Transactional Memory Scheduling Using Machine Learning Techniques

Basem Assiri, Costas Busch, Mansour Al Ghanim

Assistant Professor, Department of Computer Science, Jazan University

Professor, Division of Computer Science and Engineering, Louisiana State University

Lecturer, Department of Computer Science, Jazan University
Jazan University, Jazan, 45142, Saudi Arabia

Louisiana State University, Baton Rouge, LA 70803, USA

Abstract *— Current shared memory multi-core systems require powerful software and hardware techniques to support the performance parallel computation and consistency simultaneously. The use of transactional memory results in significant improvement of performance by avoiding thread synchronization and locks overhead. Also, transactions scheduling apparently influences the performance of transactional memory. In this paper, we study the fairness of transactions' scheduling using Lazy Snapshot Algorithm. The fairness of transactions' scheduling aims to balance between transactions types which are read-only and update transactions. In the article, we support the fairness of the scheduling procedure by a machine learning technique, which improves the fairness decisions according to transactions history. The experiments in this paper show that the throughput of the Lazy Snapshot Algorithm is improved with a machine learning support. Indeed, our experiments show that the learning significantly affects the performance if the durations of update transactions are much longer than read-only ones or when the cost of abort is very high. We also study several machine learning techniques to investigate the fairness decisions accuracy. In fact, K-Nearest Neighbor machine learning technique shows more accuracy and more suitability, for our problem, than Support Vector Machine Model, Decision Tree Model and Hidden Markov Model.*

Keywords *— Lazy Snapshot Algorithm, Transactional Memory, Fairness Values, Support Vector Machine, K-Nearest Neighbor, Decision Tree Model, Hidden Markov Model*

I. INTRODUCTION

The development in computer science results in a huge increase in data that requires high-performance processing and computation. One of the advanced techniques to enhance and improve quantity and quality of computation is parallel computing state after each action is valid and predictable [11]. We can achieve memory consistency if there are some rules to make the results of the operations' outputs predictable. For example, if x = 1, and there are two operations running simultaneously where one of them writes x = 2 and the other reads the value of x. Actually, the read operation may read the original value of x (x = 1) or the newer value (x = 2). Thus, we must have rules that decide which operation commits first, since the correctness of the execution relays on the order of operations.

In addition, an important way to deal with the difficulty of writing concurrent applications is to use transactional memory. Transactional memory enhances systems' performance because it allows avoiding locks cost and problems [14]. Transaction is a sequence of instructions that access local and shared memory. Those instructions are either read the content of the memory or write content to the memory. A transaction is called read-only transaction if it has only read instructions, and is called update transaction if it has at least one write operation. At the end of execution transaction commits or aborts. Commit means to save all the changes and effects which are made by current transaction. Abort means to ignore all actions and changes that are made by the transaction [20][9].

Actually, different memory techniques are proposed to solve many problems concurrently and to control accessing memory. Thus, asynchronous and synchronous memory algorithms are used to support the multiprocessing techniques, but we need to guarantee that there are no problems as a result of parallelism. Indeed, transactional memory uses the concept of transactions to make look-free synchronization more efficient comparing to mutual exclusion-based techniques (lock-based). Lock-based techniques enable only one thread to enter a critical section which is the part of program that may cause conflicts in parallel execution [9]. However, transactional memory enables multiple threads to execute transactions concurrently and abort transactions that have conflicts. Building on hardware transactional memory, a software transactionalmemory is produced to work as efficiently as hardware one with more flexibility in transactional programming [20].

In this paper, we suggest increasing the throughput of a classic algorithm which is the Lazy Snapshot Algorithm (LSA), by maintaining the scheduling of

the transactions. The fairness of transactions' scheduling is to balance between read-only transactions and update transactions. The read-only transactions do not hurt the system's consistency because they do not change the memory status. In addition, the duration of read-only and update transactions and the transactions' dependencies vary from one system to another which implies that different scheduling is required for different systems. For example, if the system uses an array to store data, then every piece of data would be stored in a specific row in the array (using index). In this case the transaction executes read operation which reads the value that is stored in that row, while the update transaction writes a new value to that row. On the other hand, for any system uses red black tree data structure to store data. The transaction has to traverse the tree to find the required element and read it. For an update transaction, the write operation also has to traverse the tree to the leaf, insert the new node and it may need to recolor and reshape the tree. Thus, the duration of transactions varies from system to system based on the structure of the systems and the memory. So, fairness of scheduling is to decide how many read-only transactions to be committed per update ones and this ratio is called Fairness Value (FV). The FV is selected according to machine learning technique through keeping the track of transactions' history.

By recording a prefix of transactions' execution, we find out the order and number of read-only and update transactions and pass them to the learning model. We use supervised and unsupervised machine learning models which will be explained in detail in sections III and IV. Three supervised machine learning techniques such as Support Vector Machine (SVM), Decision Tree Model (DTM) and K-Nearest Neighbor (KNN) are used for classification. According to the prefix information, SVM, DTM and KNN map the given information to the suitable FV. Furthermore, we compare the results of the supervised machine learning techniques with the Hidden Markov Model (HMM) which is an unsupervised machine learning technique. In fact, our study shows the superiority of KNN over the other models [23] [24].

The rest of this paper is organized as follows: In Section II, we discuss some related works. The design of the supervised machine learning models is presented in Section III. In Section IV, we present the design of the unsupervised machine learning model. Section V discusses our algorithm. The experiments and some results are presented in Section VI, while the Section VII concludes the paper.

II. RELATED WORKS

In this section, we first show some related works that are connected to transactional memory. Then we illustrate some works related to machine learning models. We end the section with some works that combine both of them.

Software Transaction Memory (STM) [20] is successful to support multi-processor systems. Most STM tries to avoid conflicts and guarantee progressiveness in different ways. Devietti et al. [5] show how to acquire determinism and consistency in the execution. Transactions can be classified according to whether the objects' statuses are private or shared, and whether the operations are reads or writes. They offer different techniques to handle determinism. In some situations, thread must get the token to execute a transaction and in other situations thread can execute transaction directly. The token is used to guarantee sequential execution for conflicted transactions. However, this algorithm has two problems which are dead-lock problem (when two threads block each other) and starvation (when thread may wait forever to be executed). In our algorithm, we schedule transaction according to the type of their operations.

The Multi-versions Permissive is a kind of algorithms that keeps many versions of the same object to allow more concurrency [16]. In case of conflict, this algorithm could prevent aborting by re-executing some of the conflicted transactions using the old versions. A well-known example of multi-version algorithm is LSA [18]. With LSA, we check the states of the consistency of the object version at the access moment. Therefore, we can build consistent snapshots during the execution of transaction to assure that transaction reads consistent versions and guarantees correctness of execution. The correctness of transaction's execution is verified if the snapshots of all objects versions it accesses are consistent, which will be explained in detail in section 4. However, we claim that the type, length and order of transactions affect performance [9][13][4][21]. Thus, we suggest scheduling transactions in a way that suits the transactions' content and the data structure they work on.

Moreover, one of the machine learning techniques used in our paper is SVM. SVM is a supervised machine learning technique in which we design a dataset that includes some training examples. Then, SVM classifies the data according to the given examples. The accuracy of classification of SVM will be computed according to the margins and distances among classes [12][3].

