San Jose State University
SJSU ScholarWorks

2008

Adaptive software transactional memory : dynamic contention management

Joel Cameron Frank
San Jose State University

ADAPTIVE SOFTWARE TRANSACTIONAL MEMORY: DYNAMIC CONTENTION MANAGEMENT

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Joel Cameron Frank

May 2008

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Robert Chun

Dr. Robert Chun

Dr. Mark Stamp

C Pollett

Dr. Chris Pollett

APPROVED FOR THE UNIVERSITY

Then I. Williamson 04/22/08

ABSTRACT

ADAPTIVE SOFTWARE TRANSACTIONAL MEMORY: DYNAMIC CONTENTION MANAGEMENT

by Joel Cameron Frank

This thesis addresses the problem of contention management in Software Transactional Memory (STM), which 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.

Research on this subject reveals that there are currently varying methods for collision detection, data validation, and contention management, each of which has different situations in which they become the preferred method.

This thesis introduces a dynamic contention manager that monitors current performance and chooses the proper contention manager accordingly. Performance calculations, and subsequent polling of the underlying library, are minimized. As a result, this adaptive contention manager yields a higher average performance level over time when compared with existing static implementations.

Table of Contents

List of Figures

1.0 Introduction

This section gives an overview of shared memory management schemes, including Software Transactional Memory (STM), and argues its advantages over mutexes when using non-primitive data structures. The problem addressed by this paper is discussed in section 1.3.

1.1 Shared Memory Management

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 to 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 does guarantee 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". (Wickipedia) 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.

1.2 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 to STM pertains to the overhead required to perform transactions on the shared memory. When primitive data types are used, the overhead of STM, which is required for collision detection and data validation, causes it to degrade the performance below that attained by implementing mutexes. The second disadvantage, or rather challenge, is the complexity of both implementation and 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.

1.3 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?" (Scherer et al. 2003) 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 has a corresponding application, based on data structure in use, rate of transaction request, etc.., where it is the optimal scheme to use. 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 applying the proper contention management scheme for the application.

2.0 Related Work

This section describes previous work related to STM, as well as an overview of standardized approaches to learning algorithms. It is of note that there is concurrent work developing transaction memory implemented in hardware; however this paper focuses only on the software implementations.

2.1 Non-Blocking Synchronization Algorithms

There are three standard non-blocking synchronization algorithms: wait-freedom, lock-freedom, and obstruction-freedom. (Marathe and Scott 2004) 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 to 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. (Marathe and Scott 2004) 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.

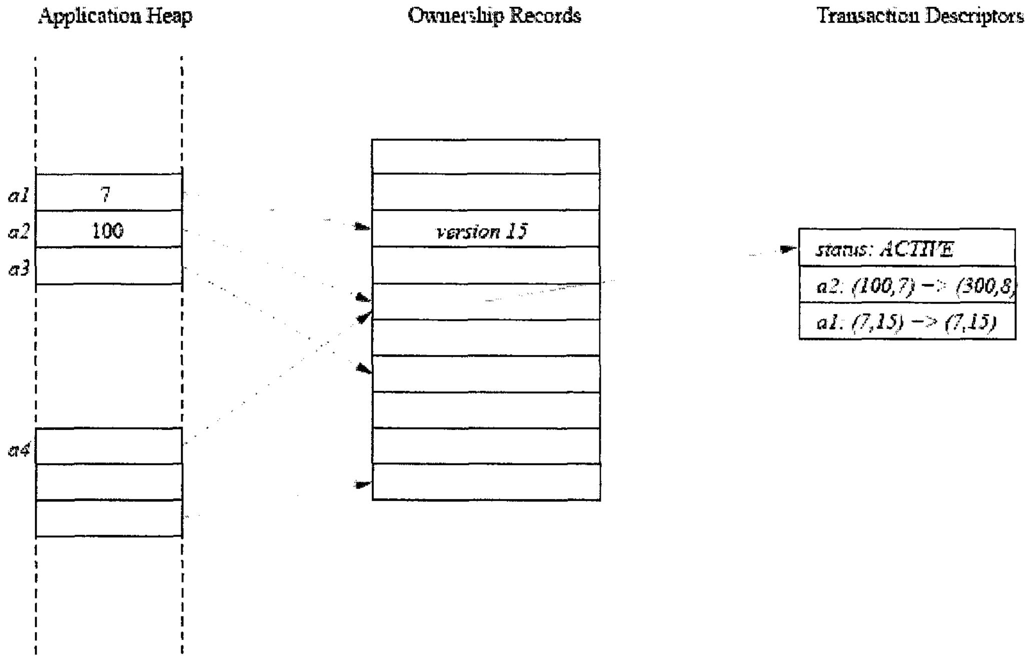

Figure 1 - STM Heap Structure Showing an Active Transaction

Each of the shared memory locations in the heap hash 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. (Herlihy et al. 2003) 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.

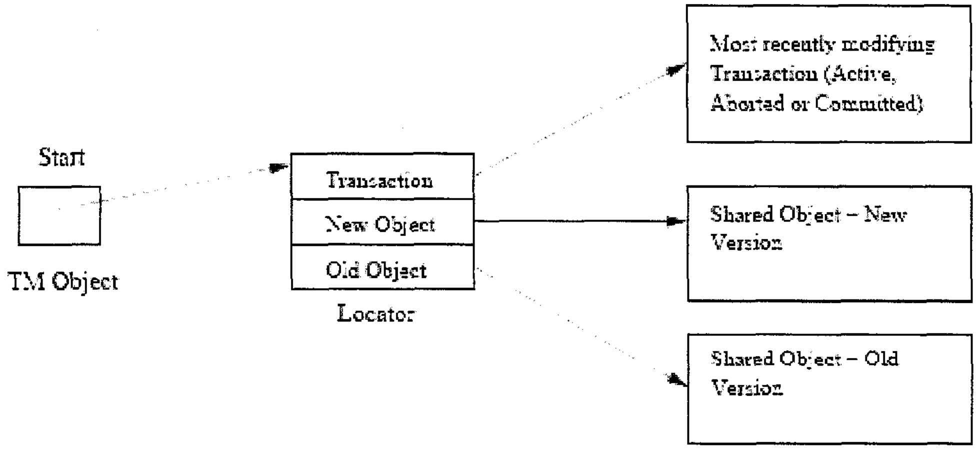
Start
TM Object
Transaction
New Object
Old Object
Locator
Most recently modifying Transaction (Active, Aborted or Committed)
Shared Object – New Version
Shared Object – Old Version

Figure 2 - Transactional Memory Object Structure

The above figure 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 increases the flexibility of the design. (Marathe and Scott 2004) 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 Locater 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. (Herlihy et al. 2003) The IntSet class uses two types of objects, nodes, which are TM objects, and List objects that are standard Java linked objects that contain 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 in to 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 opening 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". (Scherer and Scott 2005) 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 $2n + k$ 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 (Marathe and Scott 2004).

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 (Marathe and Scott 2004).

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 (Marathe and Scott 2004).

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 (Scherer and Scott 2005).

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 (Marathe and Scott 2004).

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 is learning what to do--how to map situations to actions--so as to maximize a numerical reward signal". (Kaelbling et al. 1996) 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." (Kaelbling et al. 1996)

3.0 Adaptive Contention Management

This section describes the approach used in order to achieve dynamic contention management. This includes polling times, values for base constants, and an overview of the algorithm used for ASTM.

As previously stated, the dynamic CM uses a reinforcement learning algorithm to select the proper contention manager in order to maximize the reward, which in this context is the average number of successful transactions that have been completed since the last evaluation period. For the purposes of this discussion, the reward is referred to as performance of the system. The ACM tracks the average historical performance observed for each contention manger; this stored historical performance is updated during each evaluation period. The following pseudo code describes the ACM algorithm:

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

        poll_For_Successful_Transactions();

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

                historicalPerformance[currentCM] = currentPerformance;

                found = findCmWithBetterPerformanceThan(currentPerformance);

                if ( found )
                {
                        switch_to_better_CM()

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

                        if ( randomNumber > CHANCE_TO_SWICH )
                        {
                                switchToRandomCM();

                                notifyListeners();
                        }
                }
        }

        notifyListenersOfCurrentPerformance();

