Adaptive Software Transactional Memory: A Dynamic Approach to Contention Management

Joel C. Frank, M.S.
Robert Chun, Ph.D.
Computer Science Department
San Jose State University
San Jose, California 95192
Robert.Chun@sjsu.edu

Abstract

Effectively managing shared memory in a multi-threaded environment is critical in order to achieve high performance in multi-core hardware platforms. Software Transactional Memory (STM) is a scheme for managing shared memory in a concurrent programming environment. STM views shared memory in a way similar to that of a database; read and write operations are handled through transactions, with changes to the shared memory becoming permanent through commit operations. Furthermore, its benefits are not attained until larger data structures are used. Currently there are varying methods for collision detection, data validation, and contention management, each of which has different situations in which they become the preferred method. This paper discusses problems surrounding contention management, related work addressing these problems, a new dynamic contention manager algorithm yielding an Adaptive STM (ASTM) library, experimental results comparing static versus dynamic contention management, and an analysis of the result.

Keywords: Software Transactional Memory, Contention Management, Multithreaded Software Performance, Reinforcement Learning.

1. Introduction

In the last decade, physical limitations, the two most prominent being heat and space limitations, have caused hardware designers to push for multi-core implementations in order to achieve increases in performance. As a result, software that runs efficiently on these new multi-core platforms has become increasingly important, and the major factor that determines the efficiency of multi-threaded software is how that software manages shared memory.

The historical method for protecting shared memory is to simply only allow one process access, read or write, at a time. This is guaranteed through the proper use of a locking mechanism that ensures mutual exclusion, a mutex. However, implementing a mutex in such a way that it indeed guarantees mutual exclusion, does not cause deadlocks, livelocks or starvation, is easy to debug, and does not cause priority inversion is quite difficult. Furthermore, even if all of these features are implemented properly, mutexes still limit scalability of an application due to its forced serialism.

The ultimate goal of STM is similar to that of mutexes, specifically the safe management of shared memory in order to prevent data corruption. However, the main difference between STM and mutexes is that STM is lock free. STM views shared memory in a way similar to that of a database; read and write operations are handled through transactions, with changes to the shared memory becoming permanent through commit operations. STM also shifts the responsibility of not adversely affecting other operations from the writer, which is the case with mutexes, to the reader, "who, after completing an entire transaction, verifies that other threads have not concurrently made changes to memory that it accessed in the past" [2]. This stage is called data validation, and if successful, allows the changes to be made permanent through a commit operation. The various techniques for data validation and collision detection are discussed later.

1.1 Advantages and Disadvantages of STM

As previously stated, STM is lock free, which removes most of the negative aspects of mutexes. However, STM is also much more efficient at allowing parallel operations on non-primitive data structures. Assume the shared data structure is a 10,000 node tree. Typically, multiple processes accessing the tree are not modifying the same part of the tree concurrently. As a result, there is no reason to lock the entire data structure when only a small number of nodes within the tree are being accessed at any one time. When using mutexes, the entire data structure is locked, which serializes what otherwise could be a fully parallel series of read and/or write operations. Under STM, because only individual nodes are checked out, the same series of read and/or write operations would be fully parallel (i.e. no process is forced to wait on any other process).

The first disadvantage of STM pertains to the overhead required to perform transactions on the shared memory. When primitive data types are used, the overhead required by STM for collision detection and data validation, causes it to degrade in performance below that attained by implementing mutexes. The second disadvantage, or rather challenge, is the complexity of both implementation and Application Programming Interface (API) use. It is for this reason that there is a large drive to develop a standard implementation of an STM library with an easy to use API. If one was developed, it would allow STM to overcome all of the limitations and drawbacks of mutexes.