Another supervised machine learning technique is K-Nearest Neighbor, which classifies objects based on the nearest training examples [2][19] [24]. In other words, it clusters similar data in classes. Both of them can be used in our algorithm to decide the proper FV.

DTM is proposed to reduce the complexity of sorting and search problems. It traverses the tree from the root to the leaves and in every level it takes a decision on which path to follow. The decision is taken based on a comparison of two numbers within

constant time. A decision tree allows reducing the complexity of a set of elements of size *n* to *log n* [1].

Decision tree learning uses a decision tree to map some inputs which are features or observations about an object to specific outputs. In classification, we traverse a tree and pass some levels (nodes). In each intermediate node, we run a test on the object features and based on the result we decide which branch to follow until we arrive to the leaf which is the suitable class [17].

Another idea presented by Wang suggests using different algorithms according to the inputs' types [22]. Wang uses transactional memory with a machine learning model and with an expert system. However, the experiment focuses on some hardware features such as transactional memory type and cache size. On the other hand, our algorithm uses a learning model to find the suitable FV to guide the scheduling process and improve the performance.

Another work applies Markov Chain to improve STM performance [6]. It uses the Markov Chain for scheduling to control the contention of transactions and decide on blocking temporarily when it is needed. In fact, they focus on contention and the number of transactions running in parallel regardless of the type of these transactions. However, in our paper we use the HMM which is an unsupervised learning model [10][6] [23] to decide how to schedule the transactions based on their types.

III. THE SUPERVISED MACHINE LEARNING TECHNIQUES AND DATASET

In supervised learning, we generate a function that maps inputs to suitable outputs which are called labels or classes. Experts often provide some training examples that supply systems with labels or classes. For example, in classification problems, the learner approximates a function that maps a vector into classes by looking at training examples. Thus, to use a supervised model we have to generate a dataset that includes some training examples. The training examples show how to map the features to the suitable FV. In our dataset, there are three features which are the number of reads-only, the number of updates and the order of the transactions which we call the sequence length.

Our study focuses on the throughput (commit per time) of transactional memory. Thus we maintain transactions' scheduling based on recognizing the deference between the behaviors of read-only transactions and update ones (which is explained in the introduction). Hence we consider the number of read-only and update transactions as features. Also the dependencies between them are very important so we consider the order the transactions using the sequence length feature. In fact, those are the main features that should be considered for throughput of transactional memory.

Actually, we track the prefix of transactions' history (which is a part of transactions' history that precedes the learning process) and pass it to the learning model. We count how many read-only transactions and how many updates are in the prefix. Also, we convert the order of reads and updates into one number and we call it the sequence length. Table I shows an example of how to calculate sequence length for only 10 transactions with different orders. In the first scenario, there are 5 reads followed by 5 updates. The sequence consists of two numbers which are 5 and 5, so the sequence length equals to 2. In the second scenario, the sequence consists of 1 update, 1 read, 1 update, 1 read and so on. It consists of 10 numbers, so the sequence length is 10.

Our dataset consists of four columns which are the three features followed by the suitable FV. To generate the dataset we first need to know the size of the prefix which is a piece of history we track to extract the pattern from. Then we need to find all possible combinations of the three features. For example, if the prefix size is 20, the first combination is 1 read-only, 19 updates and the sequence length is 2, which means the read-only transaction is the first or the last one in the prefix. The next combination will be 1 read-only and 19 updates and the sequence length is 3, which means the read-only transaction is somewhere in the middle of the prefix and so on. Then we pass each permutation as an input set to the LSA and we run it with all FVs we want to test. During the runs we record the throughput of all FVs on our algorithm (LSA that is explained in Section V), and the FV with the maximum throughput will be the suitable class. This way we generate all training examples in the dataset.

In Algorithm 1, we show how to design our training examples. The following procedures explain how we find the suitable FV (class) for each training example:

- We design the training examples in the dataset. We design the dataset array (dataset[][]) which has examples where each example consists of three features and suitable FV (class). The features are the number of read-only, the number of updates, and the sequence length. As

Table I. Two Different Scenarios of 10 Transactions and How to Calculate the Sequence Length (r Means a Read-only Transaction and u Means an Update Transaction)

Scenario 1	**The Sequence**	**The Sequence Length**	**Scenario 2**		**The Sequence**	**The Sequence Length**
r1 r2 r3 r4 r5 u1 u2 u3 u4 u5	5, 5	2	u1 r1 u2 r2 u3 r3 u4 r4 u5 r5		1, 1, 1, 1, 1, 1, 1, 1, 1, 1	10

shown in the algorithm, for example if the prefix *size = 20*, the first combination is 1 read-only, 19 updates and the sequence length is 2. So, we make *dataset [i][0] = i + 1* representing the number of read-only, *dataset [i][1] = y* where *y = size−−* representing the number of updates and *dataset[i][2] = k* where *k = 2* representing the sequence length.

- After we design the training example we create a prefix of history (*permutations[]*) which is a group of transactions that reflect the same numbers in the training example.
- We run the Transactional Memory algorithm (LSA) to execute the transactions in the prefix. In each run of LSA we select different FV to schedule transactions' execution and we record the throughput of each FV in *output[][]*.
- After we run all FVs (that are considered in our experiment), we record all throughputs, we find the maximum throughput and its corresponding FV, and store that in *temp[][]*.
- Then, we change the order of the transactions in the prefix by finding next permutation which preserves the same sequence length; we test it with LSA using all FVs and we store the maximum throughput in *temp[][]*.
- Next we find the maximum throughput of the all permutations that represent this training example (which is the maximum throughput in *temp[][]*, also we get the corresponding FV), and we make that FV as the suitable class of this training example (*dataset[i][4] = F V*).
- Now we design the next training example and keep doing the same process.

For simplicity and to avoid learning process overhead, in our experiment we reduce the size of the prefix to 10 transactions. In ourexperiment, we test all FVs from 1 to 9 and according to the results we select only three FVs which are 1, 4 and 8. The FV= 1 means 1 read-only transaction per 9 updates. The FV= 4 means 4 read-only transactions per 6 updates, while FV= 8 means 8 read-only transactions per 2 updates. More details about FV selecting will be illustrated later in section V.

Table II shows a sample of our dataset which consists of rows and columns. The first row gives a summary of dataset. It states that the number of examples is 54, the number of columns is 4, and shows the three classes. Then, each example is placed individually in a row. For example, in row number 2 in the table, the example 1, 9, 2, 1 means that if there are 1 read-only transaction, 9 update transactions and the sequence length is 2, then the suitable FV is 1. The FV is assigned to the training example based on Algorithm 1.

A. Support Vector Machine

The basic SVM takes a set of input data and predicts the suitable output and each given input will be classified into a suitable class [12]. Fig. 1, shows the SVM classification accuracy which is calculated according to the following formula:

$w*x + b = 0$, where

x denotes the features

w the normal vector to the hyper plane

b denotes misclassification

Higher accuracy of the classification using SVM requires bigger margins among classes.