        clearTransactionCounters();
}
```

Figure 4 - ACM Psuedocode

The algorithm starts by putting the current thread to sleep for a constant amount of time. (Note - the AdaptiveCM class implements Runnable). (See Section 4.0 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. 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.

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 (See Section 4.0 for design details) for future comparisons. Initially this map of historical performances is set to MAX_INTEGER for all contention managers. 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 set 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 out perform 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.

4.0 Design

This section describes not only the overall design of the adaptive contention manager and controller modules, but the interface to the original STM library as well. Refer to Appendix A for the source code of the AdaptiveCM module, the Controller module, and all modified classes from the original DSTM2 library. (Dynamic Software Transactional Memory Library 2.0)

4.1 Overview

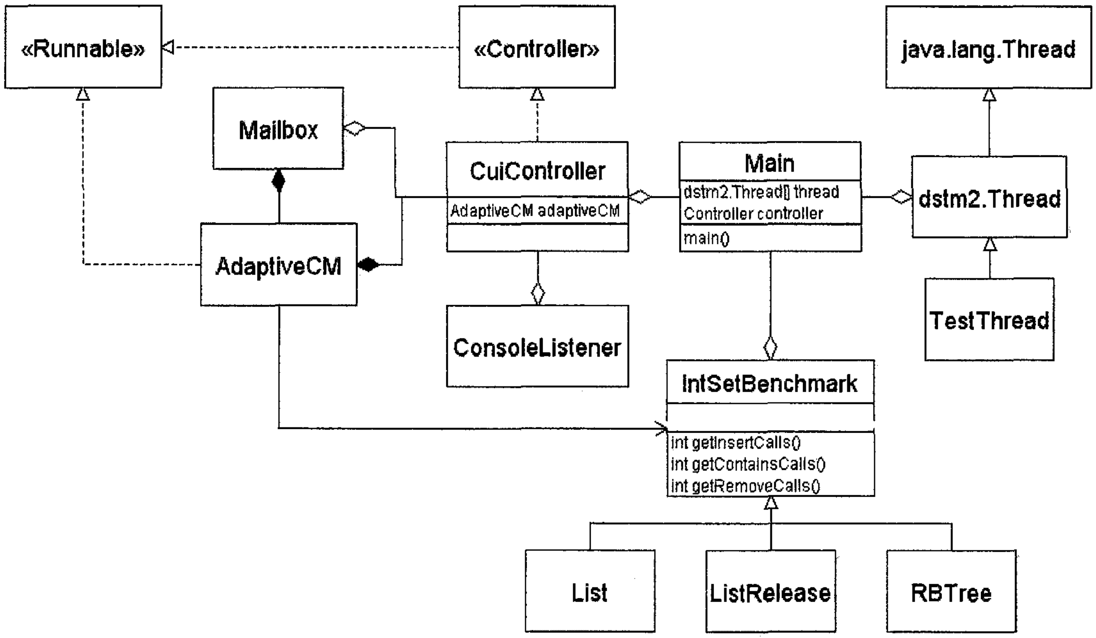

Figure 5 - Overview UML

Figure 5 shows the overview design of the system. The main thread is responsible for starting the system. Once it is initialized, it creates and starts the test threads that actually perform the transactions. The Main class is also responsible for creating the proper controller type, which in this case is the CuiController. Future development includes implementing a GUI based controller. The controller is responsible for getting the singleton AdaptiveCM object, as well as initializing it based on commands from the user. These commands are entered via the console and processed by the ConsoleListener. All communication between the controller and the adaptive contention manager is done via the Mailbox. This mailbox is created by the Controller and passed to the AdaptiveCM during creation. Once the user starts the test, the Controller informs the AdaptiveCM, who does any remaining initialization of the experiment, and then starts the process via the Main class. While running, the AdaptiveCM polls for the values of the transaction counters from the IntSetBenchmark. This benchmark is the data set being manipulated, which includes linked list, list release, and the red-black tree.

4.2 AdaptiveCM Module

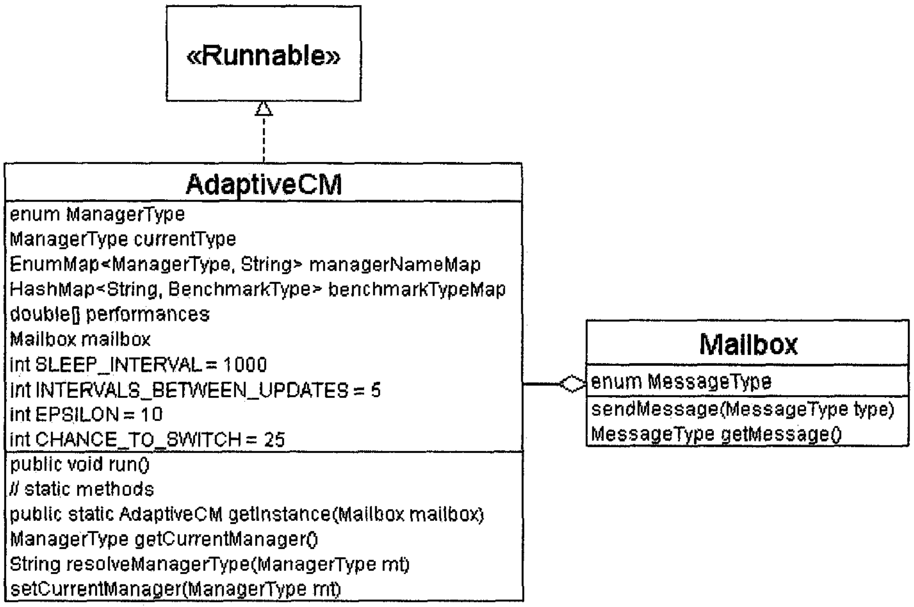
«Runnable»
AdaptiveCM
enum ManagerType
ManagerType currentType
EnumMap<ManagerType, String> managerNameMap
HashMap<String, BenchmarkType> benchmarkTypeMap
double[] performances
Mailbox mailbox
int SLEEP_INTERVAL = 1000
int INTERVALS_BETWEEN_UPDATES = 5
int EPSILON = 10
int CHANCE_TO_SWITCH = 25
public void run()
// static methods
public static AdaptiveCM getInstance(Mailbox mailbox)
ManagerType getCurrentManager()
String resolveManagerType(ManagerType mt)
setCurrentManager(ManagerType mt)
Mailbox
enum MessageType
sendMessage(MessageType type)
MessageType getMessage()

Figure 6 - Detailed Adaptive Module UML

The AdaptiveCM module is responsible for management of the various contention managers, as well communication status via its mailbox to any Controllers that are registered to receive messages. The process of message passing is non-blocking to the adaptive contention manager. This is done to minimize the overhead of the adaptive management. The AdaptiveCM class follows the singleton pattern since there is no reason to ever have more than one of these objects. The various maps and arrays contained in the AdaptiveCM class are used primarily as lookup tables or to increase performance. For example, the values for the manager enum are stored in a static array in order to speed up access times during the evaluation periods.

