmers merely annotate regions of code that must appear to execute atomically, and then a run-time system, augmented with custom hardware, executes those regions concurrently (as "transactions"). During execution, the run-time system tracks memory accesses, detects conflicts, and aborts and retries transactions as necessary to ensure that the program behavior is equivalent to one in which the execution of transactions does not overlap.

The recent addition of TM support to IBM [6,11] and Intel [5] processors brings the field of concurrent programming much closer to a state in which programmers can eschew locks in favor of transactions. However, first-generation hardware TM systems carry a number of limitations. Most significantly, these implementations are "best effort" [7], in that they do not guarantee that any transaction attempt will commit. In particular, a transaction attempt may fail if it accesses more unique locations than the hardware can support, or if there is an interrupt (e.g., a timer interrupt) during its execution. Consequently, a TM runtime that wishes to use hardware TM must provide a software fall-back path. This fall-back path also provides a means of circumventing the hard-coded conflict resolution strategy

("requester wins" [1]) that the hardware enforces, so as to allow the run-time system to improve the chance that a long-running transaction does not starve.

This research looks to expand on TM current TM algorithms found in LIBITM, such as the multi-lock, and hardware transactional work. There are two more algorithms added, one being a lazy implementation and the final being a hybrid algorithm loosely based on the algorithm presented by Oancea [8]. The work here is utilizing Intel's Threading Building Blocks [4] library to combine the worlds of speculative parallelism and transactional memory (GCC's LIBITM). It utilizes a simple framework to enable faster TM algorithm development. These results presented here attempt to quantify the cost of determinism in a concurrent world. Given a job is mostly independent and deterministic output is desired, how much does that cost in terms of performance as compared to a serial execution and to a non-deterministic parallel execution. Clearly, this should not exceed a non-deterministic parallel execution since this deterministic parallel execution is much more restrictive on parallelism, but the question comes down to, when is this better than a serial execution.

All development and testing was completed on an 64bit Ubuntu box with 8 Intel(R) Core i7-4770 chips that clock 3.40Ghz. Each core has an L1 cache of 32K, L2 cache of 256K and an L3 cache that is 8MB and runs at 800MHz. It utilized a core TBB [4] version 4.3, with slight modifications to

support a deterministic parallel_for loop, called a parallel_for_ordered loop here. An experimental version of GCC 4.9.0 was used for compilation and development, the core of LIBITM was also taken from this version of GCC.

This paper is broken into a few sections the first discussing the implementation of the hardware transactional algorithm, then the lazy algorithm, multi-lock algorithm, oanceaLite algorithm, evaluation, and finally a conclusion that wraps up the paper and looks into future work and improvements for this area of research.

HTM Implementation

This hardware transaction implementation utilizes Intel's Transactional Synchronization in Haswell, also called TSX. Essentially Intel's hardware transaction API. The HTM algorithm presented here utilizes a few global variables that can be found in Appendix A: Listing 1. A code snippet can be found in Appendix A: Algorithm 1 has all transactions attempt to complete their own transaction and loop until it is complete. Each transaction will call xbegin() and examine its output for a successful return code, the handling of all return codes is described below.

XBegin Successful

Given a successful begin of a transaction there are three options to handle. The first is when we have the currentOrderNumber != tx.range.begin,

this transaction will backoff and wait for its turn. If the currentOrderNumber == tx.range.begin and the spin lock was not held it will commit. The spin lock is a global lock over execution of xbegin(). Given the spin lock was held at the time it is forced to abort, which is detrimental since this is the next transaction that needs to commit. The abortFlag will be set, which stops all transactions from operating and backs off. This will ensure all other transactions stop, and the next desired transaction will execute and complete, hence making progress.

XBegin Unsuccessful or Aborted

The first potential option with this set is an ABORT.EXPLICIT, in this case an abort was instantiated for some reason above. The first case is when it isn't our turn and the abortFlag has not been set we will backoff and try again. If the abortFlag is set, we will spin until the abortFlag has been released. The second case when this transaction equals the currnetOrderNumber and the spinlock is held, this will wait for the lock to release, once released it tries to commit again.