At the beginning we need to test the classification accuracy of SVM. In fact first two features (which are the number of read-only and the number of update transactions) are more important, so we decide to design and test a dataset with only those two features. Also we select only 3 FVs. We use Scikit-learn to test the machine learning models in this paper (Scikit-learn is a package that is designed to introduce many machine learning algorithms and codes in Python in a simple and understandable way) [15]. The result of running the SVM classifier using the dataset of two features is shown in Fig. 2 (a). In Fig. 2 (a), the x axis represents the number of read-only and y axis represents the number of updates. The classes represent the three FVs. Each point in the figure represents a training example. The accuracy of this classification was about 73% which is not high. Some dots are classified in the wrong class. It is clear that there is misclassification which decreases the classification accuracy. Therefore, the SVM fails at

some point because SVM accuracy increases when the margin between classes is bigger. However, in our model we need to classify some dots where the number of read-only transactions is close to the number of updates and those usually affect the accuracy of SVM [12].

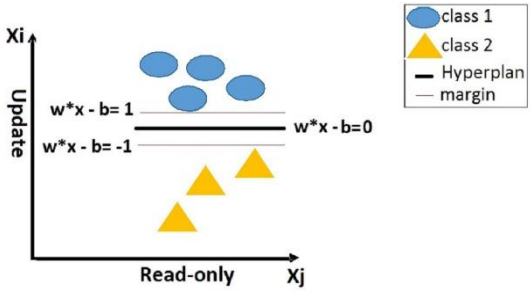

Fig.1:SVM Classifier Uses a Hyperplane to Classify a Set of Elements into Two classes. The Accuracy of Classification Depends on Margin Between Classes.

Table II. Some Training Example from our Dataset (Where the FV is Assigned to the Training Examples Based on Algorithm 1.)

54	4	1	4	8
1	9	2	1	
2	8	2	1	
2	8	5	1	
:				
:				
5	5	2	8	
5	5	5	4	
5	5	10	8	
:				
:				
9	1	2	8	
:				

B. Decision Tree Model

DTM is supervised learning method that maps input to output by following some decision rules. The model gets the input and finds out the features, then based on the features the model traverses the tree starting from the tree's root node. In each node, and based on some decision rules, the model decides which path to take until arriving to the leaf. In classification, the tree leafs are the classes, so the paths of the tree eventually map the input to a suitable class [15]. For example, any boolean rule such as if-then statement represents a tree with many branches.

In fact we decide to use DTM because its cost is very low (the cost of trees is logarithmic) which is important to avoid the learning process overhead. Also the cost is important to cope with the high speed of transactional memory. However the accuracy of DTM might be affected since even small variations in data result in huge variations in the shape of tree which affect the classification process [15].

Fig. 3 shows an example of decision tree where the rules in the nodes is set based on the examples in the dataset. In each node there is a test rule which has yes or no result. For example we test a history that has some read-only and update transactions, by compare it to some integers such a and b. We traverse the tree's tests that are designed according to the training examples until it arrives to one of the classes. Clearly the accuracy of the rules influences the accuracy of the classification.

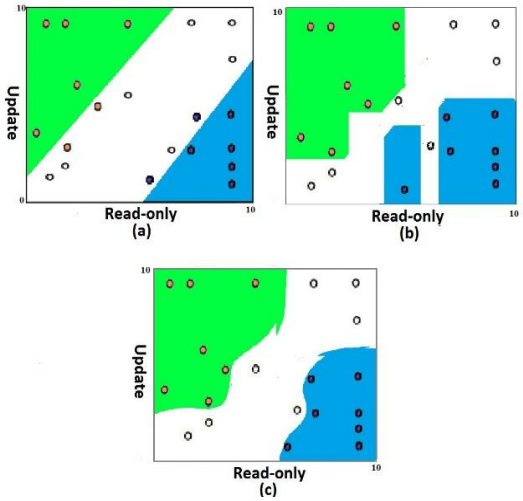

Fig.2: SVM, DTM and KNN with the Dataset of Two Features. We Use the Standard Programs that are Provided by Scikit-learn Package to Measure the Accuracy of the Three Models.

The result of running the DTM classifier using the dataset of two features is shown in Fig. 2 (b). In Fig. 2 (b), the x axis represents the number of read-only and y axis represents the number of updates. The classes represent the three FVs. Each point in the figure represents a training example. The accuracy of this classification is about 85% which is acceptable. Obviously, the dots are classified in suitable classes. However, some classes are broken into many parts since it does not consider the status of the neighbors.

C. K-Nearest Neighbor

The k-Nearest Neighbor algorithm (KNN) is a classifier that classifies objects based on closest training examples in the feature space [2][19].

Algorithm 1: Dataset Training Examples

```
CalculateSequenceLength(); // How many times we switch from read-only to update or from update to read
                           // only
int size; // The number of transaction we track
dataset[][4]; // Dataset array
permutations[]; // All permutations for the same number of read-only and update transactions
```

```
output[rangofFVs][2]; // Record the throughput of LSA using the permutations array as an input
temp[1][2]; // To stroe the temporary FV
max[1][2] = {0, 0}; // To store the best FV
y = size; // The number of updates
// The loop for number of read-only and update transactions
for from i = 0 to size – 1 do
   y--;
   // The loop for sequence length
   for from k = 2 to size do
      dataset[i][0] = i + 1; // The number of read-only
      dataset[i][1] = y; // The number of updates
      dataset[i][2] = k; // The sequence length
      // Now prepare array to find permutations
      n = i + 1;
      for from j = 0 to size – 1 do
         if (n > 0) then
            permutations[j] = 'r'; // r means read-only
            n--;
else permutations[j] = 'u'; // u means update
      // Now we test all permutations with all fairness values to decide the suitable FV
s = 0; // Counter for do while statement
   while (s < factorial(size)) do
         x = CalculateSequenceLength(permutations[]);
         if (k == x) then
            // FVs range is how many fairness values we test
            for from f = 1 to FVrangedo
               FV = F;
               output[F][0] = LSA(permutations[], size, FV); // Record the throughput of out algorithm
               // using the permutations array as an input and
                                                          // the current FV
               output[F][1] = FV;
            temp[0][0] = FindMaximumThroughput(output[][]);
            temp[0][1] = FindMaximumThroughput'sFV(output[][]);
           // Find the FV of the maximum throughput
           if (temp[0][0] > max[0][0]) then
              max[0][1] = temp[0][1]; // Keep the best  FV
        permutations[] = nextpermutations(permutations[], size); // Find next permutations
        s++;
   dataset[i][4] = max[0][1]; // The suitable FV
return;
```

Fig. 2 (c) shows the implementation of KNN on the dataset of two features [15]. Fig. 2 (c) shows that all dots are classified under the correct class. The accuracy of KNN is about 90% which is higher than SVM and higher than DTM. Indeed in Fig. 2 the dots are classified similarly using DTM in subfigure (b) and KNN in subfigure (c), while the accuracy of DTM is 85% and the accuracy of KNN is 90%. That happens because there are different ways to measure the accuracy of classification. Actually the accuracy of every model is measured in specific way [15].

Apparently, the KNN model is very flexible; it is able to classify the critical dots. The superiority of KNN over the other models will be clearly appearing when we have more critical training examples. Also when we execute our algorithm we are going to record the history of transactions and pass a piece of that history to the learning model to predict the suitable FV. So, it is possible for some systems to usually pass data which is considered as critical and difficult to classify. In short KNN model is more suitable for our system than both SVM and DTM. Thus, we decide to use KNN to support the transactions' scheduling process.

