Machine Learning-based Self-adjusting Concurrency in Software Transactional Memory Systems

Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, Francesco Quaglia
DIAG, Sapienza Università di Roma

***Abstract*—One of the problems of Software-Transactional-Memory (STM) systems is the performance degradation that can be experienced when applications run with a non-optimal concurrency level, namely number of concurrent threads. When this level is too high a loss of performance may occur due to excessive data contention and consequent transaction aborts. Conversely, if concurrency is too low, the performance may be penalized due to limitation of both parallelism and exploitation of available resources. In this paper we propose a machine-learning based approach which enables STM systems to predict their performance as a function of the number of concurrent threads in order to dynamically select the optimal concurrency level during the whole lifetime of the application. In our approach, the STM is coupled with a neural network and an on-line control algorithm that activates or deactivates application threads in order to maximize performance via the selection of the most adequate concurrency level, as a function of the current data access profile. A real implementation of our proposal within the TinySTM open-source package and an experimental study relying on the STAMP benchmark suite are also presented. The experimental data confirm how our self-adjusting concurrency scheme constantly provides optimal performance, thus avoiding performance loss phases caused by non-suited selection of the amount of concurrent threads and associated with the above depicted phenomena.**

I. INTRODUCTION

Over the last decade multi-core systems have become mainstream computing architectures so that even desktop and laptop machines are nowadays equipped with multiple processors and/or CPU-cores. Also, systems with up to 16 or 32 CPU-cores can be purchased for a few thousands dollars. This trend has lead to a growing need for the development of applications which can effectively exploit parallelism, thus bringing parallel programming out from the niche of scientific and high-performance computing.

Within this context, Software Transactional Memories (STMs) [1] have emerged as a programming paradigm tailored for the development of concurrent applications. By leveraging on the concept of atomic transactions, historically used in the field of database systems, STMs relieve programmers from the burden of explicitly writing complex, error-prone thread synchronization code. STMs provide a simple and intuitive programming model, where programmers wrap critical-section code within transactions, thus removing the need for using fine-grained lock-based synchronization approaches. Programmers' productivity is therefore improved, while not sacrificing the advantages provided by high parallelism, thus avoiding any loss in performance typically associated with serial execution scenarios, or with cases where an easy to program, coarse-grained locking approach is used.

This work was partially supported by the European Commission through the Cloud-TM project (257784) and the COST Action IC1001 Euro-TM.

Data conflicts are handled within STMs by means of conflict detection and management (CDMAN) algorithms, and most of the literature work made in this field has been aimed at designing increasingly effective CDMAN schemes, thus ultimately aiming at improving the throughput of STM applications. Example solutions can be found in [2], [3], [4], [5], [6].

On the other hand, none of the above approaches has been targeted at directly controlling and optimizing the level of parallelism, which would lead to the identification of suited values for the total amount of threads sustaining the application as a function of the workload profile. Increasing the number of concurrent threads can speed-up the application as more transactions (and/or more non-transactional code blocks) can be processed in parallel. However, increasing the number of concurrent transactions typically causes an increase of the transaction conflict rate. As a consequence, transactions may experience more abort/retry phases, which give rise to an increase of the execution time.

A rule which helps is that, in general, it is not convenient to have more active threads than the available CPU-cores [7], since for fine-grained computations (just like main memory transactions proper of STM systems) this would lead to excessive CPU-dispatching costs/latencies at the operating system level, which might even negatively impact the duration of critical sections accessing shared data. However, this is not sufficient to avoid the loss of performance due to excessive data contention, in particular in systems with a scaled-up amount of CPU-cores. In relation to the latter point, for some tests we carried out using TinySTM [3] and running the STAMP benchmark suite [8] on top of a HP ProLiant server with 16 CPU-cores, the best performance was several times achieved using less than 16 concurrent threads.