Lastly, there are some special cases where Intel provides meta-data when a transaction aborts. One such case is ABORT.CONFLICT, this is when an abort is caused by a memory conflict. If a transaction aborts for this reason it will backoff and try again. Another special case is ABORT.CAPACITY, this indicates that the memory is simply not large enough to support the

transaction pool. In this scenario the transaction will backoff and set the abortFlag to stop all transactions from operating. In every other case the transaction will simply backoff and try again.

Lazy Implementation

The lazy transactional memory algorithm [9] is implemented here to investigate how different transactional memory algorithms will affect deterministic outcomes. As is with lazy concurrency, no blocking locks are taken during processing. At commit time the transaction will ensure its correctness and obtain locks to proceed. The benefit of this method is to minimize the amount of overhead. As long as there is a low level of overlap this method should be one of the higher performing algorithms presented in this paper. It minimizes the memory load and book keeping on the software side. The global and local variables can be found in Appendix A: Listing 2 The following sections will discuss the implementation details, first starting off with a crucial component of the lazy method, the redo log. This redo log will track all the operations made by the transaction and only commit them at commit time. Its implementation is key to the success of a lazy algorithm because it must be searched constantly to see the changes and make modifications. This is followed by three critical functions, pre-load, pre-write and trycommit.

Redo Log – Binary Search Tree

The redo log here is a Binary Search Tree that contains the addresses and values that should be written to those locations given a it can commit. This tree is used by each transaction to track the changes it makes. A binary search tree was used to minimize the search time in looking for and making updates. This binary search tree is made up of nodes that are 64 byte slabs of data. Each node has a mask that shows which bytes are written in that slab. This BST utilizes integer indexes to specify nodes and slabs. This enables one to realloc nodes / slabs and still use the same index value to identify the node or slab by indexing the proper number in the nodepool or slab pool.

Looking at this BST one could look into using a red-black tree or some balanced tree to ensure the depth of the tree remains at a consistent level in relation to the nodes present in the tree. For the needs of this research, this BST was found to be sufficient for storing, and recalling the modifications made by each transactions

Pre-Load

This function prepares a transaction for reads by scanning the orecs for conflict. This method conducts a relaxed read of memory, and cycles through the orecs and records any effects. Given the orec is locked and not too new it is considered a successful read and add it to the readlog. Given the orec is very new, we will extend our time and deem it a successful read and add it to the

readlog. Given the orec is locked, we will abort and try again. This function completes and returns the starting location of the readlog.

Pre-Write

This function starts by obtaining a start time, which will be used to indicate this transactions time. The transaction will look to lock all its orecs if it hasn't already locked them. Given another transaction holds the orec this transaction will abort and try again. A memory fence is required to ensure orecs are obtained in the proper order. Lastly, these writes are added to the writelog given no aborts were taken in the prior loop.

TryCommit

The code is shown in Appendix A: Algorithm 2, walking through this psuedo-code one can observe that this starts by looking at the redo-log. Given it is empty, this transaction is clear to commit. If it is not, it will cycle through the addresses found in the redo-log and it obtains a snapshot time to indicate its presence. It also will validate all its read operations and ensure they have not changed, given a change is found it will abort. Once the validation is deemed successful this will write out all the items in the redo-log to memory. Lastly, it will clear all the logs and update the currentOrderNumber to the tx.range_end value.

ML Implementation

The mulit-lock ordered transactional implementation [2] is very similar to the standard LIBITM multi-lock implementation. It differs with the addition of a range object that contains the range_begin, range_end, range_grainsize, a global currentOrderNumber and an abortFlag. The general strategy for this implementation is to maintain as much of the multilock implementation as was there. The modifications come during the major functions such as, pre_write, pre_load, post_write, and post_load.