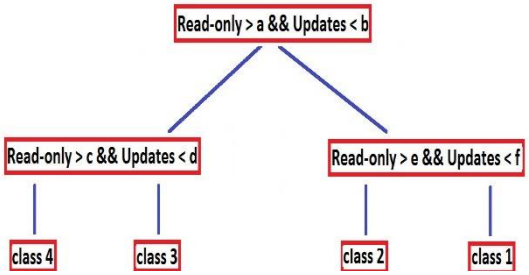

Fig.3: Decision Tree Model

Moreover, we run the KNN classifier using the dataset based on three features that appears in Table

II. At the beginning and we tested the classification accuracy of SVM, DTM and KNN using a dataset with only two features which are the number of read-only and update transactions (for clarity and for the importance of those two features). Actually for more accuracy we need to consider on more feature which is the order of the transactions. For example, the execution a bunch of read only transactions followed by a bunch of updates would be completely different from executing the same number of transactions while they are interleaved. Fig. 4 shows the result of the KNN classification on the three features where each feature is represented by an axis on the graph. The KNN is able to classify the data example into three classes which are red, blue and yellow representing the FVs 1, 4 and 8 respectively.

For simplicity's sake, we do not focus on the overfitting issue in the comparison among the three classifiers. However in our experiments (in section VI) we examine the ability of KNN model to generalize by using data that is collected online and not necessarily used in the training.

IV. AN UNSUPERVISED MACHINE LEARNING MODEL

Unsupervised learning is to model a set of inputs while here labels are unknown during training [8]. In fact, the model itself recognizes labels while it is running. In our problem we use unsupervised learning with HMM since it is very useful in sequence analysis such as in biology and in this research we have to analyze the history which is a sequence of transactions. Since HMM is one of the unsupervised machine learning models, there are unobserved hidden states *X* which must be discovered based on the observed data *Y*. HMM consists of some states and all states have probabilities that help to predict the hidden state [10]. For example, it is difficult to find a scheduling model that suits all systems by increasing throughput and reducing conflicts. Therefore, the suitable transactions' scheduling for each system is the hidden state. However, we can find the hidden state based on some observations such as system performance, the transactions' history and transactions' duration. Furthermore, by recording observations and states frequently, the first result helps to predict the second one and so on. Thus, HMM calculations rely on the following:

- The initial probabilities for hidden states.
- Transition probabilities that tell how to transit among states over time.
- Observation probabilities which are used as indications to find out the hidden states.

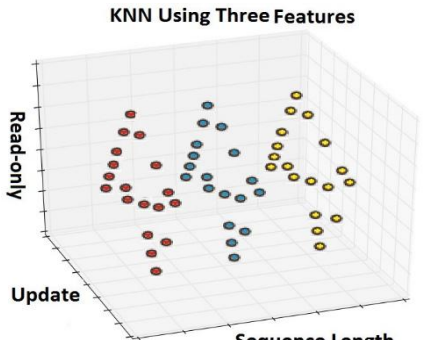

Fig.4: KNN with our Dataset

In our model the hidden states are the FVs. As we mentioned above the FVs we use are 8, 4 and 1. Fig. 5 shows the initial probabilities which we need to start. The probabilities of FVs 8 and 1 are .33 and of FV 4 is .34, so the system more likely starts with state FV= 4. Even if we start with inaccurate probabilities, the model can improve itself over the time by maintaining probabilities based on the observations. Also, as the probability of *X* changes over time, we propose transition probabilities and we favor remaining in the current state. Therefore, the state of *X* at first unit of time T_1 is more likely to remain the same at the second unit of time T_2. While the probability of switching from *X* = 8 or 1 to 4 is .35 and it is less likely to transit between 8 and 1 directly since the probability is just .15. The probability of switching from *X* = 4 to 1 or 8 is .25. The two states at the bottom of Fig. 5 shows the probabilities of our observed data which is Y_1 = read-only and Y_2 = update. Thus, the more reads favors the FV= 8, the more updates favors the FV= 1 and the equal number of both suits the FV= 4. In fact the probabilities in Fig. 5 is selected according to our investigations during the design of the dataset (Algorithm 1) that is used for supervised learning models. However, we try to select fair probabilities to reduce the influence of our decisions on the model.

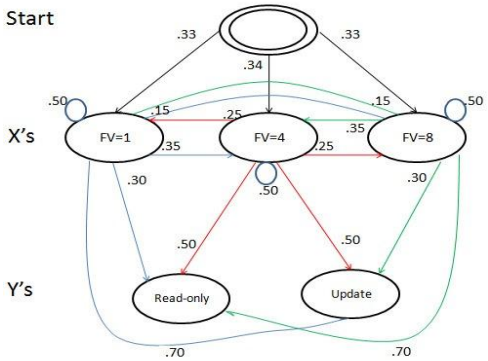

Fig.5: HMM and the Probabilities (Start is the Initial State, X's are the States and Y's are the Observations)

Fig. 6 shows how to build the sequence of states over time. It means to find the state of X_4, we need to calculate the probability of initial state X_1, the

transition probability from X_1 to X_2, the probabilities of Y_1 using X_2, the transition probability from X_2 to X_3, finding Y_2 using X_3, and the transition probability from X_3 to X_4. Indeed, we get the probability from the model (in Fig. 5) and we multiply them to calculate X_4.

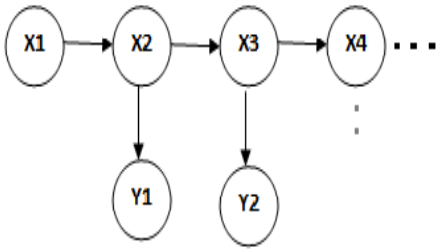

Fig.6: Sequence of States in HMM

V. THE DESIGN OF OUR ALGORITHM

A. *Background about LSA*

Using LSA [18],the memory consists of a set of objects where each object in the memory may have more than one version (a version shows the last written value to the object and the other versions show the old values). This algorithm is used to execute transactions in parallel where some read and write operations may interleave and affect the correctness of each other. Indeed, the correctness of a concurrent execution can be verified by matching it with a sequential valid execution. Many algorithms run transactions and validate a transaction's execution at the end to commit or abort, but LSA verifies the consistency at each object access point of time. Thus, an LSA is able to verify the consistency of execution during each object access by building consistent snapshots. This, can accumulatively verify the validity and the correctness of the execution.

When a transaction starts execution, the transaction sets the lower bound of its snapshots. The read operations try to read the last written version of the object. However, if the last written version was created after the transaction starts, it checks if this version violates the consistency, and if it does, it ignores that version and reads an older one which has a validity range that suits the transaction. This way LSA selects the version that produces the consistent snapshot. The write operations are executed locally and they create new versions at transaction's commit time.

Consequently, read-only transactions commit directly and they do not abort. The update transaction must validate the versions to commit or abort, and if it commits it has to get a unique commit time [18].

B. *Our Scheduling Algorithm*

At the beginning of the execution, our scheduling algorithm (Algorithm 2) executes transactions using LSA. We suppose that we know whether the transaction is read-only or update at the beginning of the execution. We record the transactions' numbers, order and types in *recordingArray[]*. The size of the *recordingArray[]* is related to how much data we want to investigate. It is preferred to keep it as small as possible to avoid the learning overhead and its negative impact on the performance. Then, we pass the recorded data to the learning model which is either KNN or HMM and we conduct online and real-time learning (HMM will treat transactions in *recordingArray[]* as a sequence of observations). The learning model processes the data and returns the best FV that suits the execution. After that, the model uses the proposed FV to schedule transactions on LSA (since different sequence of transactions affecting the execution and the performance of an algorithm. Actually the algorithm verifies the correctness based on the order of the transactions).