4.3 Controller Module

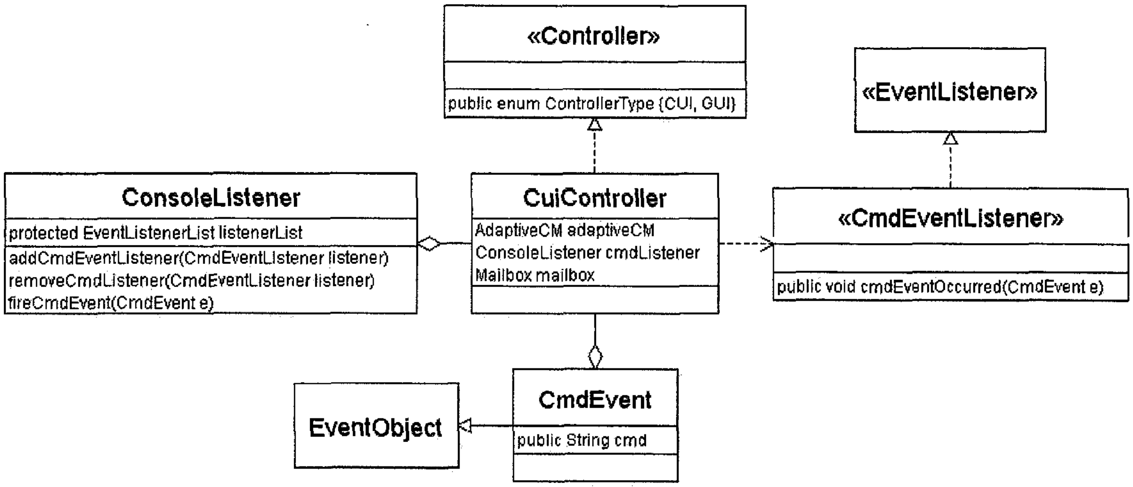
«Controller»
public enum ControllerType {CUI, GUI}
«EventListener»
ConsoleListener
protected EventListenerList listenerList
addCmdEventListener(CmdEventListener listener)
removeCmdListener(CmdEventListener listener)
fireCmdEvent(CmdEvent e)
CuiController
AdaptiveCM adaptiveCM
ConsoleListener cmdListener
Mailbox mailbox
«CmdEventListener»
public void cmdEventOccurred(CmdEvent e)
EventObject
CmdEvent
public String cmd

Figure 7 - Detailed Controller Module UML

The Controller module is responsible for getting all commands from the user and displaying updates from the AdaptiveCM via the Mailbox. The CuiController spawns a new thread to handle all input from the user. This new thread, the ConsoleListener, will fire command events to the controller, who in turn processes the command and updates any public fields in the AdaptiveCM as necessary. For the CuiController, a basic text help menu is provided in order to give the user the basic available commands to be used to setup and control the system. When requesting messages from the mailbox, the Controller thread will block until it is notified by the mailbox to wake and process the new message stored in the mailbox.

5.0 Software Tools Used

This experiment was done using Java 1.5.0.09. Although the original DSTM2 library was written for an older version of java, it was still fully functional under this newer version. The main STM library used was the DSTM2 library developed by Maurice Herlihy, et al. This library was slightly modified to support the addition of the AdaptiveCM module. These modifications included breaking up the algorithm of the Main class in order to support a staged initialization, altering the counters of the IntSetBenchmark class to support the new polling requirements, and updating the dstm2.Thread class to support dynamically setting the contention manager during runtime.

6.0 Experimental Results

This section details the experimental results, which included first determining the performance impact of the polling modifications to the DSTM2 library, and secondly comparing the performance of the ASTM to these baselines.

6.1 Impact of Polling Modifications

The initial task of the experiment was to determine the impact, if any, 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 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; as a result trials were repeated using one, ten, thirty, and fifty threads. The number of threads chosen was based on related experiments (Herlihy et al. 2003) 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 graphs show the comparison of baseline data to static ASTM performance. The first graph in each section is a composite graph showing all contention managers. For clarity, this graph is then followed by a specific comparison graph for each contention manager. As can be seen in each graph, the modifications to the DSTM2 library had no significant effect on overall performance.

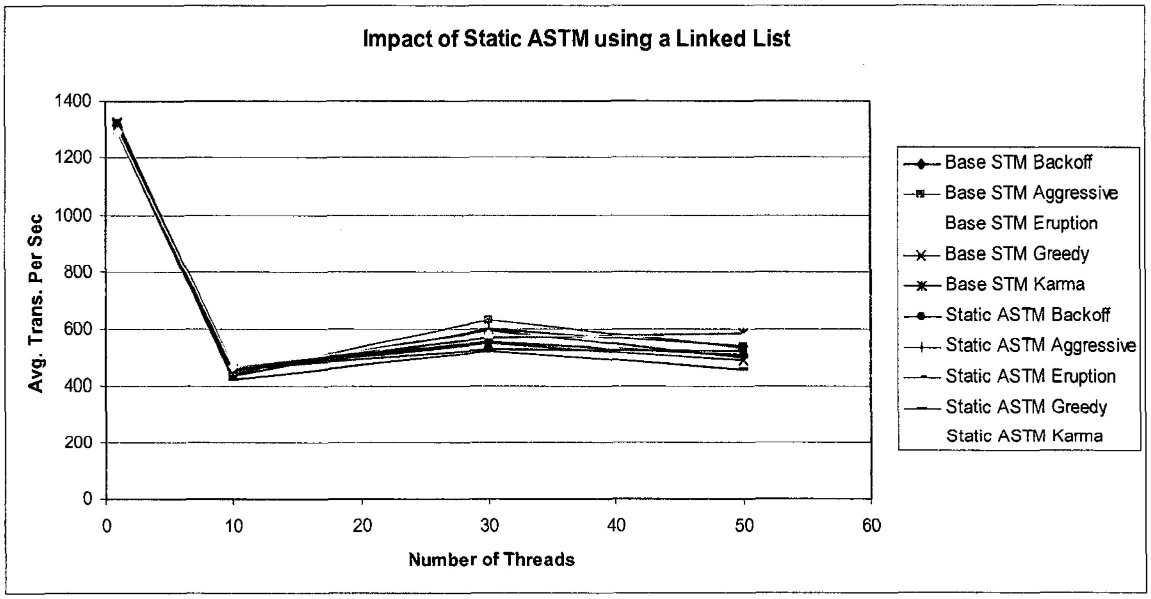

Figure 8 - Baseline vs. Static ASTM Using a Linked List

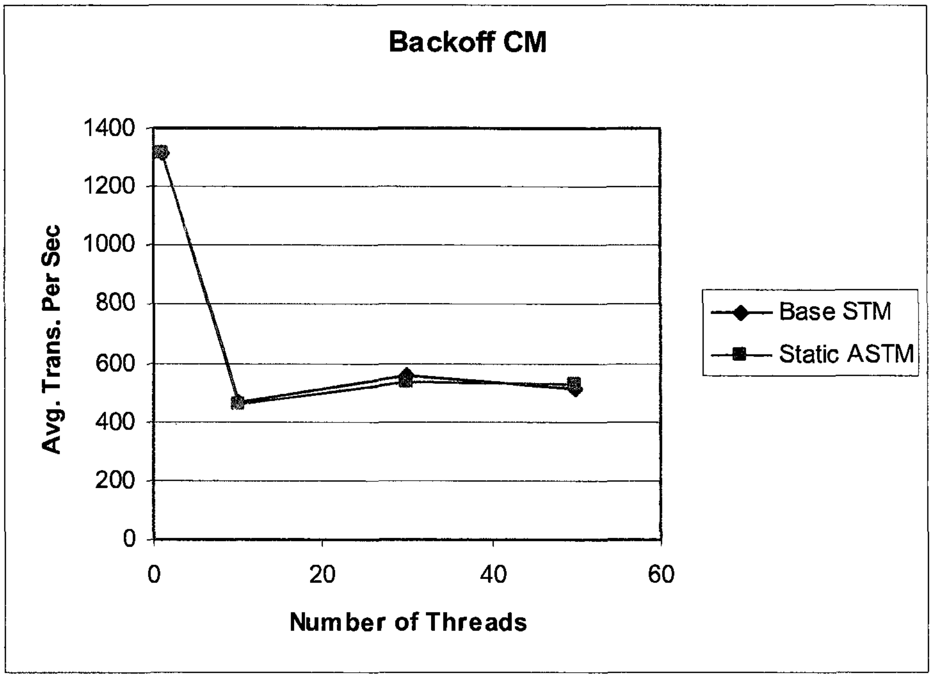

Figure 9 - Backoff CM

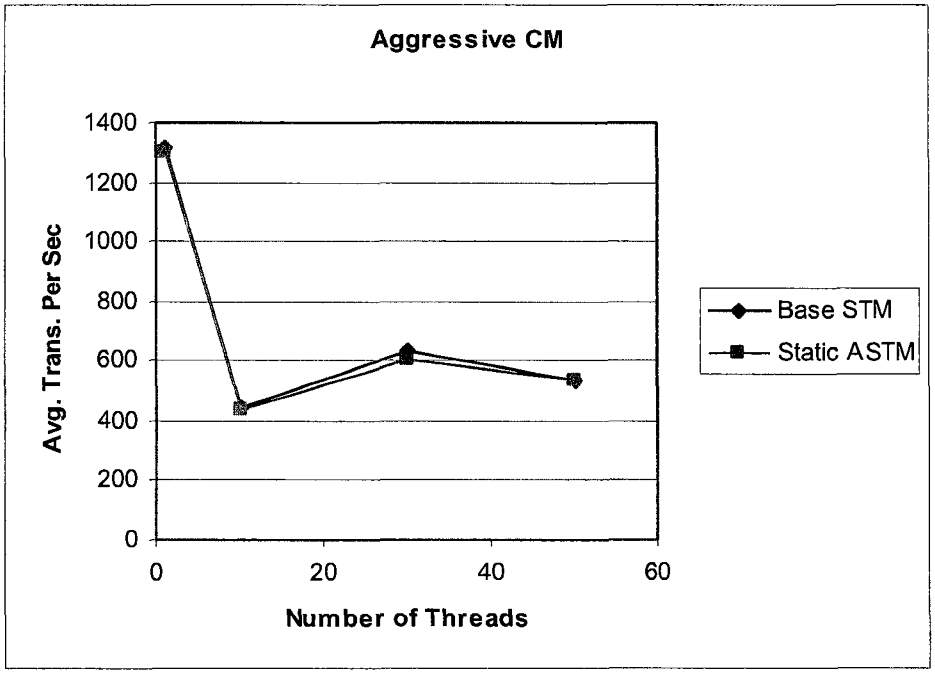