In each of these functions there is an initial check that looks if the abortFlag has been set and if the current transaction is not equal to the currentOrderNumber. If both are true it will abort. In each of these functions, if any sort of conflict is found we will abort as the standard multi-lock will but do an additional check to see if it should set the abortFlag. If the aborting transaction is equal to the currentOrderNumber, it will set the abortFlag. The abortFlag will make all other transactions stop working and abort at their earliest convenience. It ensures progress will be made and that the currentOrderNumber will advance. In this case a Compare And Set operation is not needed because at any time there will be only one transaction that can set the abortFlag, this transaction will be the next transaction to commit. (tx.range_begin == currentOrderNumber)

Trycommit handles a few cases, the first being if a transaction reaches here and is not the next desired transaction. Given this a transaction it will simply abort and start over. A transaction that is equal to the currentOrderNumber will commit. If this transaction set the abortFlag it will also reset the abortFlag before committing. By turning off the abortFlag this releases all other transaction and enables the pool of transactions to continue. The transaction will then run its validation clear its logs, move up the currentOrderNumber and commit.

OanceaLite Implementation

Instantiation of Oancea

In starting this method there are numerous globals that need to be instantiated. The first setting up the global currentOrderNumber, so the first transaction that is allowed to commit is the transaction that has a tx.range_begin value that is equal to the first loop iteration. The master_index tracking array will need to be instantiated to the length equal to the maximum active transaction count.

Transactions Entering the Pool

When transactions are initialized one must set up its local variables, this includes the three range items, range_begin, range_end, and range_grainsize. The is_active boolean and is_abort flags are set to false. During this initiation the transaction then looks to set its localAbortCount to

the globalAbortCount and acquire a master_index. A master_index is an index that it can write its logs to in each orec, that it owns as long as it is an active transaction. The Hybrid_Word was used here to enable all transactions to detect potential conflicts without CAS'ing a single variable in the orec, instead each transaction owns one index in this array that is the length of the maximum number of active transactions. In order to avoid abort order discrepancies, new transactions cannot enter the transaction pool during an abort sequence. The entering transaction acquires the abortFlag, and sets the localAbortCount to the globalAbortCount then unlocks the abortFlag without stopping execution of the active transactions. (This locking of the abortFlag does not stop all transactions, it simply prevents an abort sequence from starting) Hence the use of the isActive flag, see the rollBack section for more details on how each abort sequence is handled.

Hybrid_Word: Orec Structure

In order to make this method work a custom word had to be made to log in each orec, here we “exploded” the orec log to cover each active transaction. One could see the details of the hybrid_word in the Appendix A: Listing 4. The word has a lock on it to ensure only one transaction makes edits to the word at a time. Besides that there is a read_log and a write_log where each are the length of the maximum number of active transactions. At this time TBB will maximize the number of transactions at double the processor count.

So an 8 core hyper-threaded machine will have a read_log and write_log length of 16. Each location records the last transaction that either read or wrote to that location. When transactions are initialized they are given a master_index that is the index in the read_log and write_log. Since it is guaranteed that at any time one transaction can own an index in the read_log and write_log there is no need to CAS updating the logs and each transaction can just write their own range_begin value in their index. Avoiding a CAS is a large time savings for this algorithm. The following sections describe the code, a snippet of pseudo-code can be found in Appendix A: Algorithm 3.

Pre-Write and Pre-Load

Pre-Write and Pre-Load are very similar functions in their structure. They each start by checking if there is an abortSequence, if it is set it will unlock any orecs it is holding and wait in rollback for its turn to rollback. Given no abort sequence is active, the transactions will cycle through their orecs, acquire them, check for conflicts, and continue if no conflicts are found.

Pre-write will check for a WAR (Write after Read) conflict and WAW (Write after Write) conflict. In both scenarios an abort sequence will be started upon discovery of any of these conflicts. In each of these cases the abort sequence will begin with the current transaction causing all transactions with a range_begin greater than this transaction to stop and follow the abort sequence protocol (see details in Roll-Back section)

Pre-load checks for only a RAW (Read after Write) conflict. Pre-load will start an abort sequence in the RAW case because it is reading an orec after a write ahead has occurred. Checking for RAR (Read after Read) is not necessary, since it does not make anything invalid. If one had a constant in an orec and all transactions read from it, and no transaction wrote to this location, one would not want that to cause aborts due to read aheads. So in this case pre-load will only look for RAW conflicts.

Once an orec is confirmed to not conflict the transaction must log that it has been to this orec. By first logging in the transaction local logs. Then it will write to the orec / hybrid word log to its master_index location it claimed when it entered the transaction pool. This ensures that this transaction can properly rollback and it can give other transactions the ability to detect conflict. Without the master_index one could not see under reads, meaning only the front most read is visible, and it would be unknown who needs to rollback given an abort sequence.