In fact, we need to have continuous learning however that influences the performance of the system. Therefore, we add flexibility to the scheduling algorithm using *frequent* which is a number used to decide how frequently we call the learning model. The stability and consistency of the numbers, types and arrival times of transactions differ from one system to another. So, if the system is stable then we have large value for frequent, which means we call the learning model infrequently. In our scheduling algorithm, each transaction arrives increase a global counter *transactionCounter*. The transaction first checks the *frequent* to know if it isthe time to call the learning model or not. Then, if it is the time to call the learning model, then it records

Algorithm2: The Scheduling Algorithm

```
transactionCounter←-1;
size←x; // To set the size of recordingArray[]
recordingArray[size]; // It records transactions' order and type
recording←false;
frequent←number; // It tells how frequently we call learning model.
r← 0; // Counts reads
u← 0; // Counts updates
fvReadonly; // The value assigned by learning model
fvUpdate; // The value assigned by learning model
i← 0;
readonly← 0; // Counts the pending reads-only
update ← 0; // Counts the pending updates

  Upon receipt of a transaction do;

transactionCounter.getAndInc();
if ((transactionCountermod frequent) = 0) then
recording ←true;
if (recording = true) then
i++;
if (i<size) then
if (TransactionType = read-only) then
recordingArray[i] ← 0; // 0 means read-only
else
recordingArray[i] ← 1; // 1 means update
else
// The Learning Model was explained in Section III and IV
fvReadonly←LearningModel(recordingArray[i]);
fvUpdate← (10 – fvReadonly);
i← 0;
recording ←false;
LSA(transaction); // Start execution using LSA
else
if (TransactionType = read-only) then
readonly.getAndInc();
// When it exceeds the FV and there is no update transactions
while ((r >fvReadonly) ^ (update = 0))do
wait();
r++;
LSA(transaction); // Start execution using LSA
readonly.getAndDec();
else
update.getAndInc();
        // when it exceeds the FV and there is no read-only transactions
while ((u >fvUpdate) ^ (readonly = )) do
wait();
u++;
        LSA(transaction); // Start execution using LSA
Update.getAndDec();
If ((r >fvReadonly) ^ (u >fvUpdate)) then
r ← 0;
u ← 0;
return;
```

transactions. In the recording process, each transaction is represented in the *recordingArray*[] as a 0 if it is read-only and as a 1 if it is an update transaction. Then it executes using LSA. After the recording finishes, we pass the *recordingArray*[] to the learning model (KNN or HMM) which returns the FV. The learning model returns the number of read-only *fvReadonly* and from that we calculate the

number of updates *fvUpdate*. From this point, we schedule transactions based on the FV we have. Indeed, we ignore this FV just in case there is only one type of transaction. In the algorithm, we use *readonly* and *update* which are counters telling how many pending transactions there are of each type.

VI. EXPERIMENTAL RESULTS AND DISCUSSION

In our experiment, we use standard benchmarks to verify the benefit of learning on transactional memory scheduling. In fact, we use Bank, Linked-list and Red-black Tree benchmarks from TinySTM-1.0.5 [7]. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5- 2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 10000 milliseconds using 10 threads. In the Bank benchmark, there are three kinds of operations which are read balance, write amount and transfer from one account to another. Read balance is a read-only transaction, while write amount and transfer are update transactions, we consider any transaction that includes one write operation as an update. Initially, there are only 20% reads-only and 80% updates. The benchmark executes millions of transactions that access 1024 bank accounts in parallel.

In the Linked-list and the Red-black Tree benchmarks, there are three kinds of operation which are add node, delete node and contain which searches for specific value in the list or tree. Contain operations are considered as read-only but add node and delete node operations are update transactions. In both, initially there are 80% reads-only and only 20% updates. However, we test the three benchmarks with three different scenarios which are 80% reads-only and 20% updates, 50% reads-only and 50% updates, and 20% reads-only and 80% updates.

First, we run each benchmark against all FVs to obtain a general idea about FVs themselves and how they affect the performance. The FVs we use are ratios out of 10 such that the FV= 9 means 9 : 1, where the first number represents the number of reads-only while the other represents the number of updates. We test FVs from 1 to 9 with all three benchmarks and all scenarios. (Moreover, our algorithm permits to change the scheduling FV in response to any changes of arriving transactions which is note covered in this experiments since that we use the standard benchmarks that generate transactions in specific pattern).

Fig. 7 (a) demonstrates the throughput of FVs (from 1 to 9) with the Linked-list benchmark using different percentages of reads and writes (we test every percentage of the transactions against all FVs). The figure shows that the performance improves as we increase the number of read-only transactions and that happens with the three scenarios. It is clear that the scenario of 80% of reads only is the best, while throughput drops as we increase the number of updates. Also, within each scenario large FVs allow more reads-only to be scheduled concurrently which increase throughput as well. This happens because the reads-only usually have less durations than updates and do not abort as they do not affect the consistency. This increases the throughput and improves the performance.

The same thing happens with the Red-black Tree benchmark. Fig. 7 (b) shows the Red-black Tree throughput with all FVs. The figure shows dramatic increment in the throughput for the three scenarios when the FV allows more reads-only (using large FVs) because the reads-only is much faster than updates. In updates, a transaction has to traverse the entire tree to the leaf and it may need to change the colors of some nodes and the shape of the tree which makes the update transaction much longer. Thus, raising the number of reads-only through the percentage of generated reads-only or through scheduling improves the performance.

Fig. 7 (c) illustrates the Bank benchmark where all FVs have almost the same performance with all scenarios. This happens because of the data structure type, and because of the differences between the duration of read-only transactions and update ones is very small (as explained in the introduction). Indeed, the duration of read-only transactions here is a bit longer than the duration of updates. Thus, the scenario of 20% reads-only and 80% updates achieves the top performance.

In our experiment the FV=9 shows good performance with the Linked-list and the Red-black Tree benchmarks but it cannot be guaranteed for different algorithm and benchmarks. For example, in Bank benchmark some other FVs are better than FV=9. Indeed, our benchmark uses LSA, where any read operation tries to find the suitable version of the object (based on the transaction's timestamp). By keeping many versions, read operation is able to execute correctly and read-only transactions usually commits. Therefore, using FV=9 (to schedule 9 read-only transactions and 1 update) the 9 read-only transactions usually commit which increases the throughput.