1.2 Problem Addressed

Contentions arise when two competing transactions attempt to access the same block of memory. In these situations, at least one of the processes must be aborted. Deciding which process to abort is called contention management. "Contention management [in STM] may be summed up as the question: what do we do when two transactions have conflicting needs to access a single block of memory?" [3]. There are many different accepted contention management schemes. These range from Aggressive, which simply causes the conflicting process to abort its transaction, to Exponential Back Off, where the conflicting process temporarily aborts its transaction and re-attempts its commit transaction after exponentially increasing wait periods. Each of the contention management schemes is optimal for a corresponding application. Optimal in this context refers to the highest possible successful transaction rate. The problem arises because the act of choosing which contention manager to use is highly dependant on the type of data structure being accessed, for example primitive versus non-primitive, as well as the rate of transaction requests. It is for this reason that there is no contention management scheme that is optimal in all situations. It is the goal of this paper to develop a dynamic contention manager that adapts to the shared memory application in order to maintain an optimal rate of successful transactions by automatically applying the proper contention management scheme for the particular application.

2. Related Work

2.1 Non-Blocking Synchronization Algorithms

There are three standard non-blocking synchronization algorithms: wait-freedom, lock-freedom, and obstruction-freedom [4]. Each of these algorithms keeps processes from waiting, i.e. spinning, in order to gain access to a block of shared memory. As opposed to waiting, a process will either abort its own transaction, or abort the other transaction with which it is in contention. In contrast, algorithms that utilize a blocking scheme use mutexes to guard critical sections, thereby serializing access to these objects.

Wait-freedom has the strongest property of the three algorithms in that it guarantees that all processes will gain access to the concurrent object in a finite number of their individual time steps. As a result, deadlocks and starvation are not possible under wait-freedom algorithms.

Lock-freedom is slightly weaker in that it guarantees that within a group of processes contending for a shared object, at least one of these processes will make progress in a finite number of time steps. It is evident that lock-freedom rules out deadlock, but starvation is still possible.

Obstruction-freedom is the weakest of the three algorithms in that it guarantees that a process will make progress in a finite number of time steps in the absence of contention. This algorithm makes deadlocks not possible, however livelocks may occur if each process continually preempts or aborts the other contending processes, which results in no process making progress. It is for this reason that design and selection of contention management schemes is critical in order to ensure livelocks do not occur.

2.2 Hash Table STM Design

One of the original designs for STM made use of a hash table to store records relating to each of the active transactions. Figure 1 shows the schematic design of the STM system proposed by Harris and Fraser [4]. This design consists of three main components: the Application Heap, which consists of the blocks of shared memory that holds the actual data, the hash table of ownership records, and the transaction descriptors, which consists of a transaction entry for each of the shared memory locations to be accessed by the transaction.

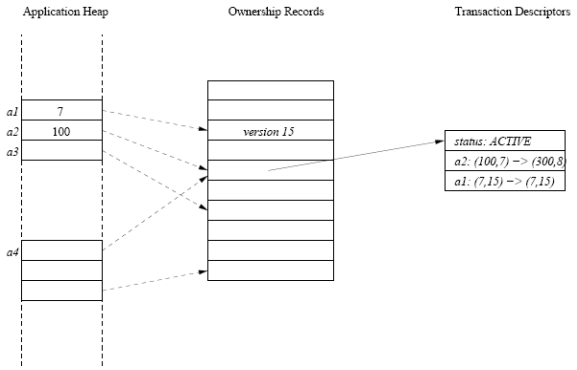

Figure 1. Heap Structure showing an active transaction

Each of the shared memory locations in the heap hashes to one on the ownership records. As a result, when a transaction owns an ownership record, it semantically owns all of the shared memory locations that hash to that ownership record (orec). During read or write operations, a process creates a transaction entry that corresponds to the shared memory location to be accessed. However, a process does not try to take ownership of the orec at this time; ownership occurs during the atomic Compare And Swap (CAS) operation. It is evident that this early design

has several drawbacks. Ownership, and subsequently access, of shared memory blocks is limited to the blocks accessible by each of the orecs. As a result, blocks of memory not required for a transaction are now unnecessarily locked during a commit transaction simply because they hash to the same value as the block of memory that is actually needed. Second, the size of the STM is static and cannot be resized during runtime without considerable overhead, which is due to suspending all transactions in order to allow the hash table to empty upon completion of all transactions and then recreating the hash table based on the new required data size.