Figure 10 - Aggressive CM

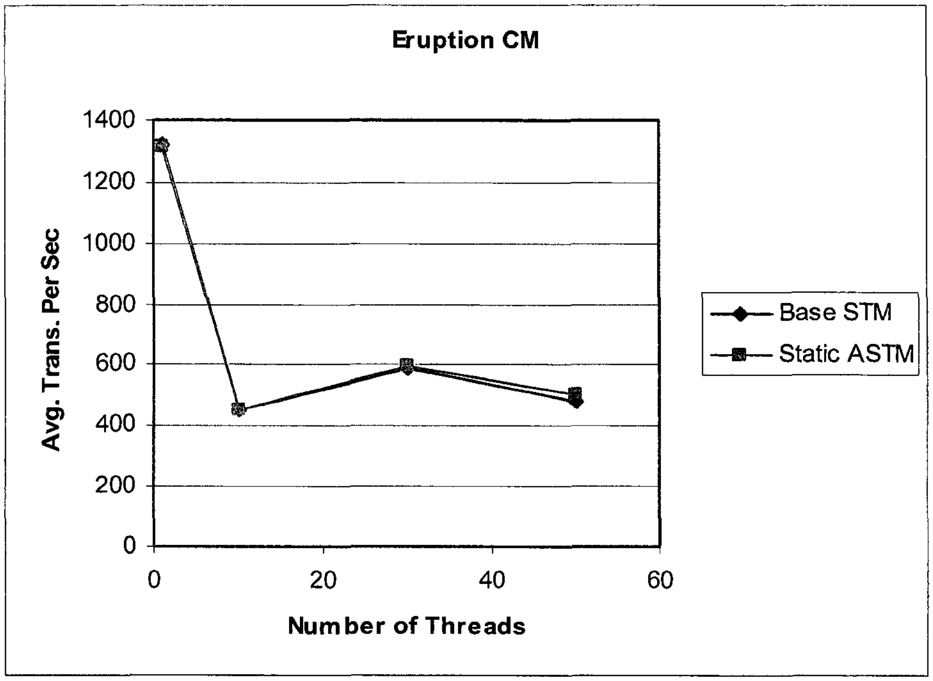

Figure 11 - Eruption CM

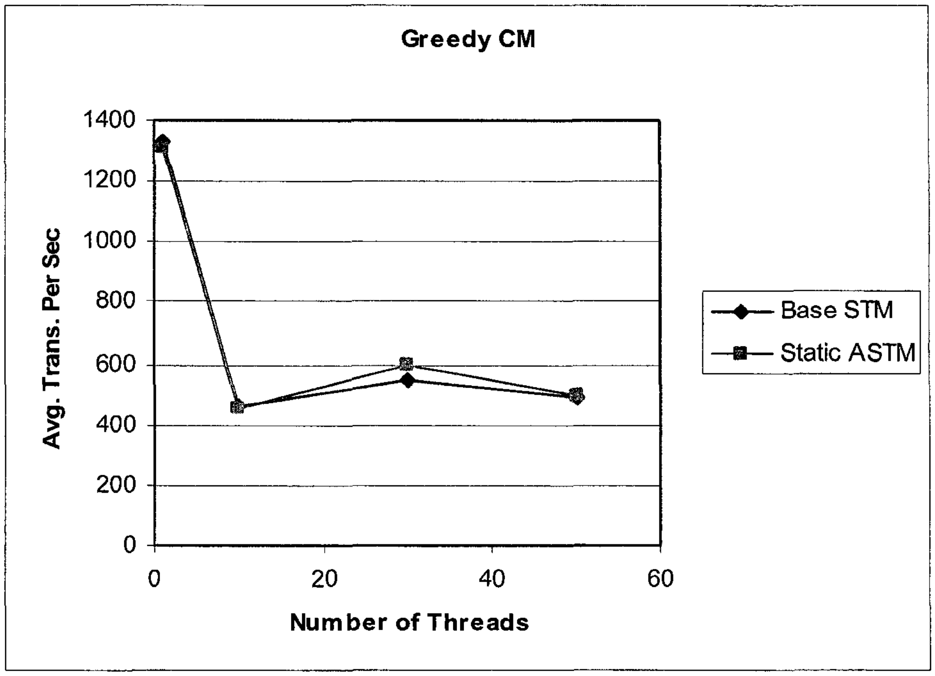

Figure 12 - Greedy CM

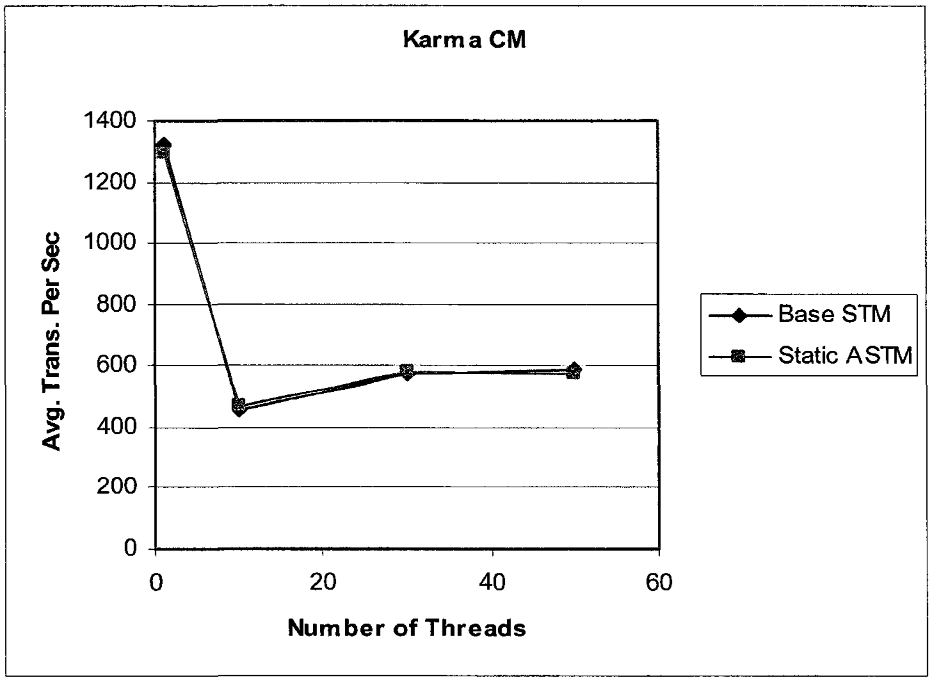

Figure 13 - Karma CM

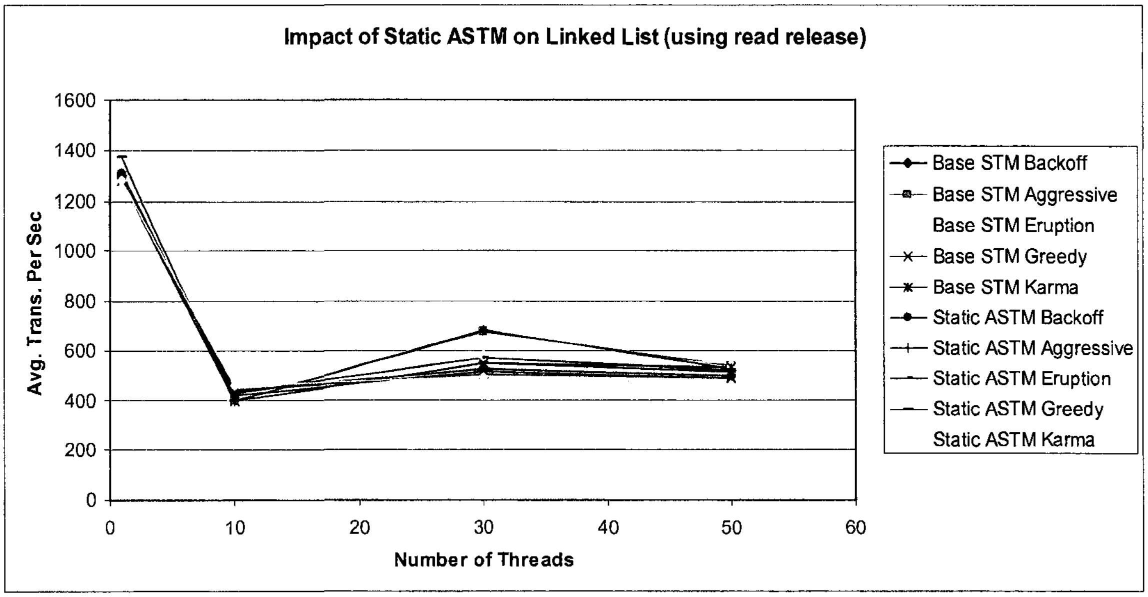

Figure 14 - Baseline vs. Static ASTM Using List Release

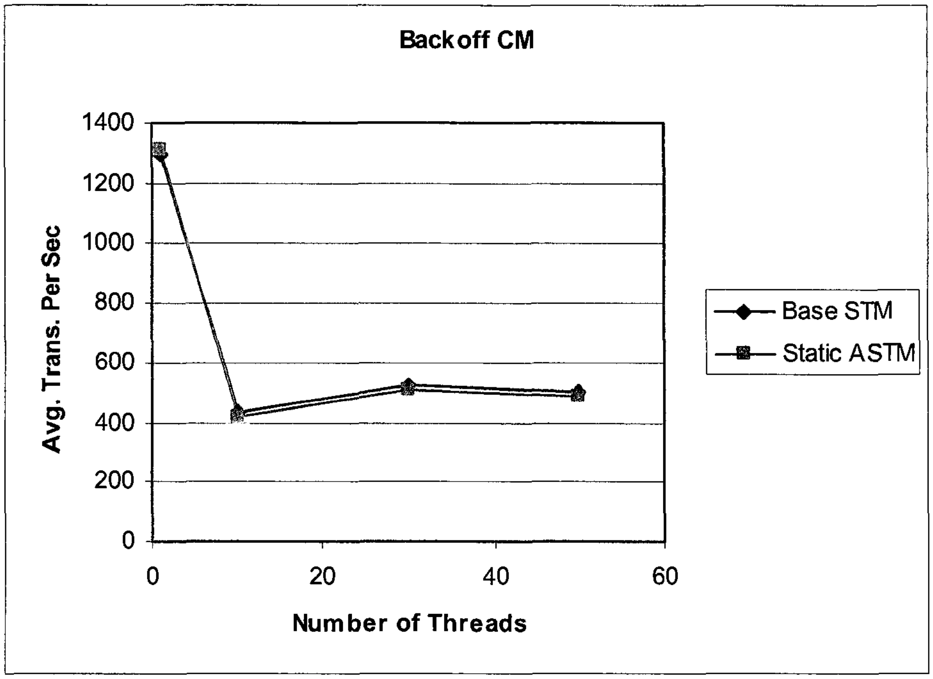

Figure 15 - Backoff CM

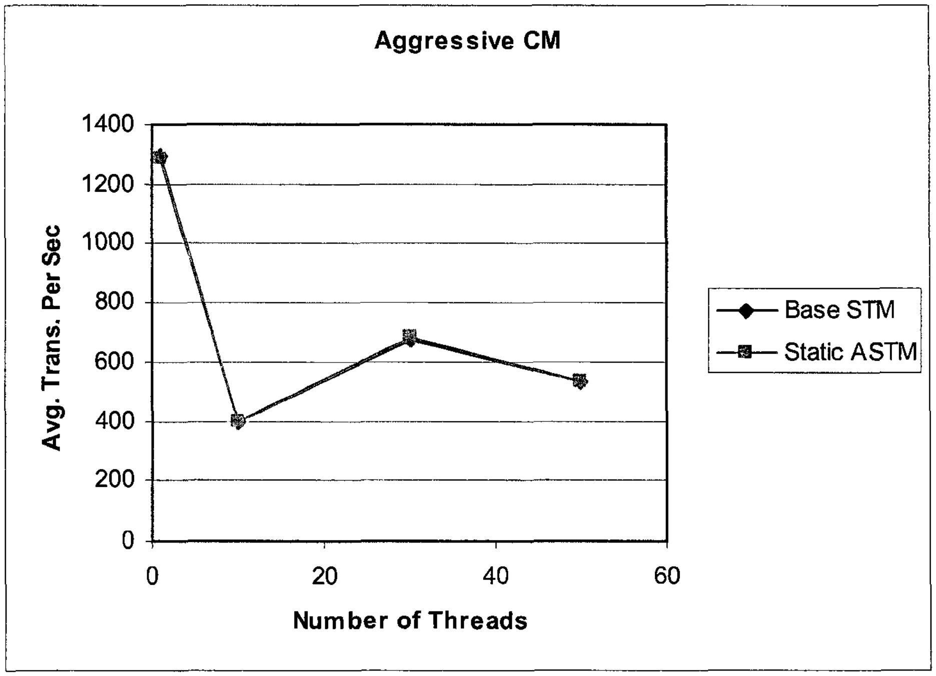

Figure 16 - Aggressive CM

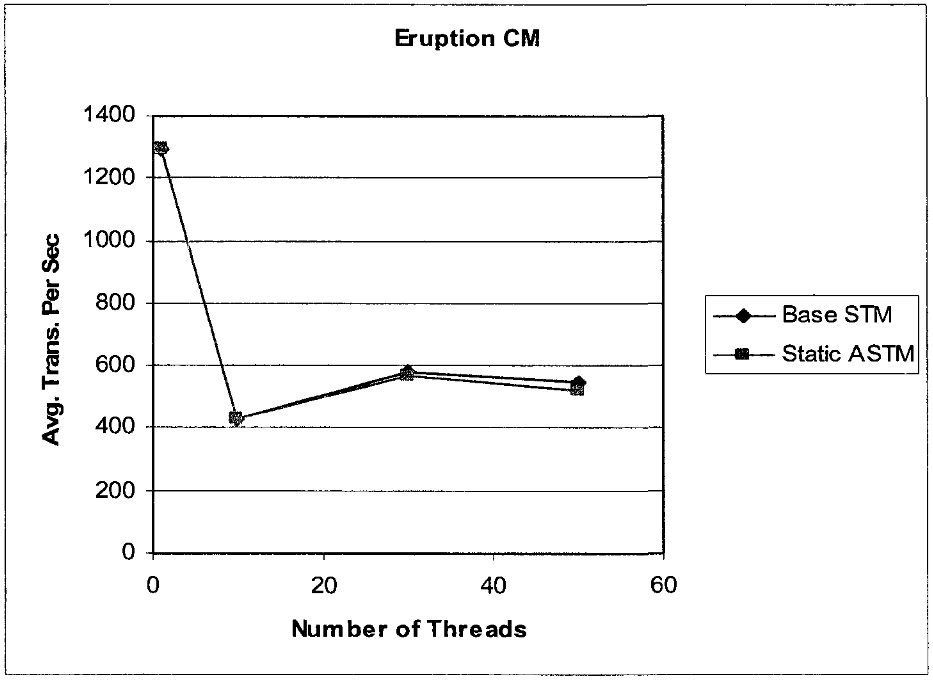

Figure 17 - Eruption CM

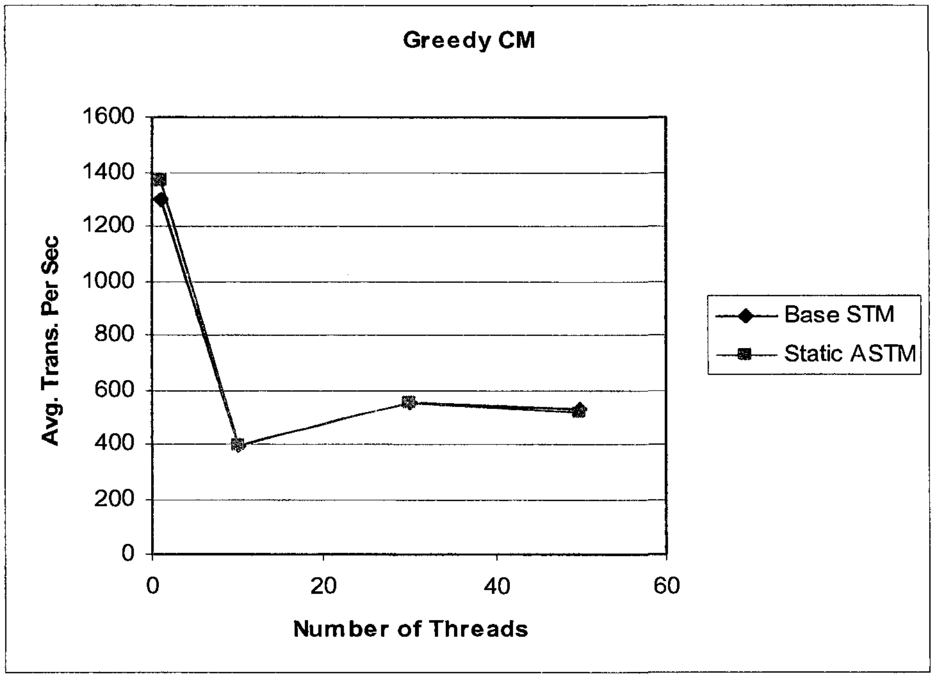

Figure 18 - Greedy CM

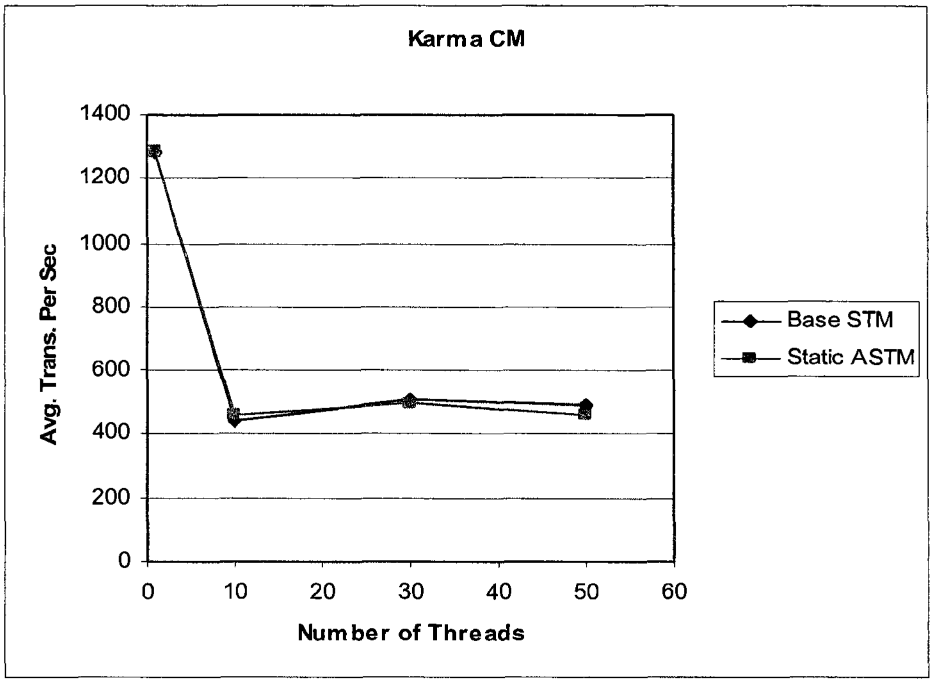

Figure 19 - Karma CM

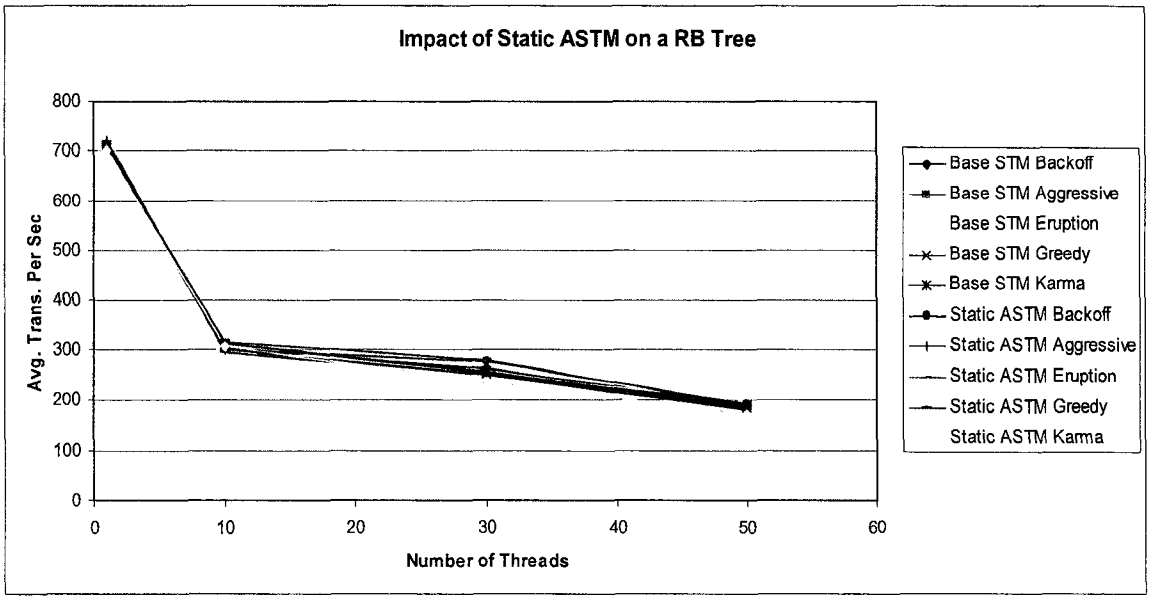

Figure 20 - Baseline vs. Static ASTM Using a Red-Black Tree

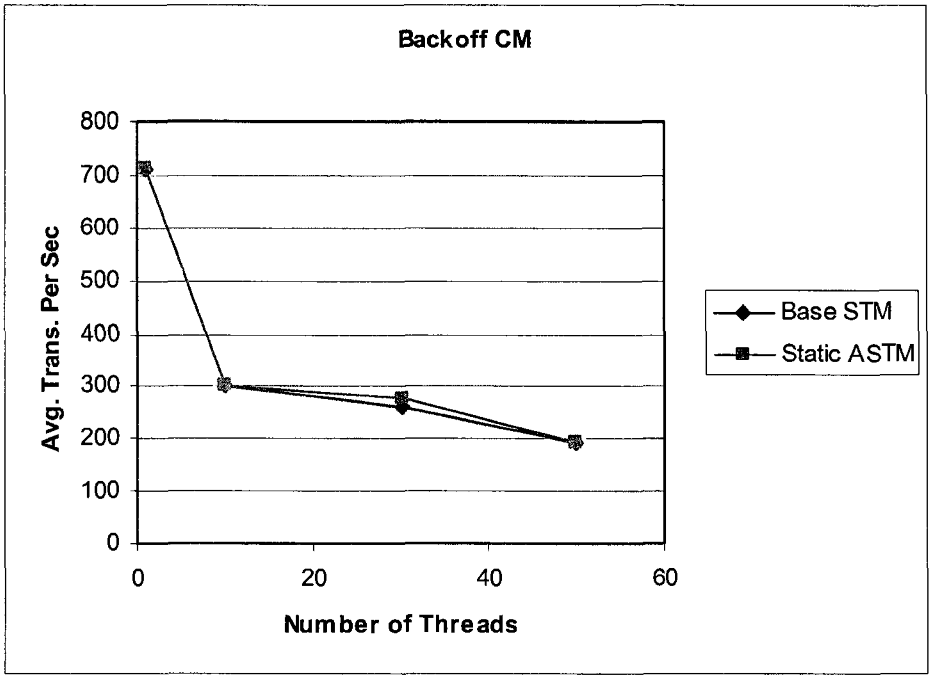

Figure 21 - Backoff CM

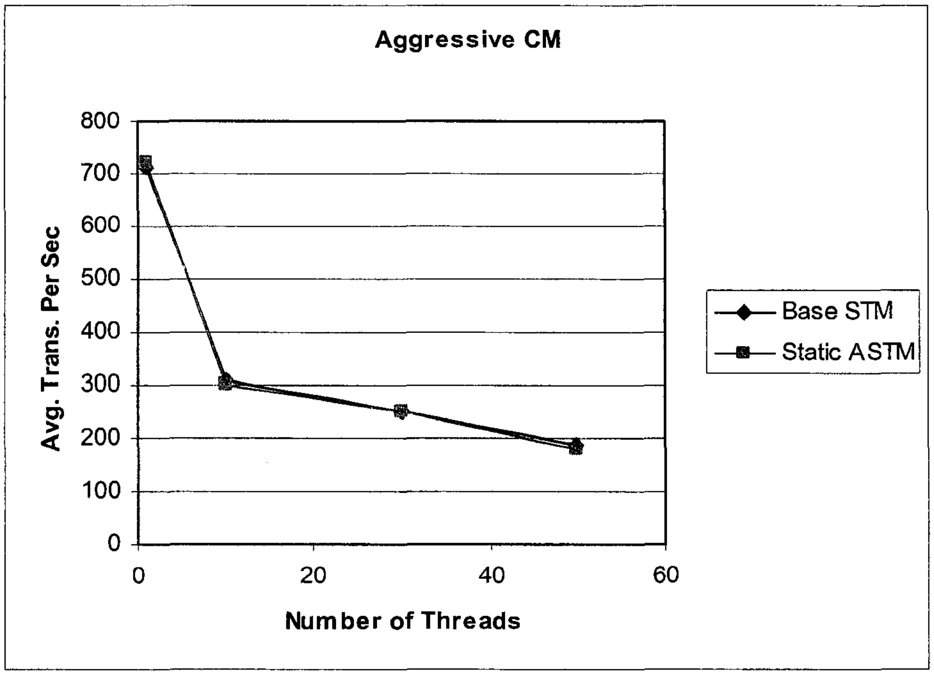

Figure 22 - Aggressive CM

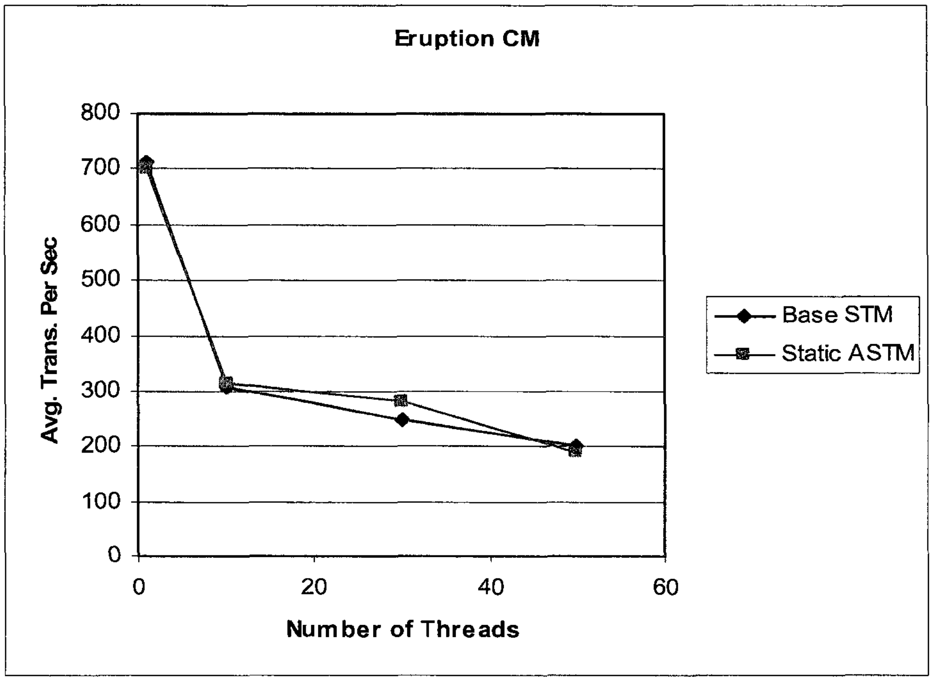

Figure 23 - Eruption CM

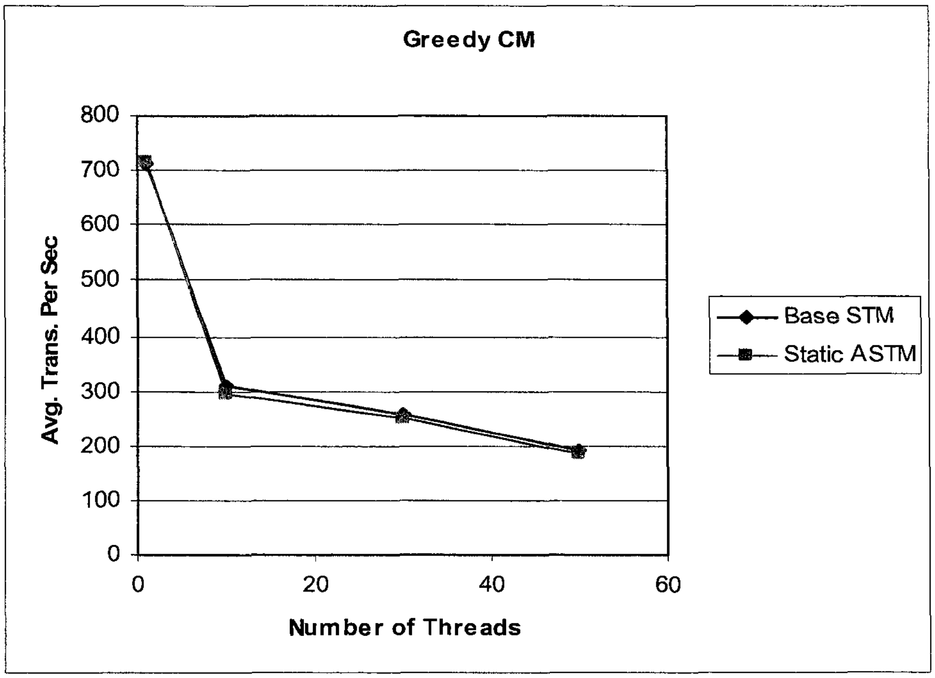

Figure 24 - Greedy CM

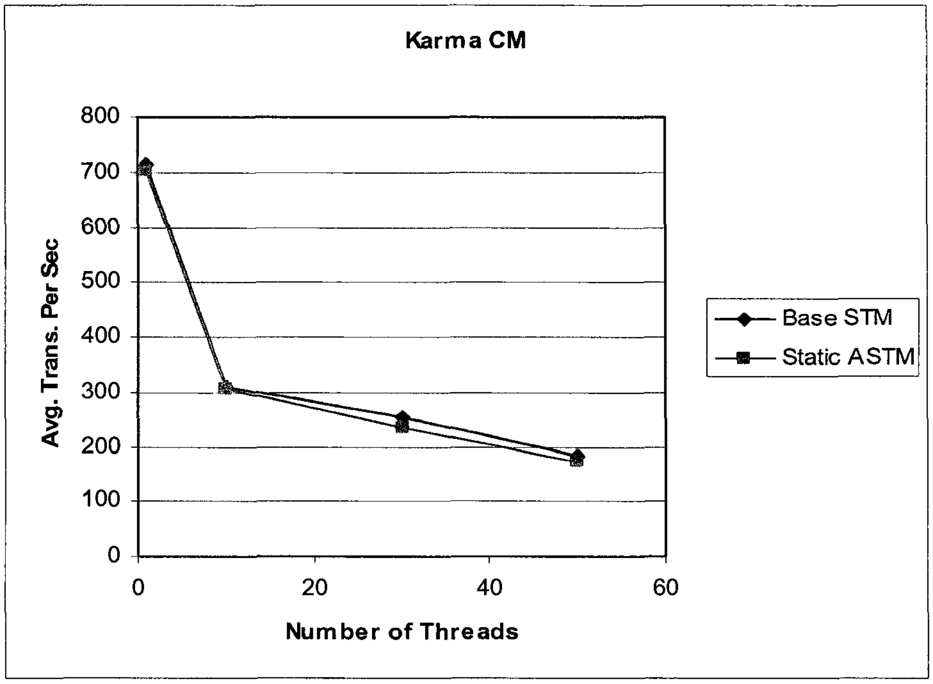

Figure 25 - Karma CM

6.2 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. 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.

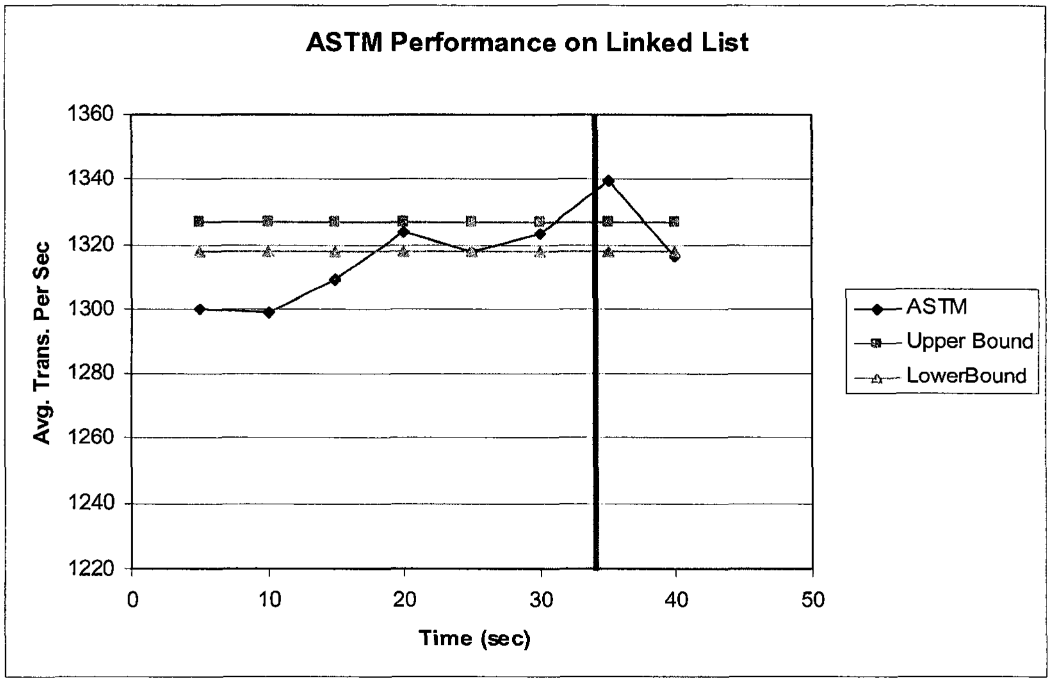

Figure 26 - ASTM Performance Using 1 Thread (Steady State CM = Karma)

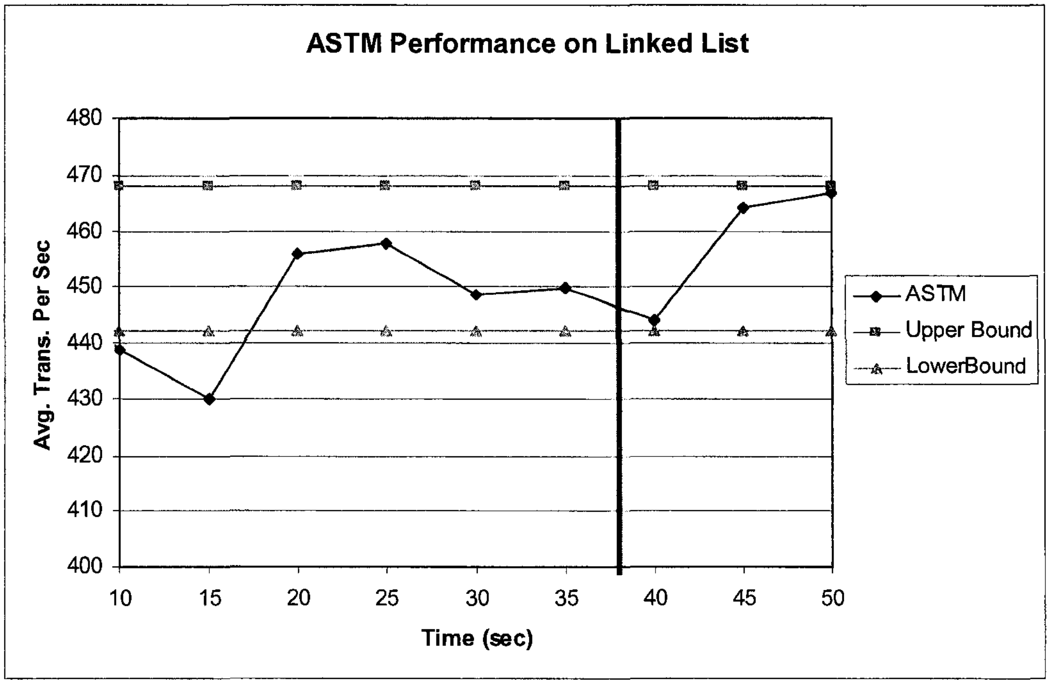

Figure 27 - ASTM Performance Using 10 Threads (Steady State CM = Eruption)

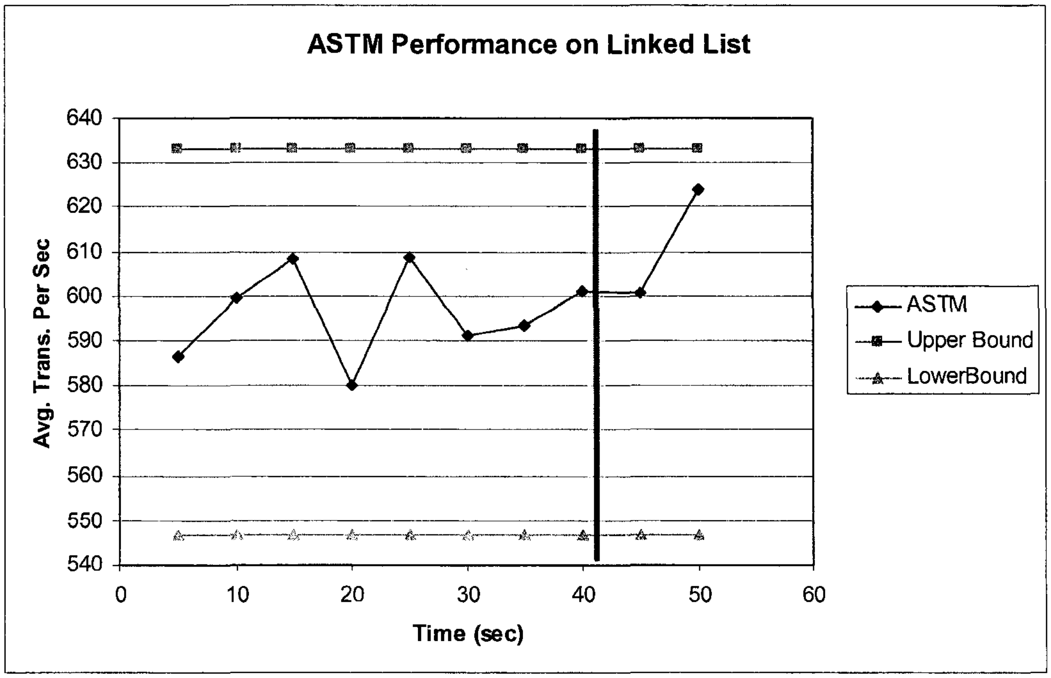

Figure 28 - ASTM Performance Using 30 Threads (Steady State CM = Aggressive)

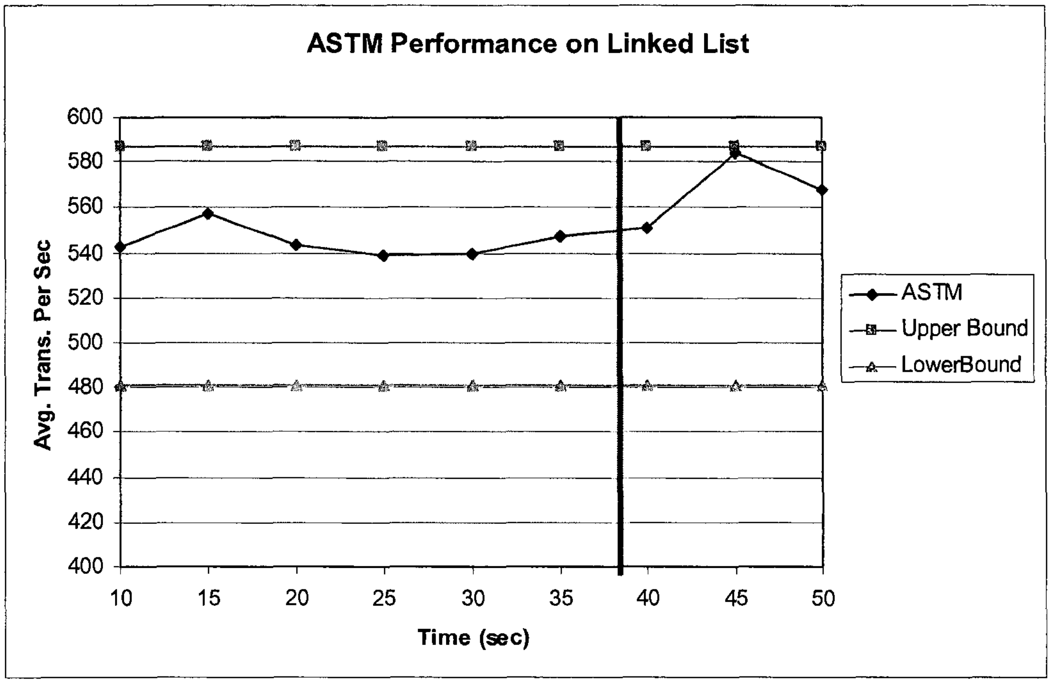

Figure 29 - ASTM Performance Using 50 Threads (Steady State CM = Aggressive)

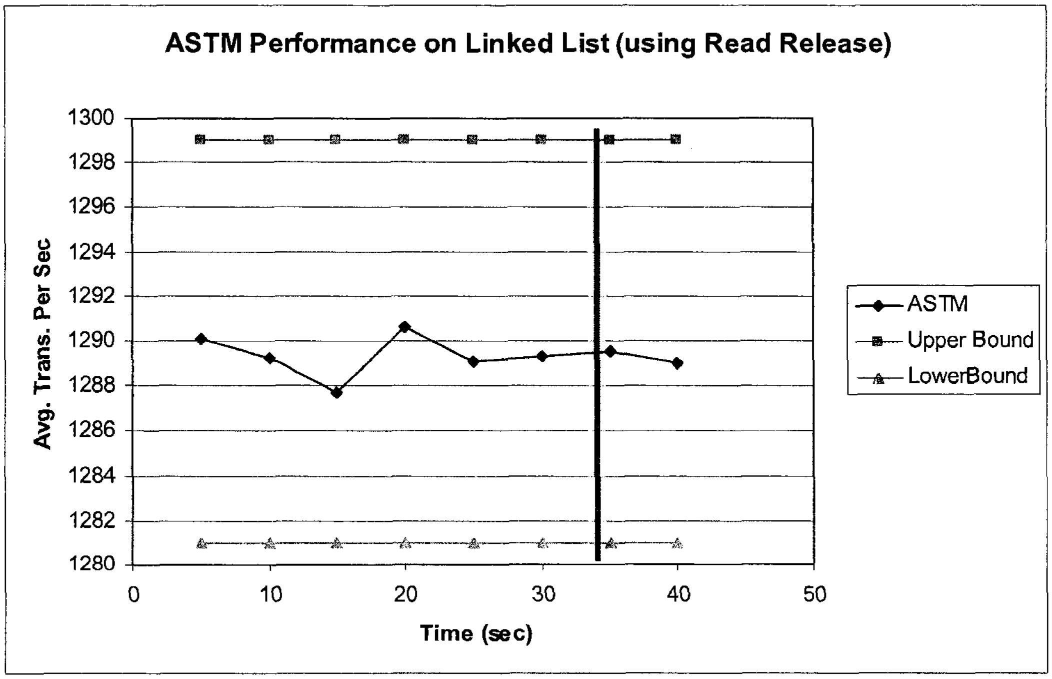

Figure 30 - ASTM Performance Using 1 Thread (Steady State CM = Aggressive)

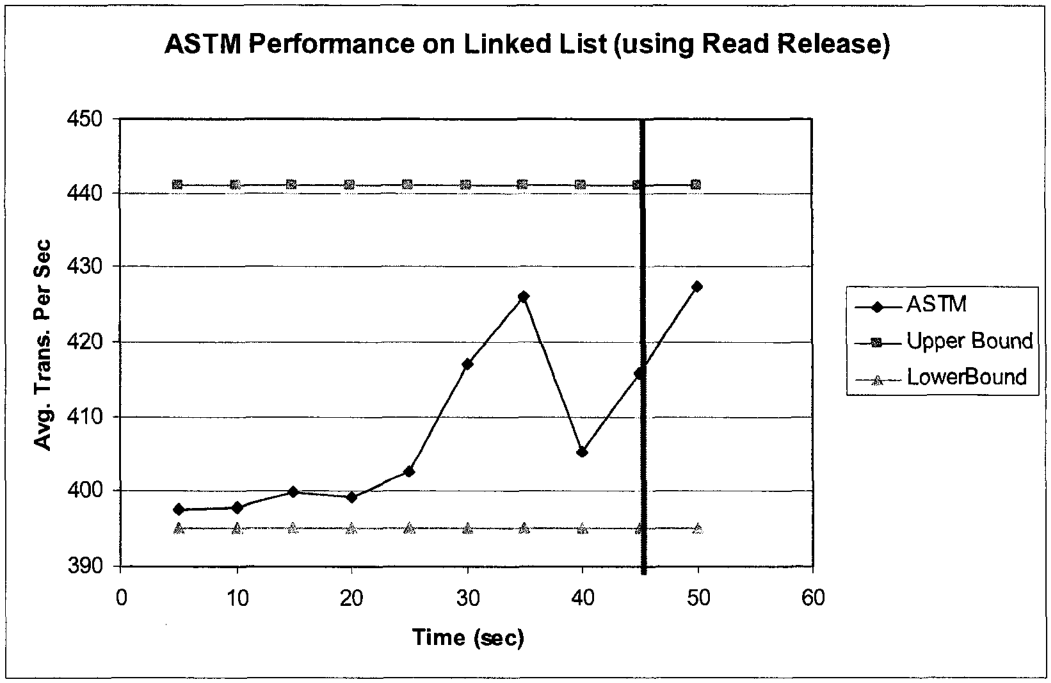

Figure 31 - ASTM Performance Using 10 Threads (Steady State CM = Karma)

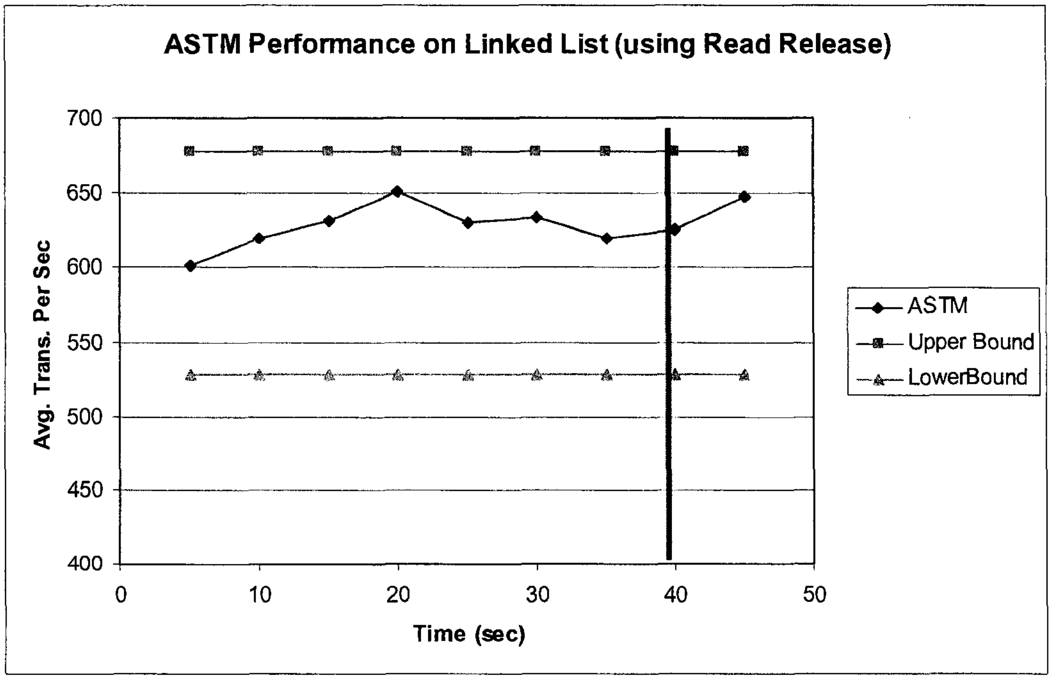

Figure 32 - ASTM Performance Using 30 Threads (Steady State CM = Aggressive)

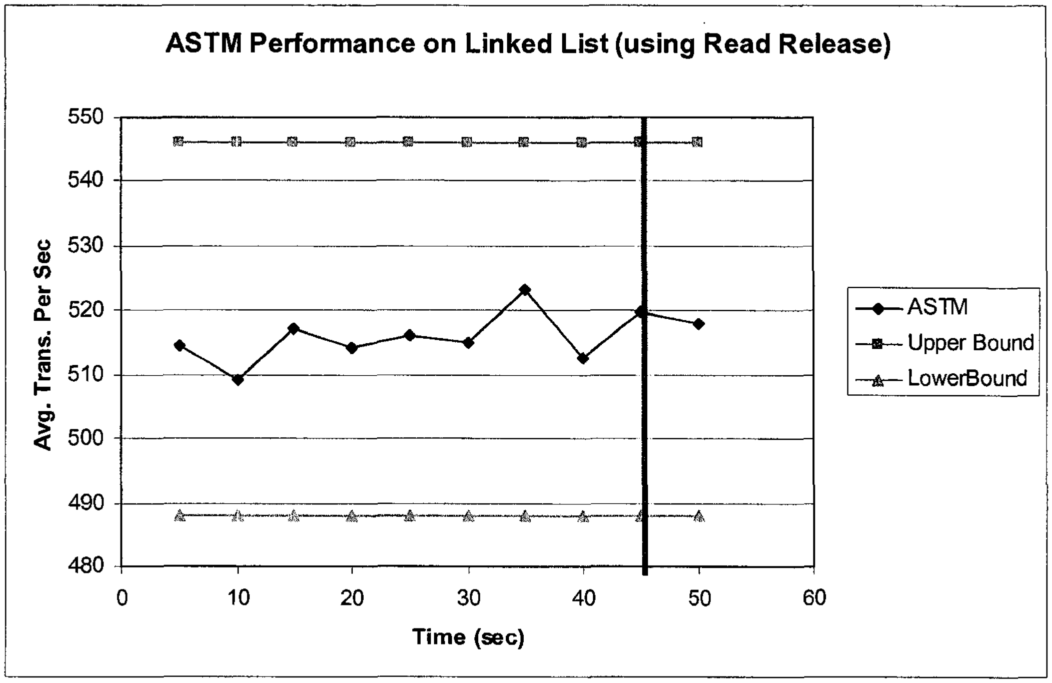

Figure 33 - ASTM Performance Using 50 Threads (Steady State CM = Eruption)

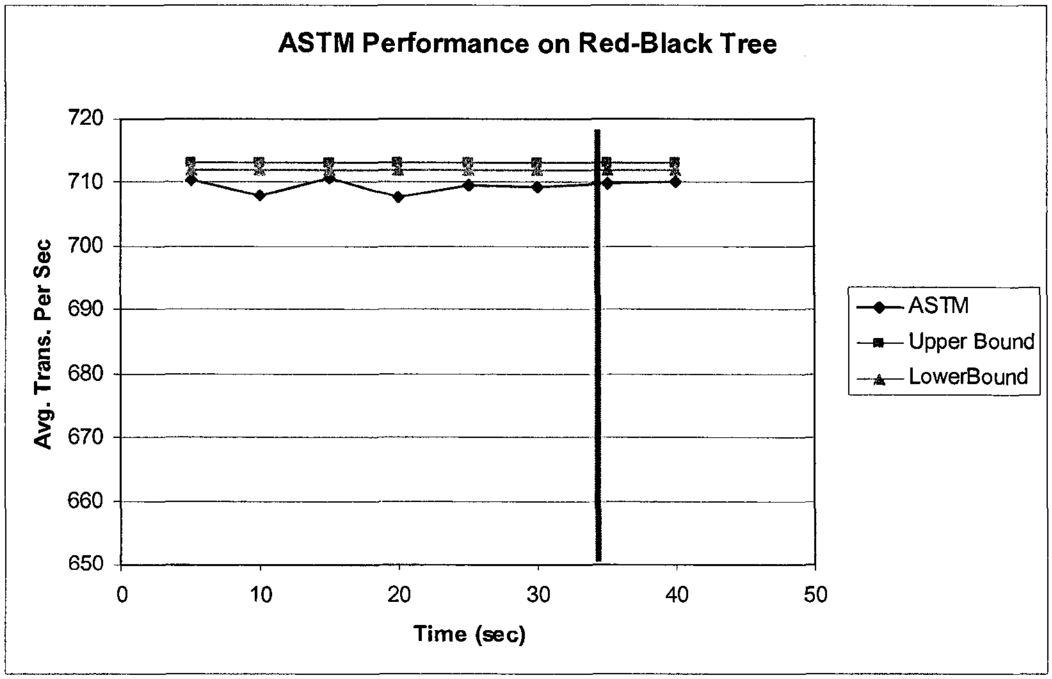

Figure 34 - ASTM Performance Using 1 Thread (Steady State CM = Greedy)

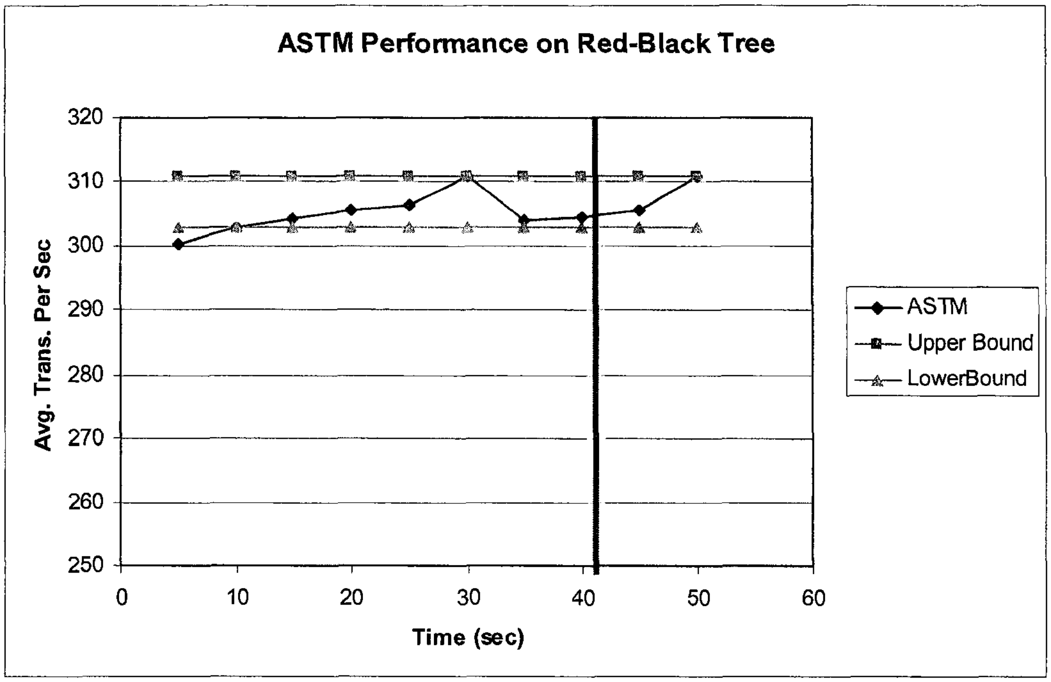

Figure 35 - ASTM Performance Using 10 Threads (Steady State CM = Backoff)

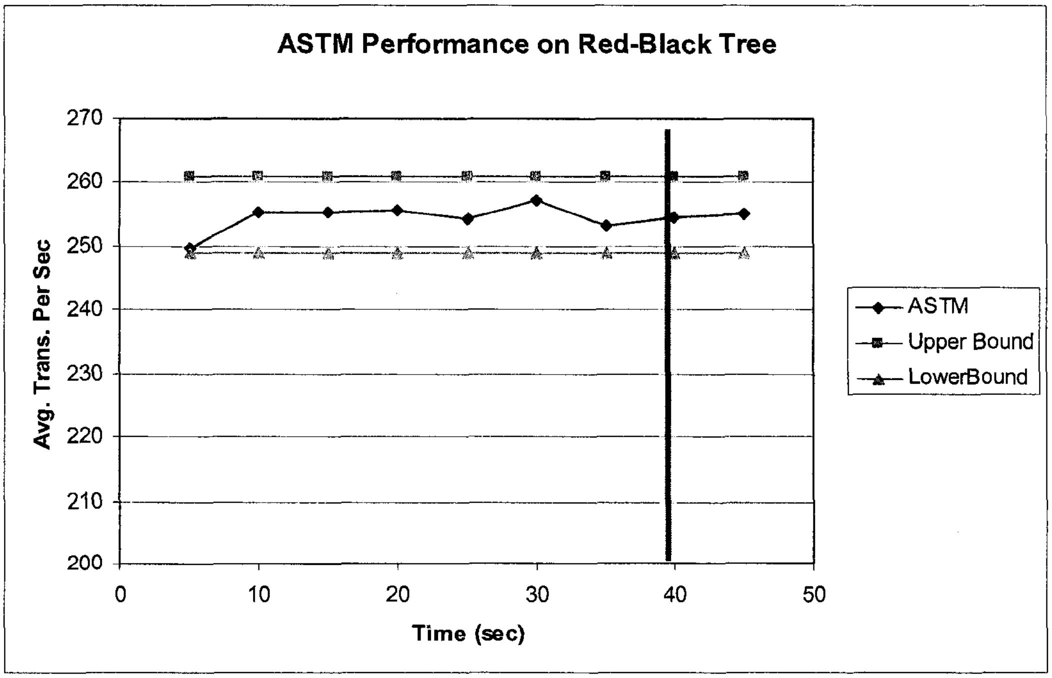

Figure 36 - ASTM Performance Using 30 Threads (Steady State CM = Greedy)

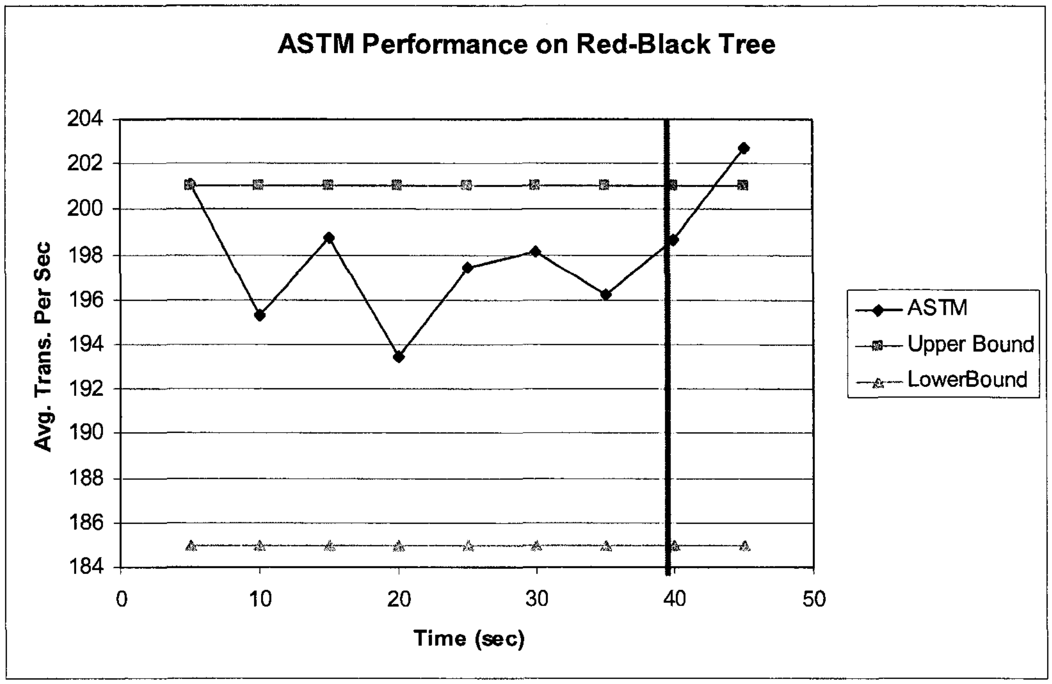

Figure 37 - ASTM Performance Using 50 Threads (Steady State CM = Eruption)

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.

7.0 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 can not be statically chosen correctly, the ASTM yields a higher average performance over time. Furthermore, in a real production type environment, not only the size of data structure would 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.

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.

8.0 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.

Improvement of the learning algorithm to become 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 were highly specialized, but far outperformed the common contention managers, the ASTM would greatly benefit by being able to utilize these specialized contention managers 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 seriously 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.

APPENDIX A. Source Code