TryCommit

Transactions that enter here have passed through both pre-write and pre-load without detecting any conflicts or being interrupted by any abort sequence. This leaves three cases in which transactions get to this point. 1) A Transaction started the abortSequence and passed to this place without conflicts, 2) A transaction that is equal to the currentOrderNumber arrived

without conflict. 3) A transaction that is not equal to the currentOrderNumber arrived without conflict.

Case 1 (started an abortSequence): This transaction must ensure that all active transactions above it have aborted. Once it confirms this it will end the abort sequence as described in the Roll-Back section.

Case 2 (==currentOrderNumber): This transaction should be able to pass straight through trycommit since it is the next transaction to commit, no abortSequence could be started that is smaller than it. It will pass and simply commit.

Case 3 (!= currentOrderNumber): This transaction will get here and spin and randomly backoff, checking two conditions. The first being if it needs to abort due to an abort sequence being started. If another transaction that is earlier than it causes an abort sequence it must go to roll-back. If during its waiting period no abort sequence smaller than it is started and its range_begin equals the currentOrderNumber it will commit.

Upon successful commit transactions will transactions will clear their logs, unlock all their orecs and move the currentOrderNumber up.

Rollback

Rollback is unique in this TM algorithm because one must rollback in the a strictly decreasing order. Given transactions 3,4,5,6,7,8 are running and transaction 4 begins and abort sequence. It will first obtain the abortFlag, then it will set the globalAbortNumber to itself and increase the globalAbortCount. It also sets its localAbortCount to the new globalAbortCount. The order of these operations is critical to ensure no transactions get locked into thinking an abort is happening when it may not be. Transactions will first check for a discrepancy between their localAbortCount and the globalAbortCount, given that discrepancy, they will then check to see if they are greater than the globalAbortNumber. If they are they will go to abort, if not they can continue as is. Setting these variables kicks off all rollback sequences. Once 4 sets this, all active transactions greater than 4 will also rollback. When a transaction sees the abortFlag is set and that it must abort, it will first unlock all of its orecs, then potentially spin to wait for its turn to abort. Once it confirms that the transaction above it has aborted it will abort. In this example, transaction 8 would have to abort, then 7, then 6, then 5, and finally 4. A transaction will rollback its own changes if there are changes to be made, and will set its own abortFlag to true. If a transaction is waiting to enter the active pool, but is in the transaction_list, its isAbort flag will already be set to true so it does not halt an abort sequence. Once this is complete these transactions will spin, and

4 will reach trycommit then release the abortFlag enabling these transactions to continue. This strict ordering is due to the fact that dependencies among transactions is very hard to monitor even in this case, this is the safest way to roll back and ensure correctness, without incurring a large amount of metadata overhead.

The transaction that initiated the abort sequence may not have to abort given the type of conflict is a read after read conflict. The code to end the abort sequence is found in the trycommit function, at this point the transaction has completed its work without conflict giving us progress. It will check the transaction list and ensure all active transactions with a range_begin greater than it have actually aborted. It will then set the globalAbortNumber to INT_MAX, update all other transactions localAbortCounts and set each transactions isAbort flag to false. This order is crucial because transactions spin after their abort as long as their isAbort is true and localAbortCount is different from the globalAbortCount. Once these transactions continue, one needs to guarantee that it won't accidentally enter an abort sequence again, which is why the globalAbortNumber is set to INT_MAX before any transaction is let loose again. This ensures that even if a transaction might see its localAbortCount different from the globalAbortCount it still will not abort, but would see that it is less than the globalAbortNumber and continue processing.

Evaluation

All development and testing was completed on an 64bit Ubuntu box with 8 Intel(R) Core i7-4770 chips that clock 3.40Ghz. Each core has an L1 cache of 32K, L2 cache of 256K and an L3 cache that is 8MB and runs at 800MHz. It utilized a core TBB version 4.3, with slight modifications to support a deterministic parallel_for loop, called a parallel_for_ordered loop here. An experimental version of GCC 4.9.0 was used for compilation and development, the core of LIBITM was also taken from this version of GCC.