2.3 Object Based STM Design

In order to overcome the limitations of STM systems that are similar to the hash table design, as well as keeping with the current OOP / OOD standard, the most widely accepted implementation for an STM library is the object based STM system for dynamic data structures [1]. This obstruction-free STM system is commonly referred to as DSTM, dynamic software transactional memory, which manages a collection of transactional objects (TM objects). These TM objects are accessed by transaction, which are temporary threads that either commit or abort.

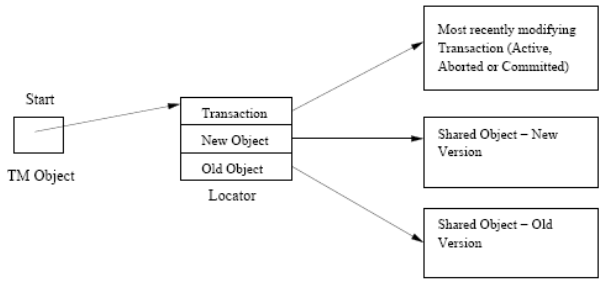

Figure 2. Transactional Memory Object Structure

Figure 2 shows the structure of a dynamic TM Object, which acts as a wrapper for each concurrent object in the data structure; these objects are simply normal Java objects, which greatly increase the flexibility of the design [4]. TM objects can be created at any time, and furthermore, the creation and initialization are not part of any transaction. The extra layer of abstraction introduced by the Locator object is required in order to essentially shift the three references, transaction status, old, and new data objects, in a single CAS operation. This can now be done by creating a new Locator object, which contains copies of the data objects, for the transaction, and then performing a CAS operation on the TM object's start reference from the old locator object to the new one.

Figure 3 shows an example implementation of DSTM using a linked list of objects to hold integers [1]. The IntSet class uses two types of objects: nodes, which are TM objects; and List objects, that are standard Java linked objects containing an integer and a reference to the next object in the linked list. It is of note, however, that the reference to the next object is of type Node, which is a TM object. This is required for the list elements to be meaningful across transactions.

The interesting work is done in the insert method. The method takes the integer value to be inserted into the linked list, and returns true if the insertion was successful. The method repeatedly tries to perform an insertion transaction until it succeeds. During the transaction, the list is traversed while opening each node for reading until the proper position in the list is found. At that point, the node is opened for writing and the new TM node is inserted into the list. If the transaction is denied, by throwing a Denied exception, the transaction calls commitTransaction in order to terminate the transaction; this is done even though it is known that the commit action will fail.

```
public class IntSet {
  private TMObject first;

  class List implements TMCloneable {
    int value;
    TMObject next;

    List(int v) {
      this.value = v;
    }

    public Object clone() {
      List newList = new List(this.value);
      newList.next = this.next;
      return newList;
    }
  }

  public IntSet() {
    List firstList = new List(Integer.MIN_VALUE);
    this.first = new TMObject(firstList);
    firstList.next =
     new TMObject(new List(Integer.MAX_VALUE));
  }

  public boolean insert(int v) {
    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread =
             (TMThread)Thread.currentThread();
    while (true) {
      thread.beginTransaction();
      boolean result = true;
      try {
        List prevList =
          (List)this.first.open(WRITE);
        List currList =
          (List)prevList.next.open(WRITE);
        while (currList.value < v) {
          prevList = currList;
          currList =
            (List)currList.next.open(WRITE);
        }
        if (currList.value == v) {
          result = false;
        } else {
          result = true;
          newList.next = prevList.next;
          prevList.next = newNode;
        }
      } catch (Denied d){}
      if (thread.commitTransaction())
        return result;
    }
  }
  ...
}
```

Figure 3. Integer Set Example of DSTM

2.4 Contention Managers

"A contention manager is a collection of heuristics that aim to maximize system throughput at some reasonable level of fairness, by balancing the quality of decisions against the complexity and overhead incurred" [5]. Simply stated, contention managers tell a transaction what to do when they encounter a conflicting transaction. There are several different schemes to perform contention management, and since the overall implementation is obstruction free, it is the responsibility of the contention manager to ensure that livelocks do not occur.

The simplest contention manager is the Aggressive contention manager. Whenever this manager detects a contention, it simply aborts the opposing transaction.