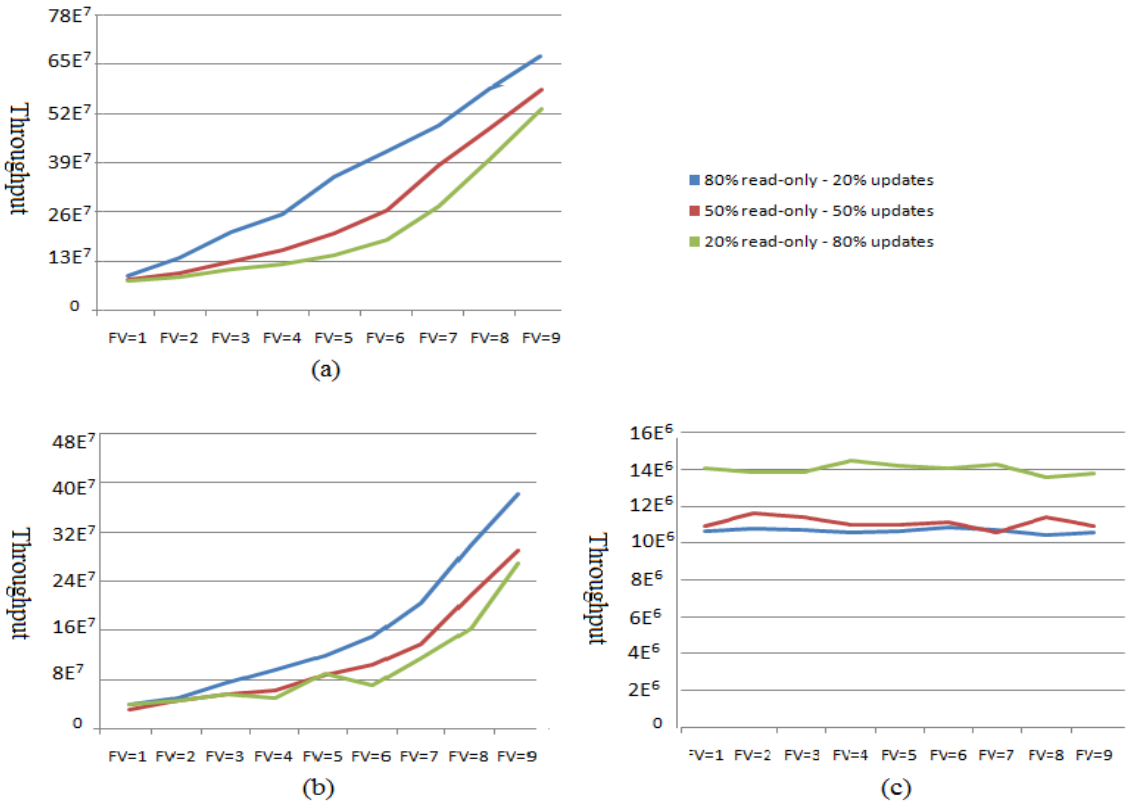

Fig.7: (a) Linked-list Throughput with All FVs. (b) Red-black Tree Throughput with All FVs. (c) Bank Throughput with All FVs.

Furthermore, having all FVs as classes in the learning model will affect the accuracy and increase the learning overhead. Therefore, we need to reduce the number of FVs in the learning model based on the results shown in Fig. 7. We select only three FVs which are 1, 4 and 8 to be the classes in our learning process. Apparently, Fig. 7 shows that the performance of FV= 1 is very close to FV= 2, the FV= 4 is comparable to 3, 5 and 6, and the performance of FV= 8 is akin to FVs 7 and 9.

In Fig. 8, we show the accuracy of learning models. We run LSA with learning models which are HMM and KNN such that the learning models check the prefix of execution and decide the suitable FVs. Then we enforce different FVs in LSA scheduling and we compare their (FVs) throughputs (such as in Fig. 7) with throughputs of LSA using HMM and LSA using KNN. In fact such comparison allows to clearly see the accuracy off the learning models. Actually we run them on the three benchmarks (Linked-list, Red-black Tree and bank) and we run each one with three scenarios (80% reads-only and 20% updates, 50% reads-only and 50% updates and 20% reads-only and 80% updates).

Fig.8(a)showsthethroughputofLSAontheLinked-listbenchmarkwhereFV =9showsthe best of the enforced FVs. LSA using HMM shows high accuracy in some cases and fails with others. Clearly, HMM returns an accurate FV when there is 80% reads-only and 20% updates but it returns inaccurate FVs with the other scenarios (50% reads-only and 50% updates and 20% reads-only and 80% updates). On the other hand, LSA using KNN returns accurate FVs with all three scenarios.

Fig. 8 (b) shows the throughput of LSA on the Red-black benchmark where F V = 9 also shows the best performance (highest throughput) of the enforced FVs. LSA using HMM performs well with the scenario of 80% reads-only and 20% updates but it does not succeed with the other two scenarios. However, LSA using KNN performs well with all three scenarios which indicates the success and accuracy of KNN in the selection of FVs.

Fig. 8 (c) shows the throughput of LSA on the Bank benchmark where all FVs perform almost the same. Thus, the accuracy of HMM and KNN cannot be judged as the selection of any FV does not show any difference on the throughput. However, it is clear that both HMM and KNN do not show negative impact on the performance because they are designed in a proper way that minimize their overhead.

Fig. 9 demonstrates a comparison of the performance of LSA using transactions' timestamps scheduling, LSA with KNN and LSA with HMM. Timestamps scheduling means to schedule transactions based on their arrival times. Using the