The benchmark used in this experiment was fairly extensive. At a high level it created a main array of some argument specified size and would populate it with random numbers, this seed was provided via command line args. There was an auxiliary array, which was also populated with random numbers (again command line seeded), this auxiliary array was used to supply random seeds for the main array. The benchmark also had arguments to specify the amount of work for each iteration, there was a MIN_WORK value, which indicated the number of random numbers to compute. Additionally, there was a WORK_MOD argument, that would add some random amount of work to each iteration, by computing no more than the WORK_MOD many random numbers. The process was achieved by stepping through the main array, fetching the auxiliary array value found at the same index. Using that aux number to seed rand_r. It used the first number mod the WORK_MOD

value to be the varied work in that iteration. The bench then entered a loop that would loop from 0 to the MIN_WORK + VARIED_WORK value. This process gave great granularity in how the benchmark worked. It could construct jobs where each iteration had the same workload, workloads that were reasonably similarly, or workloads that were very different. These three signatures could be achieved by manipulating these work arguments.

This benchmark also enabled one to specify if the test should be overlapping or non-overlapping. The above process describes the type of work and amount of work that could be specified. This next stage specifies where to save the last random number generated from the above process. This was done through 4 arguments, a CHANCE_LEFT, CHANGE_RIGHT, DIST_LEFT, and DIST_RIGHT arguments. The chance variables are the odds that an iteration i would look at another index. The distance variables would specify how far from i the modification would take place. For example, it would compute one more random value, and see if it was less than or equal to the percentage chance provided. If it was the value would be saved in index, i - DIST_LEFT, given we had a CHANCE_RIGHT the index modified would be i + DIST_RIGHT. If one were to walk out of the array by modifying a value less than index 0 or greater than index length, this would make that index be the edge. So any negative became 0 and any number greater than length was made to equal length.

The process described above shows the process varied in workload, and varied in overlapping vs. non-overlapping. The benchmark also used PAPI [10] to obtain the hardware counters on Last Level Cache references and Last Level Cache Misses. The machine did not have L1 and L2 cache counters, which would have been much more useful for this experiment. Each execution of the benchmark ran one test in serial mode, and one in speculative parallel mode then compared run times and cache statistics. Looking at the Speed up Ratios which is found by (serial_time/specpar_time), so a score of 1.0 would be same time execution, a score of 2, would mean specpar runs in half the time of the serial. The expectation was that hardware transactions would perform the best and it was unclear which of the software implementations would be best.

Looking at the results in Appendix A: Table 1 one can see the assumption that hardware transactions did perform best. One area of caution is to monitor the grainsize used for htm, given larger grainsizes, it could turn this execution into an essentially serial execution. The next best performer was the lazy implementation, followed by the ordered ml and lastly OanceaLite. Looking back at the table presented above one can see these numbers are not promising for ordered speculative parallelism. It is not terribly surprising, non-ordered parallelism is often only useful in specific scenarios where there is enough work to go around and a proper management

system is in place to ensure threads avoid common problems such as deadlock, and thrashing the cache. To enforce an ordered, or deterministic output on a parallel execution is clearly even more difficult to do well. Many more controls are required to ensure all transactions do not clobber each other out of their work.

Another type of test was executed here, which was an overlapping test. The benchmark enabled one to specify the probability of editing an index to either the left or right of the current place. It also enabled the tester to specify how far left or right to look, this was intended to test editing items off our own cacheline. Given higher levels of conflict and transactions editing more than their own cache line and potentially fighting over indices with other transactions the expectation was that this would perform significantly worse than the non-overlapping test.

Observing Appendix A: Table 2 it is surprising to see that the overlapping results are very comparable if not nearly identical to the non-overlapping tests. The led to an investigation into the benchmark, and trying to discover if our non-overlapping implementation was still fighting over the cache, or if the overlapping implementation just performed to a really high level.

After more tests and some refinements, such as padding out structs to be some multiple of 256 bytes. A potential concern was that the items

transactions operate on, even though they are “independent”' may still live in the same cache line which would cause memory sharing, presenting serious problems to all the software transactions and causing the hardware transactions to abort. Another potential cause of bad performance is Intel's pre-fetching model, given one uses “memory block 1”', the Intel Memory management will pre-fetch the next memory block. This causes problems because the next cache line may be needed by another transaction. Even if this specific transaction only utilizes “memory block 1”', we may conflict with another transaction that only utilizes “memory block 2”' due to this pre-fetching.