The most common contention manager is the Backoff manager. When a contention occurs, it follows an exponential back off pattern to spin for a randomized amount of time with mean 2^{n+k} ns, where n is the number of times the conflict has occurred and k is a provided constant. There is also an absolute limit, m, to the number of rounds a transaction may spin. From empirical testing, it has been found that values of k = 4 and m = 22 result in the best performance [4].

Another contention manager is the Karma manager. This manager decides who gets aborted by how much work each of the transactions has done so far. Although it is difficult to quantify the relative work of a transaction, the number of objects that it has opened so far is a rough indicator. The rationale behind this idea is that, in general, it makes more sense to abort a transaction that has just begun processing its changes, as opposed to one that is just about to complete its transaction. In essence, this is a priority manager where the number of objects opened thus far by the transaction is its priority. This priority is not reset when a transaction aborts, which allows shorter transactions to eventually overcome longer ones [4].

The Eruption manager is based on the idea that the more transactions blocked by a particular enemy transaction, the higher priority that enemy transaction should have. As a result, this manager is a variant of the Karma manager, whereas the priority of a transaction is based on the number of objects it has opened. However, in the Eruption manager, a blocked transaction adds its priority to that of the blocking transaction. Therefore, intuitively the more transactions that are being blocked, the faster the blocking transaction will finish [4].

The final base contention manager is the Greedy manager, which makes use of two additional fields in each transaction: a timestamp, where an older timestamp indicates a higher priority; and a Boolean to indicate whether the transaction is currently waiting on another transaction. Whenever a contention arises, if the opposing transaction has a lower priority, or it is currently waiting on another transaction, then the opposing transaction is aborted. Otherwise the current transaction will wait on the opposing one. These rules hold as long as the transaction wait times are bounded [5].

There are several other hybrid contention managers, but the ones presented here constitute the core of contention manager schemes. For example, one of these hybrid managers, the Polka manager, combines the positive aspects of Backoff and Karma [4].

2.5 Reinforcement Learning

There are many different types of machine learning algorithms; however it is reinforcement learning that is most applicable to this application. Reinforcement learning can be described as learning how to map situations to actions so as to maximize a numerical reward signal [13]. This method of machine learning is essentially characterized by trial and error. The machine is not told explicitly which actions to take in each situation, but rather determines the best course of action by interacting with the environment according to the current policy, and evaluating the reward received based upon other potential rewards. "Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision-making. It is distinguished from other computational approaches by its emphasis on learning by the individual from direct interaction with its environment, without relying on exemplary supervision or complete models of the environment" [13].

3. Adaptive Contention Management

The new dynamic contention manager algorithm described in this paper, Adaptive Contention Management (ACM), uses a reinforcement learning algorithm to select the proper contention manager in order to maximize the reward. In context of this paper, reward is the average number of successful transactions that have been completed since the last evaluation period. For the purposes of this discussion, better performance of the system will result in a higher reward. The ACM tracks the average historical performance observed for each contention manger. This stored historical performance is updated during each evaluation period. The pseudo code describing the algorithm is shown in figure 4.

The algorithm starts by putting the current thread to sleep for a constant amount of time. (Note - the AdaptiveCM class implements Runnable. See Section 4 for design details). During experimentation, it was found that a sleep time of one second worked well to balance the difference between making the time too long, which would slow down the responsiveness of the manager, versus making the time too short, which would decrease performance due to excessive polling.

Polling is accomplished by requesting the transaction counters from the base class of all testing threads within the library. Three separate counters are maintained by the dstm2.Thread class for each of the three types of

transactions: insert, remove, and contains. Each of the derived test threads updates these counters asynchronously whenever the proper type of transaction is successful. This was done to increase performance and limit unnecessary serialization of the testing threads.

```
while ( !finished )
{
        sleep( SLEEP_PERIOD);

        poll_For_Successful_Transactions();

        if ( intervalsSinceLastEval > INTERVALS_BTW_EVAL )
        {
                currentPerformance = calculateCurrentPerformance();

                historicalPerformance[currentCM] = currentPerformance;

                found =  findBetterCmThan(currentPerformance);

                if ( found )
                {
                        switch_to_better_CM()

                        notifyListeners();
                }
                else
                {
                        randomNumber = generateRandomPercentage();

                        if ( randomNumber > CHANCE_TO_SWICH )
                        {
                                switchToRandomCM();

                                notifyListeners();
                        }
                }
        }

        notifyListenersOfCurrentPerformance();

        clearTransactionCounters();
}
```