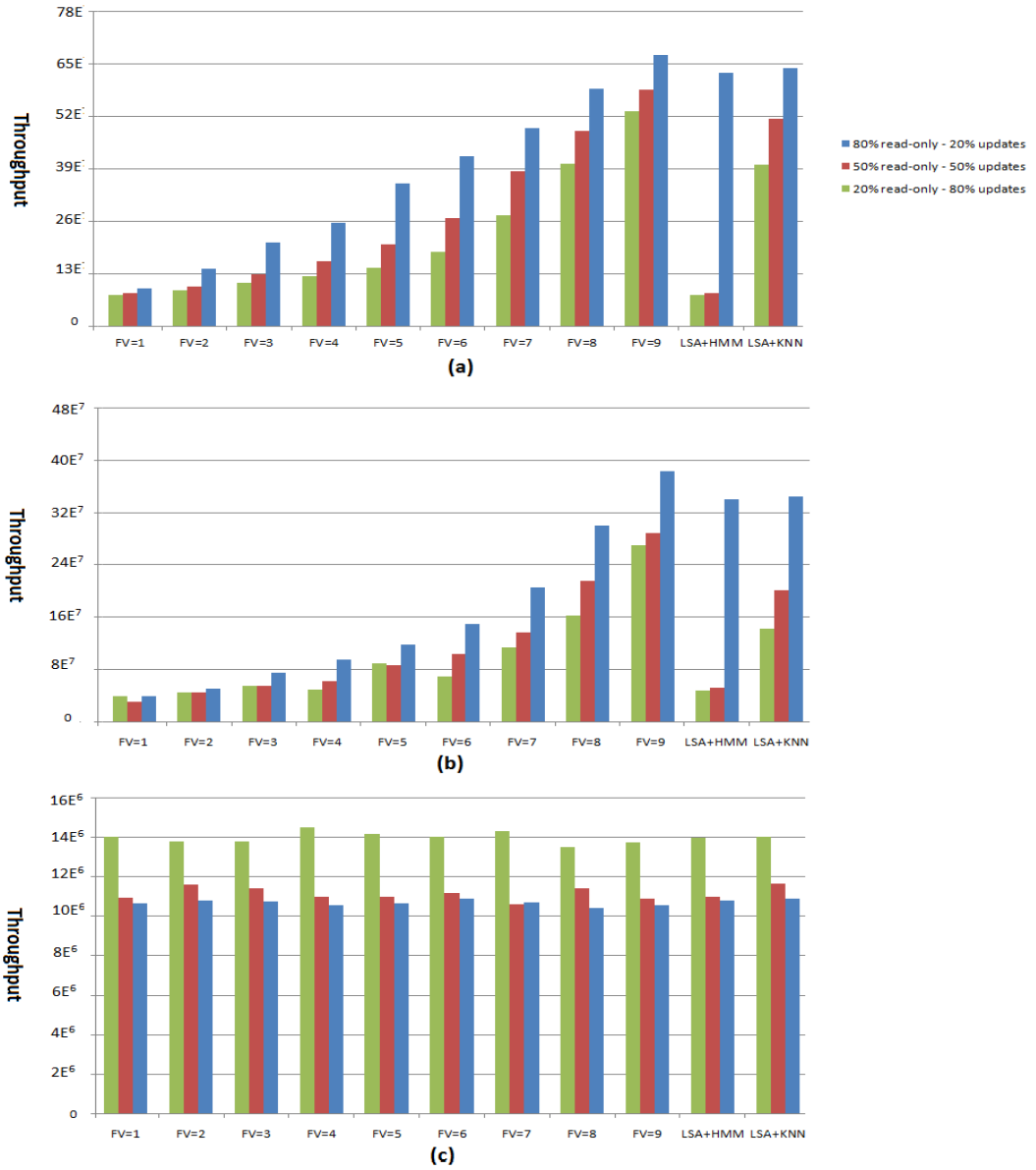

Fig.8: (a) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Linked-list Benchmark. (b) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Red-black Tree Benchmark. (c) The Accuracy of LSA Using KNN and LSA Using HMM Compared to the Enforced FVs Using the Bank Benchmark. (We Measure the Accuracy Based on the Throughput)

Linked-list benchmark, Fig. 9 (a) shows that high percentage of read-only transactions usually results in high throughput as we mentioned before. Fig. 9 (a) proves the advantage of using machine learning for transactions' scheduling. For LSA using KNN, it shows the highest throughput with all scenarios. When we use HMM, it improves the performance of LSA only when the percentage of reads-only is high. On the other hand, the performance of LSA with HMM is negatively affected as the percentage of updates increases. The figure also illustrates the risk of the learning process, as the learning accuracy problems lead to unsuitable scheduling which causes more aborts. This clearly happens with HMM when there are 50% or 80% updates. Thus, KNN obviously is more suitable to LSA scheduling.

The same thing happens with the Red-black Tree benchmark in Fig. 9 (b), where KNN is more efficient than HMM and timestamps scheduling. HMM works well only with a large number of read-only transactions and it fails when we reduce the reads-only.

In Fig. 9 (c), with the Bank benchmark, the duration of update transaction is less than read-only

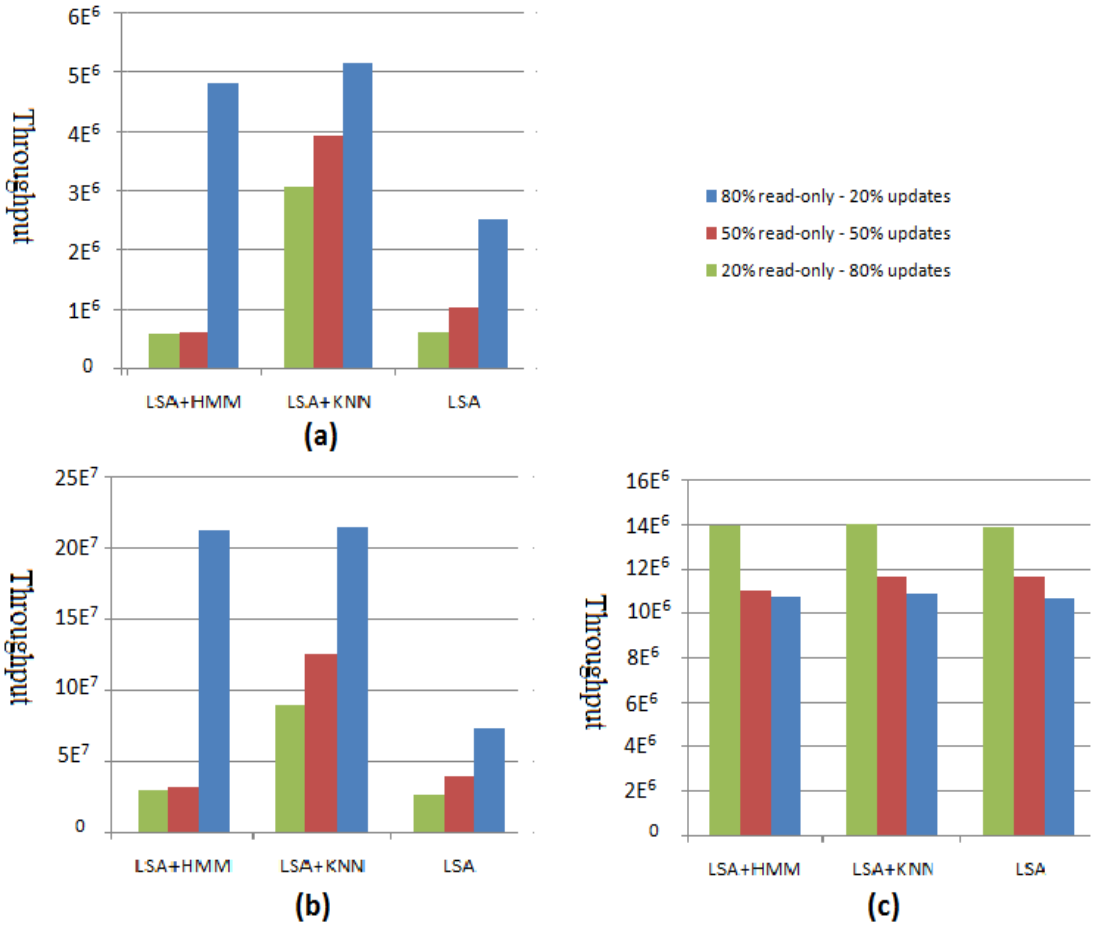

Fig.9: (a) LSA, LSA with KNN and LSA with HMM Throughputs Comparison Using Linked-list Benchmark. (b) LSA, LSA with KNN and LSA with HMM Throughputs Comparison Using Red-black Tree Benchmark. (c) LSA, LSA with KNN and LSA with HMM Throughputs Comparison Using Bank Benchmark.

one. So, the throughput drops when there are more read-only transactions. Furthermore, KNN and HMM do not have significant influence on the performance of Bank benchmark because the durations of transactions in general are very short, and the durations of updates with aborts is almost equal to the durations of reads-only.

Thus, the learning techniques are more helpful if transactions' durations in average are long, and when the costs of updates and aborts are very expensive. Also, the KNN learning model helps to achieve high throughput and better performance of LSA.

On the other hand, one might argue that FV=9 could always be used instead of going through the overhead of the learning model computations. This argument could be based on the premise, that FV=9 shows good throughput in all benchmarks. In fact, this premise can be misleading especially with systems where information has to be up to date most of the times. In such systems, we should allow more updates to commit instantaneously, so that read-only transactions read up to date values. Another reason to show that FV=9 is not always good is that some systems have increasingly high number of reads and writes competing on a limited storage (for example, small number of memory objects or linked-list with small number of nodes). This will result in higher chances of dependencies and conflicts among transactions, which results in more updates having to be aborted.