```
/**
 * This package provides an adaptive contention manager for use with the
 * DSTM2 library.  Minor changes were made to the dstm2 library in order to
 * support this package.
 */
package adaptiveCM;


import java.util.*;
import dstm2.*;
import dstm2.benchmark.*;
import static dstm2.Defaults.*;
import adaptiveCM.Mailbox.MessageType;


/**
 * @author Joel C. Frank
 *
 */
public class AdaptiveCM implements Runnable
{
        public enum ManagerType
                {AGGRESIVE, BACKOFF, ERUPTION, GREEDY, KARMA}

        public enum BenchmarkType
                {LIST, LIST_RELEASE, LIST_SNAP, RB_TREE, SKIP_LIST}

        /**
         * This current contention manager type in use
         */
        public static ManagerType currentType;

        /**
         * The class name of the adapter to use
         */
        public static String adapterClassName = ADAPTER;

        /**
         * The class name of the benchmark to use
         */
        public static String benchmarkClassName = BENCHMARK;

        /**
         * Map that relates ManagerTypes to their class names
         */
        private static EnumMap<ManagerType, String> managerNameMap;

        /**
         * Map that relates the benchmark class names to the BenchmarkType enum
 */
private static HashMap<String, BenchmarkType> benchmarkTypeMap;

/**
 * Array that stores the current expected or seen performance indexed by
 * ManagerType.  Performance is measured as the number of commits / sec.
 *
 * This map is initially set to optimistic values (MAX_INT) for each
 * manager type.  This causes this object to assume "the grass is always
 * greener" unless it knows otherwise.  This is consistant with the
 * Reinforcement Learning scheme implemented for this class.
 */
private static double[] performances;

/**
 * Reference to this singleton object
 */
private static AdaptiveCM acm = null;

/**
 * The benchmark currently being tested
 */
private static Benchmark benchmark = null;

/**
 * The current performance seen so far on the current manager type.
 * Updated based on reports from the benchmark's test thread
 */
private static double currentPerformance = 0;

/**
 * The number of intervals that have elapsed since the performance was
 * last evaluated
 */
private static int intervalsSinceLastEval = 0;

/**
 * The system time in millis of the last time the stats were reset
 */
private static long timeOfLastReset = 0;

/**
 * The elapsed time between updates
 */
private static double elapsedTime = 0;

/**
 * This mailbox is used to deliver messages to all listeners
 */
private static Mailbox mailbox;

/**
 * Flag used to know when to clean up and exit this thread
 */
public boolean finished = false;

/**
 * Flag that causes the adaptiveCM adapt to performance.  More specifically,
 * setting this flag to true will cause the performance to be evaluated
 * every iteration, which could possibly lead to an adaptive switch
 * to a more efficient contention manager for the current environment
 */
public boolean runAdaptively = true;

/**
 * static array of manager types pre-stored to help performance
 */
private final static ManagerType[] MANAGER_TYPES = ManagerType.values();

/**
 * The interval to sleep between posting call stat updates and potentially
 * checking performance
 */
private final static int SLEEP_INTERVAL = 1000;

/**
 * The number of intervals to wait between checking performance. This is
 * also when the contention manager may be changed
 */
private final static int INTERVALS_BETWEEN_UPDATES = 5;

/**
 * The minimum performance difference required between the current and
 * potential contention managers.  This epsilon value must be exceeded
 * in order to consider another contention manager to have better
 * performance.
 */
private final static int EPSILON = 10;