Figure 4. ACM Psuedo code

In order to further limit the impact the adaptive algorithm has on the base library, evaluation periods were limited such that they would not occur during each polling period. The value for this evaluation interval is primarily based on how dynamic the data structures being accessed are. The more often the data structures are changing size, the lower this evaluation interval should be set. For this experiment, since the data structures were relatively constant in size, the evaluation interval was set to five. Overall, this caused the adaptive manager to only evaluate performance once every five seconds.

The first step of evaluation is to calculate the current performance, which is an average of the number of successful transactions for the elapsed amount of time. This performance is then stored for future comparisons. Initially the historical performance for each contention manager is set to MAX_INTEGER. This forces the adaptive contention manager to try each available contention manager at least once, resulting in the required trial and error behavior of a reinforcement learning algorithm. If a contention manager is found that has better historical performance than is currently being seen, the new manager is now used and all listeners are notified of the change. If a better contention manager is not found, a random percentage value is generated and compared to the threshold value. If exceeded, a random contention manager is chosen and set, and all listeners are notified of the change. This random element was introduced to keep the adaptive contention manager from getting locked into a single contention manager at steady state while environmental conditions have changed, which would now cause a different contention manager to be able to outperform the manager that is currently set. The threshold for this random switch should be proportional to the volatility of the current environment. During these experiments, the threshold was set to 25%; however, due to the non-volatile nature of the test environment, the random behavior was not found to affect overall performance.

The last steps of the core algorithm simply post the current performance to all listeners and clear the transaction counters in preparation for the next iteration. All listener notification for inter-thread communication is done via a mailbox class that is only blocking to registered listeners. All inter-thread communications are non-blocking to the adaptive contention manager.

4. Experimental Results

4.1 Impact of Polling Modifications

The initial task of the experiment was to determine the impact, if any, that the modifications to the base DSTM2 library had on the overall performance. To test this, the original, unmodified library was run on each of the data structures, or benchmarks, which included the linked list, list release, and red-black tree. In an effort to simulate an industry-like environment, each trial run was set to a 10% update ratio. This means that 10% of the transactions were insert or remove calls, where the remaining 90% were contains type calls. The trials were then run ten times to obtain an average performance as seen on a varying number of threads. Again, this was done to simulate a real environment. The trials were repeated using one, ten, thirty, and fifty threads. The number of threads chosen was based on related experiments [1] that showed relative maxima in performance when thirty threads are used. Lastly, each trial run was repeated for each of the five core types of contention managers; these included Backoff, Aggressive, Eruption, Greedy and Karma. Once this baseline data was obtained, the same experiments were run with the AdaptiveCM, but with its adaptive algorithm turned off. For each trial, it was set statically to one of the five contention managers, and then each of the experiments from the baseline testing was repeated. The following composite graph shows the comparison of baseline data to static ACM performance using a linked list for all contention managers. Similar results were also found on the other data structures, list release and red-black tree.

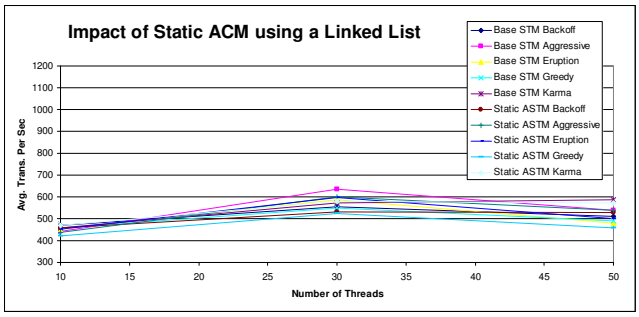

Figure 5. Baseline vs. Static ACM

In general, it can be seen that the modifications to the DSTM2 library had no significant effect on overall performance. However, for clarity, figure 6 has been included to show the above graph filtered to just show the Backoff CM (Contention Manager) data. It clearly shows the negligible impact of the polling modifications.