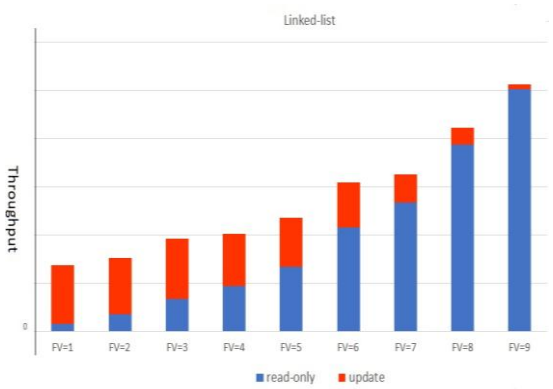

Fig.10: The Number of Read Transactions to the Number of Update Transactions.

Both scenarios are investigated in Fig. 10 and Fig. 11. In Fig. 10, FV=9 shows the highest total throughput. Although the reads are dominating the updates, this shows that huge amount of update transactions are not able to commit, so most of these

reads read outdated values (not stale). This is shown in Fig. 11 where FV=9 results in 81% of update transactions being aborted. In other words, for each committed update transaction there will be around four aborted update transactions. Thus, with FV=9 only about 19% of updates are able to commit, which is not optimal for sensitive systems that requires frequent update.

However, if the algorithm aborts read-only transactions then FV=9 could be the worse, especially with a program of very high dependencies among transactions. In short, FV=9 is a good choice with the algorithms that do not abort read-only transactions, while our model is very useful for other cases. For example, when the chance of aborting read-only and update transactions are equal.

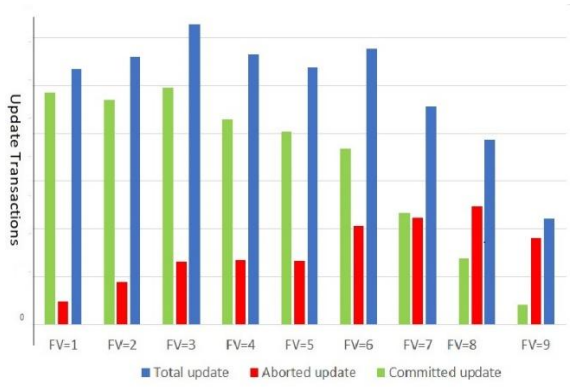

Fig.11. Total Update Transactions vs Committed and Aborted Update Transactions.

Table III shows the learning overhead for KNN and HMM that are combined with LSA using Linked-list, Red-black Tree and Bank benchmarks. The table illustrates the percentages of the execution time of the learning process out of the total execution time of KNN+LSA and HMM+LSA using the three benchmarks. The differences among the three benchmarks depend on how frequent they call the learning function. In our experiment the size of *recordingArray*[] is 20 which means every time we receive 20 transactions we call the learning function and so on. Overall the learning process overhead is reasonable comparing to the improvement on the throughput.

Table III. The Percentages of the Execution Time of the Learning Process (Learning Process Overhead) Out of the Total Execution Time of KNN+LSA and HMM+LSA Using Linked-list, Red-black Tree and Bank Benchmarks.

	KNN	HMM
Linked-list	4.8%	6.38%
Red-black Tree	10.01%	10.94%
Bank	3.28%	3.51%
Average	6.03%	6.95%

VII. CONCLUSION

In conclusion, our scheduling algorithm uses the idea of supervised and unsupervised learning models to improve the performance of transactional memory algorithms. This way of scheduling allows more flexibility to find the suitable scheduling for different problems (Linked-list, Red-black Tree and Bank). Type and duration of transactions extremely impact the performance of transactions execution. In the future, we want to test some other learning techniques with more features and using different algorithms.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation grant 1320835.

REFERENCES

[1] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. Data structures and algorithms, 1983.

[2] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician, 46(3):175–185, 1992.

[3] SS Arya and S Lavanya. A comparative study of machine learning approaches-svm and ls-svm using a web search engine based application. International Journal on Computer Science & Engineering, 4(5), 2012.

[4] HagitAttiya and Eshcar Hillel. Single-version stms can be multi-version permissive. In Distributed Computing and Networking, pages 83–94. Springer, 2011.

[5] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In ACM Sigplan Notices, volume 44, pages 85–96. ACM, 2009.

[6] Pierangelo Di Sanzo, Marco Sannicandro, Bruno Ciciani, and Francesco Quaglia. On exploring markov chains for transaction scheduling optimization in transactional memory. 7th Workshop on the Theory of Transactional Memory (WTTM 2015).

[7] Pascal Felber, Christof Fetzer, and TorvaldRiegel. Dynamic performance tuning of word-based software transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 237–246. ACM, 2008.

[8] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. The Mathematical Intelligencer, 27(2):83–85, 2005.

[9] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures, volume 21. ACM, 1993.

[10] Xuedong D Huang, Yasuo Ariki, and Mervyn A Jack. Hidden Markov models for speech recognition, volume 2004. Edinburgh University Press Edinburgh, 1990.

[11] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[12] Leonid AryehKontorovich, Corinna Cortes, and MehryarMohri. Kernel methods for learning languages. Theoretical Computer Science, 405(3):223–236, 2008.

[13] Priyanka Kumar and Sathya Peri. A timestamp based multi-version stm protocol that satisfies opacity and multi-version permissiveness. arXiv preprint arXiv:1305.6624, 2013.

[14] Karl Ljungkvist, Martin Tillenius, Sverker Holmgren, Martin Karlsson, and Elisabeth Larsson. Early results using hardware transactional memory for high-performance computing applications. In Proc. 3rd Swedish Workshop on Multi-Core Computing, pages 93–97. Go¨teborg, Sweden: Chalmers University of Technology, 2010.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[16] Dmitri Perelman, Rui Fan, and IditKeidar. On maintaining multiple versions in stm. In Proceedings of the 29th

ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pages 16–25. ACM, 2010.

[17] J. Ross Quinlan. Induction of decision trees. Machine learning, 1(1):81–106, 1986.

[18] TorvaldRiegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In Distributed Computing, pages 284–298. Springer, 2006.

[19] EkoSetiawan and AdharulMuttaqin. Implementation of k-nearest neightbors face recognition on low-power processor. TELKOMNIKA (Telecommunication Computing Electronics and Control), 13(3), 2015.

[20] NirShavit and Dan Touitou. Software transactional memory. Distributed Computing, 10(2):99–116, 1997.

[21] Mukul K. Sinha. Nonsensitive data and approximate transactions. IEEE Transactions on Software Engineering, (3):314–322, 1983.

[22] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. ACM Transactions on Architecture and Code Optimization (TACO), 8(4):54, 2012.

[23] Neelamegam, S., and E. Ramaraj. "Classification algorithm in data mining: An overview." International Journal of P2P Network Trends and Technology (IJPTT) 4.8 (2013): 369-374.

[24] Gaur, Manas, ShrutiGoel, and Eshaan Jain. "Comparison between Nearest Neighbours and Bayesian Network for demand forecasting in supply chain management." 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, 2015.