Due to this problem the implementation was changed. The benchmark used a struct instead of a singular array index to separate our data by memory blocks. The struct was padded out to be 256 bytes (The size of 2 L1 Cache Blocks: hopes to prevent pre-fetching conflicts), running a series of tests with this addition did not show any improvement. Another padding attempt was made in libitm_htm's implementation. LIBITM_HTM may be the location where this shares memory. After padding the htm structs out and running more tests this did not give any sort of improvement. Lastly, it may have been how TBB was implemented, the shared TBB items may be causing this problem. After padding it out and running many more tests it was shown that this did not give any improvement.

Take Away

This paper describes a few different transactional memory algorithms, each method varies in its implementation and performance. It is clear that these methods may not be practical for use since each of them cannot compete with a serial execution. The best method presented here approaches serial execution time, and this is done by implementing the grainsize to act as if it is serial. The software TM algorithms perform even worse, some to a large magnitude worse than serial. But given this, it is clear that transactional memory algorithms are challenging, it is clear that deterministic parallelism is also challenging, and that hardware transactions out perform software transactions. It was obvious that this area of research would be difficult, if not impossible, to out perform pure non-deterministic parallel executions, and it might be possible to perform faster than a serial implementation. In order to perform very well, it appears that one must know their hardware very well. For example, this area of research was heavily influenced by Intel's pre-fetching standard. This research attempted to make a general solution that could work on any machine, but that may not be possible at this time. Transactional Memory algorithms are highly dependent on the hardware and if one is to optimize their transactional memory algorithm they must be in tune with the hardware, more specifically with the cache.

Lastly, this research proved that one can quickly implement and test transactional memory algorithms in a deterministic environment and a non-deterministic environment. Utilizing GCC's LIBITM library and Intel's TBB library give other researchers a means to conduct research in transactional memory algorithms without reinventing the wheel. Researchers can utilize these two very large and standard libraries to investigate ideas about transactional memory. A great positive to this framework is given positive results, one does not have to make many changes to distribute this to the world. GCC and TBB are used by many people for transactional memory and speculative parallelism.

Conclusion and Future Work

Overall the work presented here is useful to the transactional memory field since this provides a simple testing ground for transactional memory algorithms. In a short period of time this research was able to look at 4 different transactional memory algorithms with very little overhead. This is a great example for the research community to quickly prototype a TM algorithm with little to no cost. This work is also impactful because it utilizes two very major libraries being GCC and TBB, even though these tests may not provide great performance results, they provide an impactful experiment that

works with two major standards in the speculative parallelism and transactional memory research area.

This research as many opportunities for future work. One potential option to move this research forward is to record each transactions beginning and ending orec. Given this range one can see if any transaction orec ranges overlap, if they do, checks must be made across orecs to ensure no conflict. If there is no overlap, it could save much of the overhead seen here, it is not clear how many cases this would be true for. Given a standard program that concerns itself with locality it may cause this modification to provide very marginal gains. Another potential improvement could be properly implementing the Oancea algorithm [8], the difficulty presented here was making this algorithm more general to work on all types of machines. It also looked to make a hybrid of oancea and the standard libitm ml implementation. This research could have also enabled transactions to work and move on instead of waiting for their commit turn as they do in the Oancea paper. Since there is no guarantee on transaction locality it put this research in a place where one transaction might have to work on numerous iterations on its own, lacking any parallelism. Another possible improvement is utilizing the range object, in this research the range was not modified in anyway, but one could get a range from TBB and split that range into more manageable sizes given observations on transaction performance. For example, if TBB gave ranges

that contained 100 iterations and that was found to be a lot of work per transaction, the TM algorithm could have split that range into 5 chunks of 20 to better divide the work presented. Having an intelligent work distributor could optimize the load for each transaction and properly separate the work so the transactions do not fight over the same cache lines. Finding better statistics on the L1, L2 and L3 cache would also be useful to exactly identify the main source of poor performance.