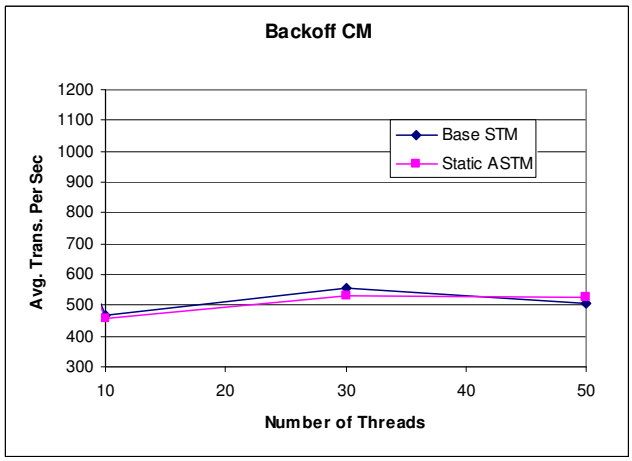

Figure 6. Baseline vs. Static ACM using Backoff CM

4.2 ASTM Performance

In order to evaluate the performance of the Adaptive STM (ASTM) library, it was tested under various conditions until reaching steady state, which was reached after no more shifts in contention manager selection were observed. Since the ASTM is by nature switching the contention manager that is currently in use, tracking which contention manager is in use over time was not important. Of critical importance, however, is the performance over time while the ASTM is attempting to reach steady state. In order to determine this, the ASTM was run adaptively on each of the benchmark data structures, and using the same thread values, from the previous section. It was also repeated ten times per trial in order to get an average performance for each of the test conditions. The following graph shows the ASTM's performance over time. The upper and lower bounds displayed on the graphs was found by using the maximum and minimum performance obtained by best and worst contention managers respectively as seen in the previous section. Due to similar results being obtained for each of the data structure, redundant graphs, which included the trials using various number of threads on each of the data structures, have been removed. The vertical line on the right side of each graph indicates the point in time when steady state was reached. Steady state refers to the point when the ACM stopped changing the contention manager currently in use.

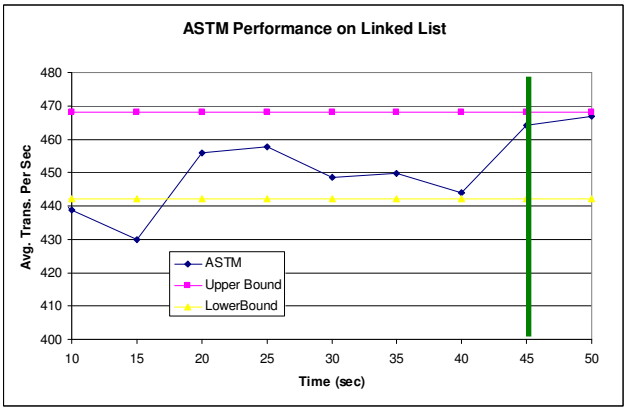

Figure 7. - ASTM Performance using 10 threads (Steady State CM = Eruption)

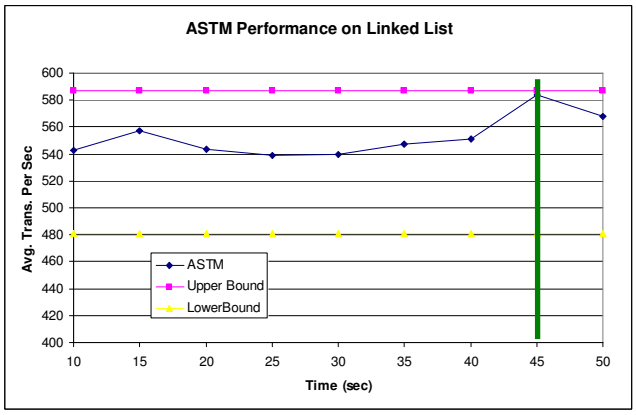

Figure 8. - ASTM Performance using 50 threads (Steady State CM = Aggressive)

As seen in the previous graphs, ASTM's performance quickly adapts to near that of the upper bound. The best performance was seen when using a linked list. In these cases, the ASTM adapted to within 4% of the upper bound. The average steady state performance of the ASTM, as seen across all threads counts and data structures, was found to be within 12% of the upper bound.