/**
 * The percentage chance to switch to a random contention manager when
 * evaluating performance and no contention manager is found that the
 * adaptiveCM 'thinks' will be more efficient.
 */
private final static int CHANCE_TO_SWITCH = 25;




/**
 * Proper method for getting/creating a singleton AdaptiveCM object.
 *
 * @return AdaptiveCM singleton object
 */
```

```
public static AdaptiveCM getInstance(Mailbox mailbox)
{
        if(acm == null)
                acm = new AdaptiveCM();

        AdaptiveCM.mailbox = mailbox;

        return acm;
}


/**
 * Private constructor for the singleton pattern.
 */
private AdaptiveCM()
{
        // initialize the static members
        AdaptiveCM.currentType = ManagerType.BACKOFF;

        AdaptiveCM.managerNameMap =
                new EnumMap<ManagerType, String>(ManagerType.class);

        AdaptiveCM.managerNameMap.put(
                        ManagerType.AGGRESIVE, "dstm2.manager.AggressiveManager");
        AdaptiveCM.managerNameMap.put(
                        ManagerType.BACKOFF, "dstm2.manager.BackoffManager");
        AdaptiveCM.managerNameMap.put(
                        ManagerType.ERUPTION, "dstm2.manager.EruptionManager");
        AdaptiveCM.managerNameMap.put(
                        ManagerType.GREEDY, "dstm2.manager.GreedyManager");
        AdaptiveCM.managerNameMap.put(
                        ManagerType.KARMA, "dstm2.manager.KarmaManager");


        AdaptiveCM.benchmarkTypeMap = new HashMap<String, BenchmarkType>();

        AdaptiveCM.benchmarkTypeMap.put(
                        "dstm2.benchmark.List", BenchmarkType.LIST);
        AdaptiveCM.benchmarkTypeMap.put(
                        "dstm2.benchmark.ListRelease", BenchmarkType.LIST_RELEASE);
        AdaptiveCM.benchmarkTypeMap.put(
                        "dstm2.benchmark.ListSnap", BenchmarkType.LIST_SNAP);
        AdaptiveCM.benchmarkTypeMap.put(
                        "dstm2.benchmark.RBTree", BenchmarkType.RB_TREE);
        AdaptiveCM.benchmarkTypeMap.put(
                        "dstm2.benchmark.SkipList", BenchmarkType.SKIP_LIST);


        AdaptiveCM.performances = new double[MANAGER_TYPES.length];

        for(int index = 0; index < MANAGER_TYPES.length; ++index)
                AdaptiveCM.performances[index] = Double.MAX_VALUE;
}
```

```
/**
 * This method runs the adaptive contention manager.  It creates/manages all
 * children threads, and cleans up upon termination of the experiment.
 */
public void run()
{
   try
   {
         // set the contention manager
         Main.managerClassName = AdaptiveCM.resolveManagerType(currentType);

         // set the adapter class
         Main.adapterClassName = AdaptiveCM.adapterClassName;

         // set the benchmark class
         Main.benchmarkClassName = AdaptiveCM.benchmarkClassName;

         // initialize all sub components now that they are set
         Main.initialize();


         AdaptiveCM.benchmark = Main.getBenchmark();
         BenchmarkType bt = AdaptiveCM.benchmarkTypeMap.get(
                           benchmark.getClass().getName());

         if(bt == null)
                  throw new IllegalArgumentException("Unknown Benchmark");

         AdaptiveCM.timeOfLastReset = System.currentTimeMillis();
         Main.startExperiment();

         int insertCalls = 0;
         int removeCalls = 0;
         int containsCalls = 0;

         while(!finished)
         {
                  java.lang.Thread.sleep(SLEEP_INTERVAL);

                  insertCalls = benchmark.getInsertCalls();
                  removeCalls = benchmark.getRemoveCalls();
                  containsCalls = benchmark.getContainsCalls();

                  if(AdaptiveCM.intervalsSinceLastEval >=
                           AdaptiveCM.INTERVALS_BETWEEN_UPDATES)
                  {
                           // get the elapsed seconds
                           AdaptiveCM.elapsedTime = (double)(
                                             (System.currentTimeMillis() -
                                             AdaptiveCM.timeOfLastReset) / 1000.0);
```

```
            AdaptiveCM.currentPerformance = (insertCalls +  removeCalls)
                / AdaptiveCM.elapsedTime;

            AdaptiveCM.intervalsSinceLastEval = 0;

            if(this.runAdaptively)
                evaluatePerformance();
            else
            {
                // just send the current performance update
                mailbox.clear();
                //mailbox.type = MessageType.STATIC_PERF;
                mailbox.currentPerformance =
                    AdaptiveCM.currentPerformance;
                mailbox.sendMessage(MessageType.STATIC_PERF);

            }
        }
        else
            ++AdaptiveCM.intervalsSinceLastEval;

        // clear the stats
        AdaptiveCM.benchmark.resetCallCounters();
        AdaptiveCM.timeOfLastReset = System.currentTimeMillis();


        // create the new update event
        mailbox.clear();
        //event.type = MessageType.CALL_STATS;
        mailbox.insertCalls = insertCalls;
        mailbox.removeCalls = removeCalls;
        mailbox.containsCalls = containsCalls;
        mailbox.sendMessage(MessageType.CALL_STATS);
    }

    Main.stopExperiment();
  }
  catch (Exception e)
  {
    e.printStackTrace(System.out);
    System.exit(0);
  }
}

/**
 * This method evaluates the current performance of the contention
 * manager.  This performance is then compared against past performance
 * seen/expected from the other contention managers.  If a contention
 * manager is found that has better potential performance, this method
 * will set the new contention manager and post any required update events.
 */
private void evaluatePerformance()
{
```

```
// update the current performance
performances[AdaptiveCM.currentType.ordinal()] =
        AdaptiveCM.currentPerformance;

// check to see if another manager type would yield better
// performance than what is currently being seen
boolean found = false;
int index = 0;

while(!found && index < MANAGER_TYPES.length)
{
        // if the current performance is worse than what was seen on the
        // other manager type, and don't switch to the same type regardless
        // of performance change
        if(performances[index] - AdaptiveCM.currentPerformance >
                EPSILON && currentType != MANAGER_TYPES[index])
        {
                found = true;

                // create the new update event
                mailbox.clear();
                //mailbox.type = MessageType.PERFORMANCE;
                mailbox.managerType = MANAGER_TYPES[index];
                mailbox.oldPerformance = performances[index];
                mailbox.currentPerformance = AdaptiveCM.currentPerformance;
                mailbox.sendMessage(MessageType.PERFORMANCE);

                // post the event to any listeners
                //this.fireUpdateEvent(event);

                // set the new manager type
                AdaptiveCM.setCurrentManager(MANAGER_TYPES[index]);

        }
        ++index;
}

if(!found)
{
        // check for random switch
        Random rand = new Random();
        int nextRand = rand.nextInt(100);
        if(nextRand < CHANCE_TO_SWITCH)
        {
                int newIndex = nextRand % MANAGER_TYPES.length;

                // create the new update event
                mailbox.clear();
                //mailbox.type = MessageType.RANDOM_SWITCH;
                mailbox.managerType = MANAGER_TYPES[newIndex];
                mailbox.oldPerformance = performances[newIndex];
                mailbox.currentPerformance = AdaptiveCM.currentPerformance;
                mailbox.sendMessage(MessageType.RANDOM_SWITCH);
                // post the event to any listeners
                //this.fireUpdateEvent(event);

                // set the new manager type
                AdaptiveCM.setCurrentManager(MANAGER_TYPES[newIndex]);
            }
            else
            {
            // just send the current performance update
            mailbox.clear();
            //mailbox.type = MessageType.STATIC_PERF;
            mailbox.currentPerformance = AdaptiveCM.currentPerformance;
            mailbox.sendMessage(MessageType.STATIC_PERF);
            }
        }
    }



/************************************************************************
 * Static Methods
 ***********************************************************************/

    /**
     * Accessor for the current manager type
     * @return the current manager type
     */
    public static ManagerType getCurrentManager()
    {
        return AdaptiveCM.currentType;
    }

    /**
     * Look up method to correlate manager type to the class name
     * @param mt the manager type
     * @return the class name of the manager type
     */
    public static String resolveManagerType(ManagerType mt)
    {
        return AdaptiveCM.managerNameMap.get(mt);
    }

    /**
     * This method sets the current contention manager to be used.
     * It also performs all housekeeping required as a result of changing
     * the contention manager
     * @param mt the new manager type
     */
    public static void setCurrentManager(ManagerType mt)
    {
        try
     {
            // store the current type
            AdaptiveCM.currentType = mt;
        Class<?> cm = Class.forName(AdaptiveCM.managerNameMap.get(mt));

        // set the contention manager type on the Thread, which is
        // checked by all child threads/transactions/etc...
        dstm2.Thread.setContentionManagerClass(cm);

        // clear the stats
        AdaptiveCM.benchmark.resetCallCounters();

        AdaptiveCM.currentPerformance = 0;
        AdaptiveCM.timeOfLastReset = System.currentTimeMillis();
     }
     catch (Exception e)
     {
        System.out.println(e.getMessage());
        System.exit(0);
     }
    }

}
```

```
package adaptiveCM;


import adaptiveCM.AdaptiveCM.ManagerType;

/**
 * @author jfrank
 *
 */
public class Mailbox {

        public   enum   MessageType   {CALL_STATS,   PERFORMANCE,   STATIC_PERF,
RANDOM_SWITCH}

        public MessageType type;
        public int insertCalls;
        public int removeCalls;
        public int containsCalls;
        public ManagerType managerType;
        public double oldPerformance;
        public double currentPerformance;
        private boolean messageAvailable;


        /**
         * Constructor that initializes the mailbox with default values
         */
        public Mailbox()
        {
                this.clear();
        }


        public synchronized void sendMessage(MessageType type)
        {
                this.type = type;
                this.messageAvailable = true;
                notifyAll();
        }


        public synchronized MessageType getMessage()
        {
                while(!this.messageAvailable)
                {
       try
       {
         wait();
       }
       catch(InterruptedException e) {}
     }

     this.messageAvailable = false;
```

```
    return this.type;
        }


        public synchronized void clear()
        {
                this.type = MessageType.CALL_STATS;
                this.insertCalls = 0;
                this.removeCalls = 0;
                this.containsCalls = 0;
                this.managerType = ManagerType.AGGRESIVE;
                this.oldPerformance = 0;
                this.currentPerformance = 0;
                this.messageAvailable = false;
        }
}


package controller;


/**
 * @author Joel C. Frank
 * This interface defines a controller for use with the AdaptiveCM package
 * connected to dstm2.
 */
public interface Controller extends Runnable {

        public enum ControllerType {CUI, GUI}

}



package controller;


import adaptiveCM.*;
import java.text.*;
import java.io.*;
import javax.swing.event.EventListenerList;


public class CuiController implements Controller, Runnable {

        /**
         * The adaptive contention manager to control
         */
        private static AdaptiveCM adaptiveCM;

        /**
         * The formatter used to format the performance double values
         */
```

```
private DecimalFormat formatter;

private boolean finished;

/**
 * The listener to get input from the user on a different thread
 */
private static ConsoleListener cmdListener;

private static Mailbox mailbox;


/**
 * Constructor
 */
public CuiController()
{
        CuiController.mailbox = new Mailbox();

        // create and start the adaptive contention manager
        CuiController.adaptiveCM  = AdaptiveCM.getInstance(mailbox);
  this.formatter = new DecimalFormat("####.##");
  CuiController.cmdListener = new ConsoleListener();
}


public void run()
{
        // register for cmd events from the console listener
        CuiController.cmdListener.addCmdEventListener(new CmdEventListener() {
                public void cmdEventOccurred(CmdEvent e) {
        try
        {
                // test for simple command
                if(e.cmd.equalsIgnoreCase("quit"))
                {
                        adaptiveCM.finished = true;
                        cmdListener.finished = true;
                        finished = true;
                }
                else if(e.cmd.equalsIgnoreCase("start"))
                {
                        java.lang.Thread t = new java.lang.Thread(adaptiveCM);
                          t.start();
                }
                else if(e.cmd.equalsIgnoreCase("help"))
                {
                        CuiController.displayHelp();
                }
                else
                {
                        System.out.println(parseCompoundCommand(e.cmd));
                }
        }
        catch (Exception exc)
        {
            System.out.println("Unknown Cmd: " + e.cmd + "\ntype " +
                    "'help' for a list of available commands");
        }

      }
        });

        this.finished = false;

        Thread cmdThread = new Thread(CuiController.cmdListener);
        cmdThread.start();

        System.out.println("Adaptive STM v1.0");
        System.out.println("type 'start' to begin (default values are loaded");
        System.out.println("Type 'help' for a list of available commands");

     while(!this.finished)
     {
        try
        {
            switch(CuiController.mailbox.getMessage())
            {
                case CALL_STATS:
                    System.out.println("Inserts:              "              +
CuiController.mailbox.insertCalls +
                        " Removes: " + CuiController.mailbox.removeCalls +
                        " Contains: " + CuiController.mailbox.containsCalls);

                    break;

                case PERFORMANCE:
                    System.out.print("Set new manager type: " +

    AdaptiveCM.resolveManagerType(CuiController.mailbox.managerType));
                    System.out.print("  oldPerf: ");

                    if(CuiController.mailbox.oldPerformance              ==
Double.MAX_VALUE)
                        System.out.print("MAX\n");
                    else
                        System.out.print(

    formatter.format(CuiController.mailbox.oldPerformance) + "\n");

                    printPerformance(mailbox.currentPerformance);
                    break;

                case STATIC_PERF:
                    printPerformance(mailbox.currentPerformance);
                    break;
```

```
                        case RANDOM_SWITCH:
                                System.out.print("Randomly set new manager type: " +

AdaptiveCM.resolveManagerType(mailbox.managerType));
                                System.out.print("  oldPerf: ");

                                if(mailbox.oldPerformance == Double.MAX_VALUE)
                                        System.out.print("MAX\n");
                                else
                                        System.out.print(

formatter.format(mailbox.oldPerformance) + "\n");

                                printPerformance(mailbox.currentPerformance);
                                break;
                }
        }
        catch (Exception e) {}
  }
}

private void printPerformance(double perf)
{
        System.out.println("Current Perf: " + formatter.format(perf));
        System.out.println();
}

/**
 * This method displays the help listing of available commands
 */
private static void displayHelp()
{
        System.out.println("\n*********************************************");
        System.out.println("************* Available Commands *************");
        System.out.println("*********************************************\n");
        System.out.println("start - starts the experiment");
        System.out.println(
                        "quit - quits the experiment and " +
                        "displays this summary");
        System.out.println("help - displays this help listing");
        System.out.println("get <field> - gets the current value for a field");
        System.out.println("set <field> <value> - sets 'value' for the " +
                        "given 'field'");
        System.out.println("\nfield options: manager | benchmark | adapt");
        System.out.println("    values for manager:\t\ttagg | backoff | eruption" +
                        " | greedy | karma");
        System.out.println("    values for benchmark:\tlist | listRelease |" +
                " rbTree | skipList");
        System.out.println("    values for adapt:\t\ttrue | false");
        System.out.println("*********************************************");
}
```

```
        /**
         * This method parses a compound command string from the user
         * @param cmd the user's command string to parse
         * @return an ack string, or an error string if the command was not
         * recognized
         */
        private static String parseCompoundCommand(String cmd)
        {
                String error = "Unknown Cmd: " + cmd + "\n**Type 'help' for a list of " +
                                "available commands**";

                String[] tokens = cmd.split(" ");
                if(tokens.length < 2 || tokens.length > 3)
                        return error;

                if(tokens[0].equalsIgnoreCase("set"))
                {
                        //process the "set" command
                        if(tokens[1].equalsIgnoreCase("manager"))
                        {
                                if(tokens[2].equalsIgnoreCase("agg"))
                                        AdaptiveCM.currentType                                  =
AdaptiveCM.ManagerType.AGGRESIVE;
                                else if(tokens[2].equalsIgnoreCase("backoff"))
                                        AdaptiveCM.currentType                                  =
AdaptiveCM.ManagerType.BACKOFF;
                                else if(tokens[2].equalsIgnoreCase("eruption"))
                                        AdaptiveCM.currentType                                  =
AdaptiveCM.ManagerType.ERUPTION;
                                else if(tokens[2].equalsIgnoreCase("greedy"))
                                        AdaptiveCM.currentType                                  =
AdaptiveCM.ManagerType.GREEDY;
                                else if(tokens[2].equalsIgnoreCase("karma"))
                                        AdaptiveCM.currentType                                  =
AdaptiveCM.ManagerType.KARMA;
                                else
                                        return error;
                        }
                        else if(tokens[1].equalsIgnoreCase("benchmark"))
                        {
                                if(tokens[2].equalsIgnoreCase("list"))
                                        AdaptiveCM.benchmarkClassName                           =
"dstm2.benchmark.List";
                                else if(tokens[2].equalsIgnoreCase("listRelease"))
                                        AdaptiveCM.benchmarkClassName =
                                                "dstm2.benchmark.ListRelease";
                                else if(tokens[2].equalsIgnoreCase("rbTree"))
                                        AdaptiveCM.benchmarkClassName                           =
"dstm2.benchmark.RBTree";
                                else if(tokens[2].equalsIgnoreCase("skipList"))
                                        AdaptiveCM.benchmarkClassName                           =
"dstm2.benchmark.SkipList";
                                else
                return error;
        }
        else if(tokens[1].equalsIgnoreCase("adapt"))
        {
            if(tokens[2].equalsIgnoreCase("true"))
            {
                CuiController.adaptiveCM.runAdaptively = true;
            }
            else if(tokens[2].equalsIgnoreCase("false"))
                CuiController.adaptiveCM.runAdaptively = false;
            else
                return error;
        }
        else
            return error;
    }
    else if(tokens[0].equalsIgnoreCase("get"))
    {
        //process the "set" command
        if(tokens[1].equalsIgnoreCase("manager"))
            return AdaptiveCM.resolveManagerType(AdaptiveCM.currentType);
        else if(tokens[1].equalsIgnoreCase("benchmark"))
            return AdaptiveCM.benchmarkClassName;
        else if(tokens[1].equalsIgnoreCase("adapt"))
        {
            if(CuiController.adaptiveCM.runAdaptively)
                return "true";
            else
                return "false";
        }
        else
            return error;
    }

    return cmd + " OK!";
}



/**
 * This class is used to get input from the user on a seperate thread
 */
private class ConsoleListener implements Runnable
{
    /**
     * The list of event listeners for this class
     */
    protected EventListenerList listenerList;