Overall, the choice of the well suited degree of concurrency is fundamental in order to obtain adequate trade-offs between parallelism and data conflict. Also, it is an orthogonal problem with respect to CDMAN, and has been shown to be addressable through pro-active transaction scheduling, as in [9] and [10]. Essentially, the approach behind these works aims at reducing the performance degradation due to transaction aborts by avoiding to schedule the execution of transactions whose associated conflict probability is estimated to be high. However, by exclusively considering the transaction rollback probability, several relevant parameters having an impact on the actual transaction wasted time (and hence on the transaction execution latency) are not included, such as the workload profile of the running application as well as the effects due to the underlying hardware (e.g. in relation to the caching hierarchy).

In this paper we present an approach relying on machine learning, which also tackles the aforementioned shortcomings, where we use a neural network [11] to enable the performance prediction of STM applications as a function of the concurrency level (by also indirectly capturing the above mentioned workload profile and hardware effects). The neural network is trained using a data set obtained by profiling the workload generated by the application. Then, at run time, a statistical characterization of the application workload is periodically generated, which is used by a control algorithm as input to the neural network in order to predict the wasted transaction execution time. The prediction is finally exploited by the control algorithm to regulate the concurrency level with the aim at maximizing the application throughput.

To evaluate the effectiveness of our self-adjusting concurrency proposal, we have implemented the whole architecture by leveraging on TinySTM [3], which is a popular open-source STM layer written in C language, and we have performed an extended experimental study by relying on applications selected from the STAMP benchmark suite [8]. In our architecture, we indirectly control the maximum number of concurrent transactions by directly controlling the actual number of active threads in the different phases of the application run. This has been done in order to support a fair comparison with the baseline case where TinySTM applications are run with no self-adjusting concurrency scheme, given that TinySTM only provides support for managing threads (not for selecting the maximum number of concurrent transactions). By the experimental data the overhead introduced by the implemented self-adjusting concurrency functionalities (vs the baseline case) reveals almost negligible, which, together with the effectiveness of the proposed machine-learning based prediction method, allow our solution to provide optimal performance across the whole set of tested workloads.

The reminder of this paper is organized as follows. Related work is discussed in Section II. A recall on neural networks is provided in Section III. The model of target STM applications is presented in Section IV. The architecture of the self-adjusting concurrency proposal and its implementation are described in Section V and Section VI, respectively. The results of the experimental evaluation are provided in Section VII.

II. Related Work

In [12] an analytical modeling approach has been proposed to evaluate the performance of STM applications as a function of the number of concurrent threads and other workload configuration parameters. This kind of approach is targeted at building mathematical tools allowing the analysis of the effects of the contention management scheme on performance. For this reason a detailed knowledge of the specific CDMAN scheme used by the target STM is required, which is instead not required by the approach we are currently proposing.

The work in [13] presents an analytical model taking as input a workload characterization of the application, expressed in terms of transaction profiles, contention probability and consumption of hardware resources. The model predicts the application execution time as function of the number of concurrent threads sustaining the application, however the prediction is a representation of the average system behavior over the whole lifetime of the application. Hence, differently from our proposal, no ability to capture run-time variations (with consequent adaptation of the level of concurrency) is envisaged.

The proposal in [14] is targeted at evaluating scalability aspects of STM systems. It relies on the usage of different types of functions (such as polynomial, rational and logarithmic functions) to approximate the performance of the application when considering different amounts of concurrent threads. The approximation process is based on measuring the speed-up of the application over a set of runs, each one executed with a different number of concurrent threads, and then on calculating the proper function parameters by interpolating the measurements, so as to generate the final function used to predict the speed-up of the application vs the number of threads. Differently from our proposal, a limitation of this approach is due to the fact that the workload profile of the application is not taken into account, hence the prediction may prove unreliable when the profile gets changed wrt the one used during measurement and interpolation phases.