References

1. J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.
2. P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.
3. M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.
4. Intel Corporation. Threaded Building Blocks. Available as www.threadingbuildingblocks.org.
5. Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.
6. C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.
7. Y. Lev and J.W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel*

Programming, Salt Lake City, UT, Feb. 2008.

8. C. E. Oancea, A. Mycroft, and T. Harris. A Lightweight In-Place Implementation for Software Thread-Level Speculation. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.
9. M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.
10. University of Tennessee. PAPI: Performance Application Programming Interface. Available as http://icl.cs.utk.edu/papi/.
11. A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

Appendix A

Listing 1: HTM Metadata

Thread Variables

range_begin : Integer - Number specifying the start iteration number of the parallel_for loop from TBB

range_end : Integer : - Number specifying the last iteration number +1 of the parallel_for loop from TBB

range_grainsize : Long Unsigned - Number of iterations a thread should handle, specified in parallel_for from TBB

Global Variables

abortFlag : Atomic Boolean - Flag to stop all transactions from working and ensure progress is made

currentTxNumber : Atomic Integer - Number that identifies the range_begin of the transaction that is next to commit

retryCount : Integer - Upper bound on retries before acquiring global lock to make progress

waitFraction : Double - Number to multiply by the prior transaction time to be set as the backoff time

Algorithm 1: Begin and Commit Instrumentation for HTM

Algorithm 1: Begin and commit intstumentation for HTM

```
function TxBeginHTM()
1   while true do
    // Will backoff for backoffFraction * last tx cycles time
2   if backoff == true then
        backOff()
    // If not next and abortFlag stop working and spin
3   while abortFlag ∧ tx.range_begin ≠
    currentOrderNumber do spin
4   xbegin()
5   if XBEGIN_STARTED then
6       if tx.range_begin ≠ currentOrderNumber then
7           backoff = true
8           xabort()
        // If next and spinlock is free, can continue to commit
9       else if spinlockisfree then
10          break
        // If next and spinlock is taken, need to wait, stop all
        others
11      else
12          backoff = true
13          abortFlag = 1
14          xabort()
15  else
16      if ABORT.EXPLICIT then
17          if tx.range_begin ≠ currentOrderNumber
            then
18              backoff = true
19              xabort()
20          else
21              if spinlockheldbyother then
22                  backoff = true
23                  abortFlag = true
24      else if ABORT.CONFLICT then
25          backoff = true
26      else if ABORT.CAPACITY then
27          backoff = true
28          abortFlag = true
29      else
30          backoff = true

function TxCommitHTM()
1   if abortFlag == true then
2       abortFlag = false
3   currentOrderNumber = tx.range_end
```

Listing 2: Lazy Metadata

Thread Variables

range_begin : Integer - Number specifying the start iteration number of the parallel_for loop from TBB

range_end : Integer : - Number specifying the last iteration number +1 of the parallel_for loop from TBB

range_grainsize : Long Unsigned - Number of iterations a thread should handle, specified in parallel_for from TBB

commit_count : Integer Unsigned – Count of number of transactions that have already committed

redo_log : Binary Search Tree - Binary Search Tree containing key value pairs of [MemoryAddress, Value]

Global Variables

currentTxNumber : Atomic Integer - Number that identifies the range_begin of the transaction that is next to commit

Algorithm 2: commit Instrumentation for LAZY

Algorithm 2: Commit intstumentation for LAZY

```
   function TRYCOMMIT()
1      if tx.redolog_bst.isEmpty() then
2          clear readlog
3          currentOrderNumber = tx.range_end
4          return true
5      for all redolog items do
6          write changes to memory locations
7      Get a commit time
8      Validate no one interleaved and committed since start
9      if Invalidated then
           returnfalse
10     for each in writeLog do
           store
11     clear all logs
12     return our commit time
13     currentOrderNumber = tx.range_end
14     return true
```

Listing 3: ML Metadata

Thread Variables

range_begin : Integer - Number specifying the start iteration number of the parallel_for loop from TBB

range_end : Integer : - Number specifying the last iteration number +1 of the parallel_for loop from TBB

range_grainsize : Long Unsigned - Number of iterations a thread should handle, specified in parallel_for from TBB