    /**
     * The buffered reader to read from the console
     */
     private BufferedReader in;

     /**
      * flag to tell this thread to stop
      */
     public boolean finished = false;




     public ConsoleListener()
     {
             // create the buffered reader to read from the console
             this.in = new BufferedReader(new InputStreamReader(System.in));

             //create the listener list to be used for update events
  this.listenerList = new EventListenerList();
     }


     public void run()
     {
             while(!this.finished)
             {
                     try
                     {
                             String input = this.in.readLine();

                             CmdEvent e = new CmdEvent(this, input);
                             this.fireCmdEvent(e);
                     }
                     catch (IOException e) {}
             }
     }

     /**
      * This methods allows classes to register for CmdEvents
      * @param listener the listener to add
      */
public void addCmdEventListener(CmdEventListener listener) {
  this.listenerList.add(CmdEventListener.class, listener);
}

/**
 * This methods allows classes to unregister for CmdEvents
 * @param listener the listener to remove
 */
public void removeCmdListener(CmdEventListener listener) {
  this.listenerList.remove(CmdEventListener.class, listener);
}

/**
      * This method posts an UpdateEvent to all registered listeners
```

```
                    * @param e the UpdateEvent to post
                    */
               private void fireCmdEvent(CmdEvent e) {
                  Object[] listeners = this.listenerList.getListenerList();

                  // Each listener occupies two elements - the first is the listener class
                  // and the second is the listener instance
                  for (int i=0; i<listeners.length; i+=2) {
                     if (listeners[i]==CmdEventListener.class) {
                        ((CmdEventListener)listeners[i+1]).cmdEventOccurred(e);
                     }
                  }
               }

          }


}


package controller;

import java.util.EventObject;


/**
 * @author Joel C. Frank
 * This class is a wrapper class for sting based commands.
 */
public class CmdEvent extends EventObject
{
          private static final long serialVersionUID = 42;

          public String cmd;

          public CmdEvent(Object o, String cmd)
          {
                    super(o);
                    this.cmd = cmd;
          }

}
```

```
package controller;

import java.util.EventListener;


/**
 * @author Joel C. Frank
 * Listener interface for CmdEvents.  Classes wishing to receive
 * CmdEvents must implement this interface
 */
public interface CmdEventListener extends EventListener
{
        public void cmdEventOccurred(CmdEvent e);
}
```

```
/*
 * Main.java
 *
 * Copyright 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 * Clara, California 95054, U.S.A. All rights reserved.
 *
 * Sun Microsystems, Inc. has intellectual property rights relating to
 * technology embodied in the product that is described in this
 * document. In particular, and without limitation, these
 * intellectual property rights may include one or more of the
 * U.S. patents listed at http://www.sun.com/patents and one or more
 * additional patents or pending patent applications in the U.S. and
 * in other countries.
 *
 * U.S. Government Rights - Commercial software.
 * Government users are subject to the Sun Microsystems, Inc. standard
 * license agreement and applicable provisions of the FAR and its
 * supplements. Use is subject to license terms. Sun, Sun
 * Microsystems, the Sun logo and Java are trademarks or registered
 * trademarks of Sun Microsystems, Inc. in the U.S. and other
 * countries.
 *
 * This product is covered and controlled by U.S. Export Control laws
 * and may be subject to the export or import laws in other countries.
 * Nuclear, missile, chemical biological weapons or nuclear maritime
 * end uses or end users, whether direct or indirect, are strictly
 * prohibited. Export or reexport to countries subject to
 * U.S. embargo or to entities identified on U.S. export exclusion
 * lists, including, but not limited to, the denied persons and
 * specially designated nationals lists is strictly prohibited.
 */

package dstm2;
import static dstm2.Defaults.*;
import dstm2.benchmark.Benchmark;
import controller.Controller.ControllerType;
import controller.*;

public class Main {

        private static int numThreads = THREADS;
        private static int experiment = EXPERIMENT;
        public static String managerClassName = null;
        private static Class managerClass = null;
        public static String benchmarkClassName = BENCHMARK;
        private static Class benchmarkClass = null;
        public static String adapterClassName = ADAPTER;
        private static Controller controller = null;
        private static ControllerType controllerType = ControllerType.CUI;
        private volatile static Benchmark benchmark = null;
        private static long startTime;
        private static Thread[] thread;
 /**
  * @param args the command line arguments
  * usage: dstm.benchmark.Main  -b <benchmarkclass> [-m <managerclass>] [-t <#threads>] [-n <#time-
in-ms>] [-e <experiment#>] [-f <factory>"
  */
 public static void main(String args[]) {

  // discard statistics from previous runs
  Thread.clear();

  // Parse and check the args
  int argc = 0;
  try {
   while (argc < args.length) {
    String option = args[argc++];
    if(option.equals("-c"))
    {
       String type = args[argc];
       if(controllerType.equals("cui"))
              controllerType = ControllerType.CUI;
       else if(type.equals("gui"))
              controllerType = ControllerType.GUI;
       else
              reportUsageErrorAndDie();
    }
    else if (option.equals("-m"))
     managerClassName = args[argc];
    else if (option.equals("-b"))
     benchmarkClassName = args[argc];
    else if (option.equals("-t"))
     numThreads = Integer.parseInt(args[argc]);
    else if (option.equals("-e"))
     experiment = Integer.parseInt(args[argc]);
    else if (option.equals("-a"))
     adapterClassName = args[argc];
    else
     reportUsageErrorAndDie();
    argc++;
   }
  } catch (NumberFormatException e) {
   System.out.println("Expected a number: " + args[argc]);
   System.exit(0);
  } catch (Exception e) {
   reportUsageErrorAndDie();
  }


  // create and start the adaptive contention manager
  switch(controllerType)
       {
              case CUI: controller = new CuiController(); break;
```

```
        //case GUI: controller = new GuiController(); break;
        default: break;
    }

 java.lang.Thread t = new java.lang.Thread(controller);
 t.start();
}

public static void initialize()
{
    // Initialize contention manager.
    try
    {
        managerClass = Class.forName(managerClassName);
      Thread.setContentionManagerClass(managerClass);
    }
    catch (ClassNotFoundException ex)
    {
        reportUsageErrorAndDie();
    }

    // Initialize adapter class
    Thread.setAdapterClass(adapterClassName);

    // initialize benchmark
    try
    {
        benchmarkClass = Class.forName(benchmarkClassName);
        benchmark = (Benchmark) benchmarkClass.newInstance();
    }
    catch (InstantiationException e)
    {
        System.out.format("%s does not implement "
                + "dstm.benchmark.Benchmark: %s\n", benchmarkClass, e);
        System.exit(0);
    }
    catch (ClassCastException e)
    {
        System.out.format("Exception when creating class %s: %s\n",
                benchmarkClass, e);
        System.exit(0);
    }
    catch (Exception e)
    {
        e.printStackTrace(System.out);
        System.exit(0);
    }
}

public static void startExperiment()
{
    // Set up the benchmark
    startTime = 0;
    thread = new Thread[numThreads];
    System.out.println("Benchmark: " + benchmarkClass);
    System.out.println("Adapter: " + adapterClassName);
    System.out.println("Contention manager: " + managerClassName);
    System.out.println("Threads: " + numThreads);
    System.out.println("Mix: " + experiment + "% updates");

    try {
        for (int i = 0; i < numThreads; i++)
            thread[i] = benchmark.createThread(experiment);
        startTime = System.currentTimeMillis();
        for (int i = 0; i < numThreads; i++)
            thread[i].start();
    } catch (Exception e) {
        e.printStackTrace(System.out);
        System.exit(0);
    }
}

public static void stopExperiment()
{
    try
    {
        Thread.stop = true;     // notify threads to stop
     for (int i = 0; i < numThreads; i++)
      thread[i].join();
    }
    catch (Exception e) {
     e.printStackTrace(System.out);
     System.exit(0);
    }

    long stopTime = System.currentTimeMillis();

    double elapsed = (double)(stopTime - startTime) / 1000.0;

    // Run the sanity check for this benchmark
    try {
        benchmark.sanityCheck();
    } catch (Exception e) {
        e.printStackTrace(System.out);
    }

    long committed = Thread.totalCommitted;
    long total = Thread.totalTotal;
    if (total > 0) {
        System.out.printf("Committed: %d\n" +
                       "Total: %d\n" +
                       "Percent committed: (%d%%)\n",
                       committed,
                       total,
                       (100 * committed) / total);
        } else {
                System.out.println("No transactions executed!");
        }
        benchmark.printReport();
        System.out.println("Elapsed time: " + elapsed + " seconds.");
        System.out.println("--------------------------------------");
  }

  public static Benchmark getBenchmark() { return benchmark; }

  private static void reportUsageErrorAndDie() {
    System.out.println("usage: dstm2.Main [-c <controller type (cui|gui)>] [-b <benchmarkclass>] [-m
<managerclass>] [-t <#threads>] [-e <experiment#>] [-a <adapter>]");
    System.exit(0);
  }

}
```

```
/*
 * Thread.java
 *
 * Copyright 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 * Clara, California 95054, U.S.A.  All rights reserved.
 *
 * Sun Microsystems, Inc. has intellectual property rights relating to
 * technology embodied in the product that is described in this
 * document.  In particular, and without limitation, these
 * intellectual property rights may include one or more of the
 * U.S. patents listed at http://www.sun.com/patents and one or more
 * additional patents or pending patent applications in the U.S. and
 * in other countries.
 *
 * U.S. Government Rights - Commercial software.
 * Government users are subject to the Sun Microsystems, Inc. standard
 * license agreement and applicable provisions of the FAR and its
 * supplements.  Use is subject to license terms.  Sun, Sun
 * Microsystems, the Sun logo and Java are trademarks or registered
 * trademarks of Sun Microsystems, Inc. in the U.S. and other
 * countries.
 *
 * This product is covered and controlled by U.S. Export Control laws
 * and may be subject to the export or import laws in other countries.
 * Nuclear, missile, chemical biological weapons or nuclear maritime
 * end uses or end users, whether direct or indirect, are strictly
 * prohibited.  Export or reexport to countries subject to
 * U.S. embargo or to entities identified on U.S. export exclusion
 * lists, including, but not limited to, the denied persons and
 * specially designated nationals lists is strictly prohibited.
 */

package dstm2;
```

```
import dstm2.exceptions.AbortedException;
import dstm2.exceptions.GracefulException;
import dstm2.exceptions.PanicException;
import dstm2.exceptions.SnapshotException;
import dstm2.factory.AtomicFactory;
import dstm2.factory.Factory;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.Callable;
import static dstm2.Defaults.*;
/**
 * The basic unit of computation for the transactional memory.  This
 * class extends <code>java.lang.Thread</code> by providing methods to
 * begin, commit and abort transactions.
 *
 * Every <code>Thread</code> has a contention manager, created when
 * the thread is created.  Before creating any <code>Thread</code>s,
 * you must call <code>Thread.setContentionManager</code> to set the
 * class of the contention manager that will be created.  The
 * contention manager of a thread is notified (by invoking its
 * notification methods) of the results of any methods involving the
 * thread.  It is also consulted on whether a transaction should be
 * begun.
 *
 * @see dstm2.ContentionManager
 */
public class Thread extends java.lang.Thread {
 /**
  * Contention manager class.
  */
 protected static Class contentionManagerClass;

 /**
  * Adapter class.
  */
 protected static Class<dstm2.factory.Adapter> adapterClass;

 /**
  * Set to true when benchmark runs out of time.
  **/
 public static volatile boolean stop = false;
 /**
  * number of committed transactions for all threads
  */
```

```
public static long totalCommitted = 0;
/**
 * total number of transactions for all threads
 */
public static long totalTotal = 0;
/**
 * number of committed memory references for all threads
 */
public static long totalCommittedMemRefs = 0;
/**
 * total number of memory references for all threads
 */
public static long totalTotalMemRefs = 0;

static ThreadLocal<ThreadState> _threadState = new ThreadLocal<ThreadState>() {
 protected synchronized ThreadState initialValue() {
  return new ThreadState();
 }
};
static ThreadLocal<Thread> _thread = new ThreadLocal<Thread>() {
 protected synchronized Thread initialValue() {
  return null;
 }
};

private static int MAX_NESTING_DEPTH = 1;

private static Object lock = new Object();

// Memo-ize factories so we don't have to recreate them.
private static Map<Class,Factory> factoryTable
  = Collections.synchronizedMap(new HashMap<Class,Factory>());

/**
 * Create thread to run a method.
 * @param target execute this object's <CODE>run()</CODE> method
 */
public Thread(final Runnable target) {
 super(new Runnable() {
  public void run() {
   ThreadState threadState = _threadState.get();
   threadState.reset();
   target.run();
   // collect statistics
   synchronized (lock){
    totalCommitted += threadState.committed;
    totalTotal += threadState.total;
    totalCommittedMemRefs += threadState.committedMemRefs;
    totalTotalMemRefs += threadState.totalMemRefs;
   }
  }
 });
}
```

```
/**
 * no-arg constructor
 */
public Thread() {
 super();
}

/**
 * Establishes a contention manager.  You must call this method
 * before creating any <code>Thread</code>.
 *
 * @see dstm2.ContentionManager
 * @param theClass class of desired contention manager.
 */
public static void setContentionManagerClass(Class theClass) {
 Class cm;
 try {
  cm = Class.forName("dstm2.ContentionManager");
 } catch (ClassNotFoundException e) {
  throw new PanicException(e);
 }
 try {
  contentionManagerClass = theClass;
  _threadState.get().manager = (ContentionManager)Thread.contentionManagerClass.newInstance();
 } catch (Exception e) {
  throw new PanicException("The class " + theClass
     + " does not implement dstm2.ContentionManager");
 }
}

/**
 * set Adapter class for this thread
 * @param adapterClassName adapter class as string
 */
public static void setAdapterClass(String adapterClassName) {
 try {
  adapterClass = (Class<dstm2.factory.Adapter>)Class.forName(adapterClassName);
 } catch (ClassNotFoundException ex) {
  throw new PanicException("Adapter class not found: %s\n", adapterClassName);
 }
}

/**
 * Tests whether the current transaction can still commit.  Does not
 * actually end the transaction (either <code>commitTransaction</code> or
 * <code>abortTransaction</code> must still be called).  The contention
 * manager of the invoking thread is notified if the onValidate fails
 * because a <code>TMObject</code> opened for reading was invalidated.
 *
 * @return whether the current transaction may commit successfully.
 */
static public boolean validate() {
 ThreadState threadState = _threadState.get();
 return threadState.validate();
}

/**
 * Gets the current transaction, if any, of the invoking <code>Thread</code>.
 *
 * @return the current thread's current transaction; <code>null</code> if
 *       there is no current transaction.
 */
static public Transaction getTransaction() {
 return _threadState.get().transaction;
}

/**
 * Gets the contention manager of the invoking <code>Thread</code>.
 *
 * @return the invoking thread's contention manager
 */
static public ContentionManager getContentionManager() {
 return _threadState.get().manager;
}

/**
 * Create a new factory instance.
 * @param _class class to implement
 * @return new factory
 */
static public <T> Factory<T> makeFactory(Class<T> _class) {
 try {
  Factory<T> factory = (Factory<T>) factoryTable.get(_class);
  if (factory == null) {
   factory =  new AtomicFactory<T>(_class, adapterClass);
   factoryTable.put(_class, factory);
  }
  return factory;
 } catch (Exception e) {
  throw new PanicException(e);
 }
}

/**
 * Execute a transaction
 * @param xaction execute this object's <CODE>call()</CODE> method.
 * @return result of <CODE>call()</CODE> method
 */
public static <T> T doIt(Callable<T> xaction) {
 ThreadState threadState = _threadState.get();
 ContentionManager manager = threadState.manager;
 T result = null;
 try {
  while (!Thread.stop) {
   threadState.beginTransaction();
   try {
        result = xaction.call();
      } catch (AbortedException d) {
      } catch (SnapshotException s) {
        threadState.abortTransaction();
      } catch (Exception e) {
        e.printStackTrace();
        throw new PanicException("Unhandled exception " + e);
      }
      threadState.totalMemRefs += threadState.transaction.memRefs;
      if (threadState.commitTransaction()) {
        threadState.committedMemRefs += threadState.transaction.memRefs;
        return result;
      }
      threadState.transaction.attempts++;
      // transaction aborted
    }
    if (threadState.transaction != null) {
      threadState.abortTransaction();
    }
  } finally {
    threadState.transaction = null;
  }
  // collect statistics
  synchronized (lock){
    totalTotalMemRefs = threadState.totalMemRefs;
    totalCommittedMemRefs = threadState.committedMemRefs;
    totalCommitted += threadState.committed;
    totalTotal += threadState.total;
    threadState.reset();  // set up for next iteration
  }
  throw new GracefulException();
}
/**
 * Execute transaction
 * @param xaction call this object's <CODE>run()</CODE> method
 */
public static void doIt(final Runnable xaction) {
  doIt(new Callable<Boolean>() {
    public Boolean call() {
      xaction.run();
      return false;
    };
  });
}

/**
 * number of transactions committed by this thread
 * @return number of transactions committed by this thread
 */
public static long getCommitted() {
  return totalCommitted;
}
```

```
/**
 * umber of transactions aborted by this thread
 * @return number of aborted transactions
 */
public static long getAborted() {
 return totalTotal -  totalCommitted;
}

/**
 * number of transactions executed by this thread
 * @return number of transactions
 */
public static long getTotal() {
 return totalTotal;
}