As we hinted before, the issue of performance degradation due to excessive data contention has been addressed using pro-active transaction scheduling, which relies on collecting information related to data contention over the recent past of the application. In [9] a control algorithm dynamically changes the number of threads which can concurrently execute transactions on the basis of the observed transaction conflict rate. It is decreased when the rate exceeds a threshold, while it is incremented when the rate is lower than another threshold. In the approach proposed in [10], incoming transactions are enqueued and sequentialized when an indicator, referred to as *contention intensity*, exceeds a pre-established threshold. The contention intensity is dynamically calculated depending on the number of aborted vs committed transactions. In the proposal presented in [15], a transaction is sequentialized when a potential conflict with other running transactions is predicted. The prediction leverages on the estimation of the expected transaction read-set and write-set (on the basis of the past behavior of the same or other transactions). Actually, the sequentializing mechanism is activated only when the amount of aborted vs committed transactions exceeds a given threshold. Compared to our approach, all the above proposals do not directly estimate the wasted time due to aborted transactions (vs the level of concurrency), while they only indirectly attempt to control the wasted time according to heuristics schemes.

As for machine learning, to the best of our knowledge, it has been used in the context of transactional memories by two works. In [16], machine learning techniques are used to select the best performing CDMAN algorithm. Conversely, in [17], machine learning is used to select the most suitable thread mapping, i.e., the placement of application threads on different CPU-cores, in order to get optimized performance. The goals of both these works are different and orthogonal with respect to our one since we focus on the regulation of the overall concurrency level within the system.

III. Brief Recall on Neural Networks

A Neural Network (NN) is a machine learning method [11] providing the ability to approximate various kinds of functions, including real-valued ones. Inspired to the neural structure of the human brain, an NN consists of a set of interconnected processing elements which cooperate to compute a specific function, so that, provided a given input, the NN can be used to calculate the output of the function. By relying on a learning algorithm, the NN can be trained to approximate an unknown function f exploiting a data set $\{(\mathbf{i}, \mathbf{o})\}$ (training set), which is assumed to be a statistical representation of the function f such that, for each element $(\mathbf{i}, \mathbf{o})$, $\mathbf{o} = f\{\mathbf{i}\} + \delta$, where δ is a random variable (also said *noise*).

IV. Model of the STM Application

We denote with m the actual number of concurrent threads characterizing the execution of the STM application. Threads can be activated and deactivated over the application lifetime so that the value of m can be dynamically changed. The execution flow of each thread is characterized by the interleaving of transactions and non-transactional code blocks (ntc). Any transaction starts with a *begin* operation and ends with a *commit* operation. Also, during the execution of the transaction, the thread can perform both (A) read and write operations on a set S of shared data objects, and (B) code blocks where it does not accesses shared data objects (e.g. it accesses variables within its own stack). Data objects read (written) by a transaction are included in its read-set (write-set). If a data conflict between concurrent transactions occurs, one of the conflicting transactions is aborted and is subsequently re-started. An ntc block is allowed to start right after the thread executes the commit operation of a transaction, and ends right before the execution of the begin operation of the subsequent transaction along the same thread.

V. System Architecture

The self-adjusting concurrency approach we propose leverages on three architectural building blocks, namely:

- A Statistics Collector (SC);
- A Neural Network (NN); and
- A Control Algorithm (CA).

The system architecture is depicted in Figure 1. When a workload sampling interval terminates (hence on a periodic basis), CA gets from SC a set of values characterizing the application workload. In our design, the acquired characterization is assumed to be representative of the workload profile of the application for the near future.

NN is able to predict the average wasted transaction execution time spent by the application, i.e., the average time spent executing aborted transactions, as a function of (A) a given set of values characterizing the workload and (B) a given number of concurrent threads sustaining the application. CA exploits NN to calculate, over a range of values for the number of concurrent threads, the expected wasted time that will characterize the application execution in the near future. Then, on the basis of this outcome, CA determines the number of threads that is expected to provide the best application throughput, and keeps active such a number of threads during the subsequent workload sampling interval.

In what follows, a detailed description of the fuctionalites/features of each component within the system architecture is provided.