5.0 CONCLUSION

Despite the slight overhead that polling imposes, using an adaptive approach to contention management will guarantee in all cases that the contention manager that yields the highest possible performance will be used. In an ideal case, the ASTM library would not be needed.

However, since the ideal contention manager cannot be statically chosen correctly, the ASTM yields a higher average performance over time. Furthermore, in a real production type environment, not only would the size of data structure be changing often, but also the type of data structure as well. This volatile environment makes it impossible to correctly choose the ideal contention manager. ASTM, however, is not burdened by these limitations since it is adaptive.

ASTM also yields a more consistent performance. Each baseline experiment was run ten times, which resulted in roughly a 54% variance. When compared to the ASTM's average variance of 7%, it is apparent that the ASTM yields not only a higher average performance, but a much more stable one as well. This variance is further compounded by differences in performance from one machine to another.

6.0 FUTURE WORK

The work presented in this paper shows that a dynamic approach to contention management outperforms that of static implementations. However, there are several areas in which this algorithm may be improved.

Making the learning algorithm more predictive is one way in which the ASTM library could be improved. The current implementation only monitors the performance of the system as the reward for reinforcement learning. Perhaps there are other benchmarks that could be used in order to predict which contention manager will be the ideal one. This would greatly reduce the time required for the adaptive contention manager to reach steady state.

There is currently ongoing work to both improve the core contention managers and to create new contention managers that outperform those currently known. If these new contention managers far outperformed the common contention managers, the ASTM would greatly benefit by being able to utilize these contention managers, even if they are highly specialized and optimized for a narrow range of applications, when appropriate.

Shifting some of the transactional load to hardware is another way that STM in general may be improved. The high overhead of STM, due to the additional layers of object abstraction on top of the base memory required for transaction processing, could be mitigated, or at least greatly reduced, by developing STM friendly memory in hardware. Memory architectures that were specifically designed for STM greatly reduce the need for complex software architectures that, in essence, force the standard memory architecture to accomplish something for which it was not designed.

7.0 REFERENCES

[1] Maurice Herlihy, Victor Luchangco, Mark Moir, III William N. Scherer, *Software Transactional Memory for Dynamic-Sized Data Structures*. 2003.

[2] Scherer, W.N. III, Scott, M.L., *Contention Management in Dynamic Software Transaction Memory*. 2004.

[3] Virendra J. Marathe, Michael L. Scott. *A Qualitative Survey of Modern Software Transactional Memory Systems*. 2004.

[4] William N. Scherer III, Michael L. Scott. *Advanced Contention Management for Dynamic Software Transactional Memory*. 2005

[5] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, Michael L. Scott. *Lowering the Overhead of Software Transactional Memory*. Tech. Report Nr. TR 893. 2006.

[6] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon. *Toward a Theory of Transactional Contention Managers*. 2006.

[7] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, Bratin Eswaran Saha. *Unlocking Concurrency: Multicore Programming with Transactional Memory*. ACM Queue, 4(10):24—33. 2006.

[8] Torvald Riegel, Christof Fetzer, Pascal Felber. *Time-based Transactional Memory with Scalable Time Bases*. 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 2007.

[9] Yossi Lev, Mark Moir, Dan Nussbaum. *PhTM: Phased Transactional Memory*. Workshop on Transactional Computing (TRANSACT). 2007.

[10] Robert Ennals. *Efficient Software Transactional Memory*. Technical Report Nr. IRC-TR-05-051. Intel Research Cambridge Tech Report. 2005.

[11] Richard S. Sutton and Andrew G. Barton. *Reinforcement Learning: An Introduction*. The MIT Press. 1998.

[12] Kaelbling, L.P., Littman, M.L., and Moore, A.W. *Reinforcement Learning: A Survey*. Vol 4, pages 237-285. 1996.

[13] Sun Microsystems. *Dynamic Software Transactional Memory Library 2.0*. Retrieved from http://www.sun.com/download/products.xml?id=453fb28e on August 2007.