/**
 * Register a method to be called every time this thread validates any transaction.
 * @param c abort if this object's <CODE>call()</CODE> method returns false
 */
public static void onValidate(Callable<Boolean> c) {
 _threadState.get().onValidate.add(c);
}
/**
 * Register a method to be called every time the current transaction is validated.
 * @param c abort if this object's <CODE>call()</CODE> method returns false
 */
public static void onValidateOnce(Callable<Boolean> c) {
 _threadState.get().onValidateOnce.add(c);
}
/**
 * Register a method to be called every time this thread commits a transaction.
 * @param r call this object's <CODE>run()</CODE> method
 */
public static void onCommit(Runnable r) {
 _threadState.get().onCommit.add(r);
}
/**
 * Register a method to be called once if the current transaction commits.
 * @param r call this object's <CODE>run()</CODE> method
 */
public static void onCommitOnce(Runnable r) {
 _threadState.get().onCommitOnce.add(r);
}
/**
 * Register a method to be called every time this thread aborts a transaction.
 * @param r call this objec't <CODE>run()</CODE> method
 */
public static void onAbort(Runnable r) {
 _threadState.get().onAbort.add(r);
}
/**
 * Register a method to be called once if the current transaction aborts.
 * @param r call this object's <CODE>run()</CODE> method
 */
public static void onAbortOnce(Runnable r) {
  _threadState.get().onAbortOnce.add(r);
}
/**
 * get thread ID for debugging
 * @return unique id
 */
public static int getID() {
  return _threadState.get().hashCode();
}

/**
 * reset thread statistics
 */
public static void clear() {
  totalTotal = 0;
  totalCommitted = 0;
  totalCommittedMemRefs = 0;
  totalTotalMemRefs = 0;
  stop = false;
}

/**
 * Class that holds thread's actual state
 */
public static class ThreadState {

  int depth = 0;
  ContentionManager manager;

  private long committed = 0;        // number of committed transactions
  private long total = 0;            // total number of transactions
  private long committedMemRefs = 0; // number of committed reads and writes
  private long totalMemRefs = 0;     // total number of reads and writes

  Set<Callable<Boolean>> onValidate = new HashSet<Callable<Boolean>>();
  Set<Runnable>          onCommit   = new HashSet<Runnable>();
  Set<Runnable>          onAbort    = new HashSet<Runnable>();
  Set<Callable<Boolean>> onValidateOnce = new HashSet<Callable<Boolean>>();
  Set<Runnable>          onCommitOnce   = new HashSet<Runnable>();
  Set<Runnable>          onAbortOnce    = new HashSet<Runnable>();

  Transaction transaction = null;

  /**
   * Creates new ThreadState
   */
  public ThreadState() {
    try {
      manager = (ContentionManager)Thread.contentionManagerClass.newInstance();
    } catch (NullPointerException e) {
      throw new PanicException("No default contention manager class set.");
    } catch (Exception e) {  // Some problem with instantiation
      throw new PanicException(e);
    }
  }

  /**
   * Resets any metering information (commits/aborts, etc).
   */
  public void reset() {
    committed = 0;        // number of committed transactions
    total = 0;            // total number of transactions
    committedMemRefs = 0; // number of committed reads and writes
    totalMemRefs = 0;     // total number of reads and writes
  }

  /**
   * used for debugging
   * @return string representation of thread state
   */
  public String toString() {
    return
        "Thread" + hashCode() + "["+
        "committed: " +  committed + "," +
        "aborted: " + ( total -  committed) +
        "]";
  }

  /**
   * Can this transaction still commit?
   * This method may be called at any time, not just at transaction end,
   * so we do not clear the onValidateOnce table.
   * @return true iff transaction might still commit
   */
  public boolean validate() {
    try {
      // permanent
      for (Callable<Boolean> v : onValidate) {
        if (!v.call()) {
          return false;
        }
      }
      // temporary
      for (Callable<Boolean> v : onValidateOnce) {
        if (!v.call()) {
          return false;
        }
      }
      return transaction.validate();
    } catch (Exception ex) {
      return false;
    }
  }
```

```
/**
 * Call methods registered to be called on commit.
 */
public void runCommitHandlers() {
 try {
  // permanent
  for (Runnable r: onCommit) {
   r.run();
  }
  // temporary
  for (Runnable r: onCommitOnce) {
   r.run();
  }
  onCommitOnce.clear();
  onValidateOnce.clear();
 } catch (Exception ex) {
  throw new PanicException(ex);
 }
}

/**
 * Starts a new transaction.  Cannot nest transactions deeper than
 * <code>Thread.MAX_NESTING_DEPTH.</code> The contention manager of the
 * invoking thread is notified when a transaction is begun.
 */
public void beginTransaction() {
 transaction = new Transaction();
 if (depth == 0) {
  total++;
 }
 // first thing to fix if we allow nested transactions
 if (depth >= 1) {
  throw new PanicException("beginTransaction: attempting to nest transactions too deeply.");
 }
 depth++;
}

/**
 * Attempts to commit the current transaction of the invoking
 * <code>Thread</code>.  Always succeeds for nested
 * transactions.  The contention manager of the invoking thread is
 * notified of the result.  If the transaction does not commit
 * because a <code>TMObject</code> opened for reading was
 * invalidated, the contention manager is also notified of the
 * inonValidate.
 *
 *
 * @return whether commit succeeded.
 */
public boolean commitTransaction() {
 depth--;
 if (depth < 0) {
        throw new PanicException("commitTransaction invoked when no transaction active.");
      }
      if (depth > 0) {
        throw new PanicException("commitTransaction invoked on nested transaction.");
      }
      if (depth == 0) {
        if (validate() && transaction.commit()) {
          committed++;
          runCommitHandlers();
          return true;
        }
        abortTransaction();
        return false;
      } else {
        return true;
      }
    }

    /**
     * Aborts the current transaction of the invoking <code>Thread</code>.
     * Does not end transaction, but ensures it will never commit.
     */
    public void abortTransaction() {
      runAbortHandlers();
      transaction.abort();
    }

    /**
     * Call methods registered to be called on commit.
     */
    public void runAbortHandlers() {
      try {
        // permanent
        for (Runnable r: onAbort) {
          r.run();
        }
        // temporary
        for (Runnable r: onAbortOnce) {
          r.run();
        }
        onAbortOnce.clear();
        onValidateOnce.clear();
      } catch (Exception ex) {
        throw new PanicException(ex);
      }
    }
  }
}
```

```
/*
 * Defaults.java
 *
 * Copyright 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 * Clara, California 95054, U.S.A.  All rights reserved.
 *
 * Sun Microsystems, Inc. has intellectual property rights relating to
 * technology embodied in the product that is described in this
 * document.  In particular, and without limitation, these
 * intellectual property rights may include one or more of the
 * U.S. patents listed at http://www.sun.com/patents and one or more
 * additional patents or pending patent applications in the U.S. and
 * in other countries.
 *
 * U.S. Government Rights - Commercial software.
 * Government users are subject to the Sun Microsystems, Inc. standard
 * license agreement and applicable provisions of the FAR and its
 * supplements.  Use is subject to license terms.  Sun, Sun
 * Microsystems, the Sun logo and Java are trademarks or registered
 * trademarks of Sun Microsystems, Inc. in the U.S. and other
 * countries.
 *
 * This product is covered and controlled by U.S. Export Control laws
 * and may be subject to the export or import laws in other countries.
 * Nuclear, missile, chemical biological weapons or nuclear maritime
 * end uses or end users, whether direct or indirect, are strictly
 * prohibited.  Export or reexport to countries subject to
 * U.S. embargo or to entities identified on U.S. export exclusion
 * lists, including, but not limited to, the denied persons and
 * specially designated nationals lists is strictly prohibited.
 */

package dstm2;

/**
 *
 * @author Maurice Herlihy
 */
public class Defaults {
 /**
  * how many threads
  **/
 public static final int THREADS = 10;
 /**
  * benchmark duration in milliseconds
  **/
 public static final int TIME = 5000;
 /**
  * uninterpreted arg passed to benchmark
  **/
 public static final int EXPERIMENT = 10;
 /**
```

```
  * fully-qualified contention benchmark name
  **/
 public static final String BENCHMARK = "dstm2.benchmark.List";
 /**
  * fully-qualified contention manager name
  **/
 public static final String MANAGER = "dstm2.manager.AggressiveManager";
 /**
  * fully-qualified factory name
  **/
 public static final String FACTORY = "dstm2.factory.shadow.Factory";
 /**
  * fully-qualified adapter name
  **/
 public static final String ADAPTER = "dstm2.factory.shadow.Adapter";;

}
```

```
/*
 * IntSetBenchmark.java
 *
 * Copyright 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 * Clara, California 95054, U.S.A.  All rights reserved.
 *
 * Sun Microsystems, Inc. has intellectual property rights relating to
 * technology embodied in the product that is described in this
 * document.  In particular, and without limitation, these
 * intellectual property rights may include one or more of the
 * U.S. patents listed at http://www.sun.com/patents and one or more
 * additional patents or pending patent applications in the U.S. and
 * in other countries.
 *
 * U.S. Government Rights - Commercial software.
 * Government users are subject to the Sun Microsystems, Inc. standard
 * license agreement and applicable provisions of the FAR and its
 * supplements.  Use is subject to license terms.  Sun, Sun
 * Microsystems, the Sun logo and Java are trademarks or registered
 * trademarks of Sun Microsystems, Inc. in the U.S. and other
 * countries.
 *
 * This product is covered and controlled by U.S. Export Control laws
 * and may be subject to the export or import laws in other countries.
 * Nuclear, missile, chemical biological weapons or nuclear maritime
 * end uses or end users, whether direct or indirect, are strictly
 * prohibited.  Export or reexport to countries subject to
 * U.S. embargo or to entities identified on U.S. export exclusion
 * lists, including, but not limited to, the denied persons and
 * specially designated nationals lists is strictly prohibited.
 */

package dstm2.benchmark;


import dstm2.exceptions.GracefulException;
import dstm2.Thread;
import dstm2.benchmark.Benchmark;
import dstm2.util.Random;
import java.util.Iterator;
import java.util.concurrent.Callable;

/**
 * This abstract class is the superclass for the integer set benchmarks.
 * @author Maurice Herlihy
 * @date April 2004
 */
public abstract class IntSetBenchmark implements Benchmark, Iterable<Integer> {

  /**
   * How large to initialize the integer set.
   */
```

```
protected final int INITIAL_SIZE = 8;

/**
 * After the run is over, synchronize merging statistics with other threads.
 */
static final Object lock = new Object();
/**
 * local variable
 */
int element;
/**
 * local variable
 */
int value;

/**
 * Number of calls to insert()
 */
int insertCalls = 0;

public int getInsertCalls() { return insertCalls; }

/**
 * number of calls to contains()
 */
int containsCalls = 0;

public int getContainsCalls() { return containsCalls; }

/**
 * number of calls to remove()
 */
int removeCalls = 0;

public int getRemoveCalls() { return removeCalls; }

public void resetCallCounters() {
        insertCalls = 0;
        containsCalls = 0;
        removeCalls = 0;
}

/**
 * amount by which the set size has changed
 */
int delta = 0;

/**
 * Give subclass a chance to intialize private fields.
 */
protected abstract void init();

/**
```

```
 * Iterate through set. Not necessarily thread-safe.
 */
public abstract Iterator<Integer> iterator();

/**
 * Add an element to the integer set, if it is not already there.
 * @param v the integer value to add from the set
 * @return true iff value was added.
 */
public abstract boolean insert(int v);

/**
 * Tests wheter a value is in an the integer set.
 * @param v the integer value to insert into the set
 * @return true iff presence was confirmed.
 */
public abstract boolean contains(int v);

/**
 * Removes an element from the integer set, if it is there.
 * @param v the integer value to delete from the set
 * @return true iff v was removed
 */
public abstract boolean remove(int v);

/**
 * Creates a new test thread.
 * @param percent Mix of mutators and observers.
 * @return Thread to run.
 */
public Thread createThread(int percent) {
  try {
    TestThread testThread = new TestThread(this, percent);
    return testThread;
  } catch (Exception e) {
    e.printStackTrace(System.out);
    return null;
  }
}

/**
 * Prints an error message to <code>System.out</code>, including a
 *  standard header to identify the message as an error message.
 * @param s String describing error
 */
protected static void reportError(String s) {
  System.out.println(" ERROR: " + s);
  System.out.flush();
}

public void printReport() {
  System.out.println("Insert/Remove calls:\t" + (insertCalls + removeCalls));
  System.out.println("Contains calls:\t" + containsCalls);
```

```
}

private class TestThread extends Thread {
 IntSetBenchmark intSet;
 /**
  * Thread-local statistic.
  */
 int myInsertCalls = 0;
 /**
  * Thread-local statistic.
  */
 int myRemoveCalls = 0;
 /**
  * Thread-local statistic.
  */
 int myContainsCalls = 0;
 /**
  * Thread-local statistic.
  */
 int myDelta = 0;       // net change
 public int percent = 0; // percent inserts

 TestThread(IntSetBenchmark intSet, int percent) {
  this.intSet = intSet;
  this.percent = percent;
 }

 public void run() {
  Random random = new Random(this.hashCode());
  random.setSeed(System.currentTimeMillis()); // comment out for determinstic

  boolean toggle = true;
  try {
   while (true) {
    boolean result = true;
    element = random.nextInt();
    if (Math.abs(element) % 100 < percent) {
     if (toggle) {       // insert on even turns
      value = element / 100;
      result = Thread.doIt(new Callable<Boolean>() {
       public Boolean call() {
        return intSet.insert(value);
       }
      });
      //myInsertCalls++;
      insertCalls++;
      if (result)
       myDelta++;
     } else {            // remove on odd turns
      result = Thread.doIt(new Callable<Boolean>() {
       public Boolean call() {
        return intSet.remove(value);
       }
```

```
            });
            //myRemoveCalls++;
            removeCalls++;
            if (result)
              this.myDelta--;
          }
          toggle = !toggle;
        } else {
          Thread.doIt(new Callable<Void>() {
            public Void call() {
              intSet.contains(element / 100);
              return null;
            }
          });
          //myContainsCalls++;
          containsCalls++;
        }
      }
    } catch (GracefulException g) {
      // update statistics
      synchronized (lock) {
        //insertCalls   += myInsertCalls;
        //removeCalls   += myRemoveCalls;
        //containsCalls += myContainsCalls;
        delta         += myDelta;
      }
      return;
    }
  }
}

public void sanityCheck() {
  long expected = INITIAL_SIZE + delta;
  int length = 1;

  int prevValue = Integer.MIN_VALUE;
  for (int value : this) {
    length++;
    if (value < prevValue) {
      System.out.println("ERROR: set  not sorted");
      System.exit(0);
    }
    if (value == prevValue) {
      System.out.println("ERROR: set has duplicates!");
      System.exit(0);
    }
    if (length == expected) {
      System.out.println("ERROR: set has bad length!");
      System.exit(0);
    }
  }
  System.out.println("Integer Set OK");
}
```

```
  /**
   * Creates a new IntSetBenchmark
   */
  public IntSetBenchmark() {
    int size = 2;
    init();
    Random random = new Random(this.hashCode());
    while (size < INITIAL_SIZE) {
      if (insert(random.nextInt())) {
        size++;
      }
    }
  }

}
```

BIBLIOGRAPHY

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

Dice, David and Nir Shavit. "What Really Makes Transactions Faster?" *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (2006).

Dynamic Software Transactional Memory Library 2.0. "Dynamic Software Transactional Memory Library 2.0." Sun Microsystems. http://www.sun.com/download/products.xml?id=453fb28e (accessed August 2007).

Ennals, Robert. "Efficient Software Transactional Memory." Technical Report Nr. IRC-TR-05-051. Intel Research Cambridge Tech Report (2005).

Herlihy, Maurice, Victor Luchangco, Mark Moir, and William N. Scherer III. "Software Transactional Memory for Dynamic-Sized Data Structures." *Proceedings of the Twenty-Second annual Symposium on Principles of Distributed Computing (PODC)* (2003): 221.

Kaelbling, L.P., M. L. Littman,, and A.W. Moore. "Reinforcement Learning: A Survey." In *Volume 4*, 237-285. 1996.

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

Lourenco , Joao M.S. and Goncalo T. Cunha. "Testing patterns for software transactional memory engines." *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging* (2007):36-42.

Marathe ,Virendra J., and Michael L. Scott. "A Qualitative Survey of Modern Software Transactional Memory Systems." *Technical Report Nr. TR 839. University of Rochester Computer Science Dept.* (2004).

Marathe , Virendra J., Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. "Lowering the Overhead of Software Transactional Memory." *Technical Report Nr. TR 893. University of Rochester Computer Science Dept.* (2006).

Marathe , Virendra J., William N. Scherer III, and Michael L. Scott. "Design Tradeoffs in Modern Software Transactional Memory Systems." *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (2004).

Minh, Chi Cao, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees." *Proceedings of the 34th Annual International Symposium on Computer Architecture.* (2006).

Riegel , Torvald, Pascal Felber, and Christof Fetzer. "A Lazy Snapshot Algorithm with Eager Validation." *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006. Volume 4167 of Lecture Notes in Computer Science.* (2006): 284-298.

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

Scherer, William N III, and Michael L Scott. "Contention Management in Dynamic Software Transaction Memory." *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs* (2004).

Scherer, William N. III, and Michael L. Scott. "Advanced Contention Management for Dynamic Software Transactional Memory." *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing.* (2005).

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

Tabba, Fuad, Cong Wang, James R. Goodman, and Mark Moir. "NZTM: Nonblocking, Zero-Indirection Transactional Memory." *Workshop on Transactional Computing (TRANSACT)* (2007).

Welc, Adam, Antony L. Hosking, and Suresh Jagannathan. "Transparently Reconciling Transactions with Locking for Java Synchronization." *European Conference on Object-Oriented Programming* (2006): 148-173.

Wikipedia. "Software Transactional Memory." http://en.wikipedia.org/wiki/Software_transactional_memory (accessed February 1, 2008).

Wikipedia . "Machine Learning." http://en.wikipedia.org/wiki/Machine_learning (accessed February 12 2008).