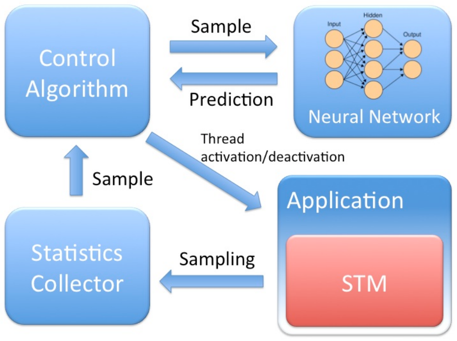

Fig. 1. System architecture.

A. Statistics Collection

At the end of each sampling interval, SC makes an estimation of the below listed statistical parameters:

- the average read-set size (rs_{size});
- the average write-set size (ws_{size});
- the average execution time (t_{time}) for committed transactions;
- the average execution time for ntc blocks (ntc_{time}).

In addition, SC calculates two indexes which provide an estimation of the conflict affinity of transactions upon executing a read or a write operation, respectively. The first one (rw_{aff}) provides an estimation of the probability that an object read by a transaction is also written by other concurrent transactions. The second one (ww_{aff}) provides an estimation of the probability that an object written by a transaction is also written by other concurrent transactions.

In order to determine rw_{aff} and ww_{aff}, SC performs an estimation of the probability distribution of the read operations and of the write operations over the set S of shared data objects. The rw_{aff} index is obtained by calculating the dot product between the distribution of read operations and the distribution of write operations, while the ww_{aff} index can be evaluated by the dot product of the distribution of write operations with itself.

B. Performance Prediction

The rationale for the set of statistics calculated by SC, as presented in the previous paragraph, is that we consider the following function as the fulcrum of performance prediction:

$$w_{time} = f(rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, t_{time}, ntc_{time}, k),$$

where w_{time} is average wasted transaction execution time and k is the number of concurrently running threads. Intuitively, we expected that, keeping fixed the other input parameters, w_{time} increases if one of the following inputs rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, t_{time} and k increases. Conversely, we expect w_{time} to decrease if ntc_{time} gets increased.

The goal of NN is to provide an approximation f_N of the function f. To this purpose, the data set used to train NN consists of a set of samples, each one derived by observing the application during a training time interval, which includes the following quantities (where we use the apex t to indicate that they are related to the training phase):

- the same set of statistics calculated by SC, i.e., rs^t_{size}, ws^t_{size}, t^t_{time}, ntc^t_{time}, rw^t_{aff}, ww^t_{aff};

- the average wasted transaction execution time w_{time}^t;
- the number k^t of active threads.

Hence, a training sample $(\mathbf{i}, \mathbf{o})$ is such that $\mathbf{i} = (rs_{size}^t, ws_{size}^t, t_{time}^t, ntc_{time}^t, rw_{aff}^t, ww_{aff}^t, k^t)$ and $\mathbf{o} = (w_{time}^t)$.

C. Controlling the Concurrency Level

At the end of each sampling interval, CA gets the vector $(rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, r_{time}, ntc_{time})$ from SC. Then, for each i such that $1 \leq i \leq max_{thread}$ (where max_{thread} is the maximum amount of concurrent threads admitted for the application), it generates the vector $\mathbf{v_i} = \{rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, t_{time}, ntc_{time}, i\}$ and predicts $w_{time,i} = f_N(\mathbf{v_i})$ by relying on NN. After, it uses the set of predictions $\{(w_{time,i})\}$ to estimate the number m of concurrent threads which is expected to maximize the application throughput along the subsequent observation period. Specifically, exploiting t_{time} and ntc_{time} as predictions of the average execution time of committed transactions and of ntc blocks, respectively, m is equal to the value of i, with $1 \leq i \leq max_{thread}$, for which

$$\frac{i}{w_{time,i} + t_{time,i} + ntc_{time}} \tag{1}$$

is maximized. Note that $w_{time,i}+t_{time,i}+ntc_{time}$ corresponds to the predicted average execution time between the commit operations of two consecutive transactions along a given thread when there are i active threads. Finally, during the subsequent sampling interval, CA keeps active m threads, deactivating (if active) the remaining $max_{thread} - m$ threads.