Global Variables

abortFlag : Atomic Boolean - Flag to stop all transactions from working and ensure progress is made

currentTxNumber : Atomic Integer - Number that identifies the range_begin of the transaction that is next to commit

Listing 3: OanceaLite Metadata

Thread Variables

range_begin : Integer - Number specifying the start iteration number of the parallel_for loop from TBB

range_end : Integer : - Number specifying the last iteration number +1 of the parallel_for loop from TBB

range_grainsize : Long Unsigned - Number of iterations a thread should handle, specified in parallel_for from TBB

local_abort_count : Atomic Integer – The transaction local count of abort sequences that have occured

is_abort : Atomic Boolean – The flag that indicates if this transaction has aborted during this sequence

is_active : Atomic Boolean – The flag that indicates if this transaction as entered the transaction pool

master_index : Integer – The number in the master array that this transaction should log its reads and writes

Hybrid_Word Variables

lock : Atomic Boolean – A Boolean that indicates if this word is being editted

read_log[] : Atomic Integer Array – An array of length: maximum transaction count. Holds the record of who has read this orec

write_log : Atomic Integer Array – An array of length: maximum transaction count. Holds the record of who has written this orec

Global Variables

abortFlag : Atomic Boolean - Flag to stop all transactions from working and ensure progress is made

currentTxNumber : Atomic Integer - Number that identifies the range_begin of the transaction that is next to commit

globalAbortNum : Atomic Integer – The number that identifies the range_begin of the transaction that started the abort sequence

globalAbortCount : Atomic Integer - The number that counts how many abort sequences have happened

Algorithm 3: Pre-Write, TryCommit and Roll-Back instrumentation for OanceaLite

Algorithm 3: Pre-Write, TryCommit, Roll-Back intstumentation for OANCEALITE

```
   function TxWriteOANCEA()
1      checkAbortSequence()
2      for eachorectoload do
           // Tries to lock orec, if exceeds a TRYLIMIT will
           abort and begin an abortSequence
3          acquireOrec()
4          if is_write_conflict() then
5              release_current_orec()
6              abortProtocol()
7          else if is_read_conflict() then
8              release_current_orec()
9              abortProtocol()
10         else
               // Success: No conflict found
               // Saves the orec memory location and prior
               value there
11             tx.add_to_writelog()
               // Puts own number into exploded orec in its
               owned master_index
12             orec.write_log[tx.master_index] =
               tx.range_begin
13     tx.undo_log.log()

   function TxCommitOANCEA()
1      if abortFlag == true then
2          abortFlag = false
3      currentOrderNumber = tx.range_end

   function TxRollbackOANCEA()
1      if abortFlag == true then
2          abortFlag = false
3      currentOrderNumber = tx.range_end
```

Table 1: Non-overlapping Benchmark Results

Method	**Average Speedup**
HTM	0.8642
LAZY	0.1631
ML	0.1684
OANCEALITE	0.0079

Table 2: Overlapping Benchmark Results

Method	**Average Speedup**
Overlapping Look-ahead	
HTM	0.8149
LAZY	0.1517
ML	0.1672
OANCEALITE	0.0073
Overlapping Look-behind	
HTM	0.8112
LAZY	0.1518
ML	0.1673
OANCEALITE	0.0073
Overlapping Look-both	
HTM	0.8136
LAZY	0.1567
ML	0.1669
OANCEALITE	0.0076

Vita

Stephen Louie was born to Robert and Juliet Louie in Middletown New Jersey in November of 1991. He graduated from Red Bank Catholic High School in 2010 and attended Lehigh University for his undergraduate work. In 2014 he graduated from Lehigh with a Bachelors of Science in Integrated Business and Engineering where he focused on Computer Science. Stephen continued his education at Lehigh and is working towards a Masters of Engineering in Computer Science, to be completed May 2015. During his time in college Stephen held three summer internships, the first in the summer of 2012 at Northrop Grumman, here he worked in the Space Software Division. During the summer of 2013 he was an intern at Becton Dickinson and in 2014 he was an intern at Microsoft. Stephen has signed to work at Cisco Systems as a full time software engineer starting in the summer of 2015.