VI. Implementation

We have implemented a fully featured Self-Adjusting Concurrency STM (SAC-STM) based on the architecture proposed in the previous sections. The STM layer has been implemented by relying on the release of TinySTM version 1.0 for Unix systems. We used the facilities natively offered by TinySTM to determine w_{time}, t_{time} and ntc_{time}. In addition, we instrumented TinySTM code to gather samples to evaluate rs_{size} and ws_{size}. In order to compute the access distribution of read/write operations, we added a read counter and a write counter for each element of the lock vector. At the end of the commit phase of a successfully committing transaction, the read (write) counter for each lock associated with an item in the read (write) set of the transaction gets incremented. To keep low the overhead associated with the sampling mechanisms, statistics are gathered by a single thread (that we name *master* thread). This choice also permits not to affect system scalability as thread synchronization mechanisms are avoided at all within the added statistics-collection modules. Actually, the master thread is randomly selected among the active threads at the beginning of each sampling interval. A sampling interval terminates after the master thread has committed n subsequent transactions. At the end of each sampling interval, the master thread calculates the aggregated statistics and then, by relying on an implementation we have developed for NN, it calculates the number m of concurrent threads which is expected to maximize the throughput according the approach depicted in Section V-C. Finally, it keeps active m threads (out of the maximum number of max_{thread} threads) during the next sampling interval.

We actually tested two thread activation/deactivation mechanisms. The first one leverages on a shared array with max_{thread} elements. The master thread sets to 1 (0) the elements associated with the threads which have to be deactivated (activated). The slave threads (namely the remaining $max_{thread} - 1$ threads) check their corresponding value before executing a new transaction, by trapping into a busy waiting phase while the value is 1. The second mechanism leverages on a shared array of max_{thread} POSIX semaphores initialized to 0. In this case, the master thread increments (decrements) the semaphores associated with the threads which have to be deactivated (activated), and the slave threads, on check, perform a *wait-for-zero* operation on the associated semaphore. Note that in this case, when a thread is deactivated, it actually sleeps (thus not consuming CPU cycles) until it is reactivated. On the other hand, the two different approaches provide different reactiveness since the usage of semaphores imposes sleep-ready thread transitions at the operating system level. In order to further study the effects of thread wake-up and rescheduling, for the case where we rely on semaphores we have studied two different configurations. The first one is such that the m threads to be maintained active (across the max_{thread} threads) are selected according to a round-robin scheme, while the second configuration is such that always the same set of threads are kept active, except for those that have to be newly activated or deactivated, which might reduce the cost of thread reschedule. We note anyway that the latter configuration is not suitable for all kinds of applications. Specifically, it can not be used for applications that make a pre-partitioning of the work among threads, because, in this case, sleeping threads may prevent the application to fairly make progress for a relatively long time interval. Overall, the different configurations we consider allow us to study differentiated trade-offs involving both performance and applicability aspects.

Finally, our implementation of NN consists of an acyclic feed-forward full connected network [11] that has been coded by leveraging on FANN open-source libraries (version 2.2.0) [18]. NN has an input layer containing seven nodes and an output layer containing a single node, according to the number of input and output parameters of the function f to be estimated.

VII. Experimental Evaluation

In this section we present the results of an experimental study we carried out to evaluate the effectiveness of our proposal. We run applications from the STAMP benchmark suite on top of the above described implementation of SAC-STM, which has been hosted by an HP ProLiant server equipped with two AMD Opteron^{TM}6128 Series Processor, each one having eight hardware cores (for a total of 16 cores), and 32 GB RAM, running a Linux Debian distribution with kernel version 2.7.32-5-amd64. We present the results for three different STAMP applications, Kmeans, Intruder and Genome. These applications span from low to high percentage of time spent executing transactions (vs non-transactional code blocks), and from low to high data contention levels. When running these applications with different workload configuration parameters on top of the native version of TinySTM, the

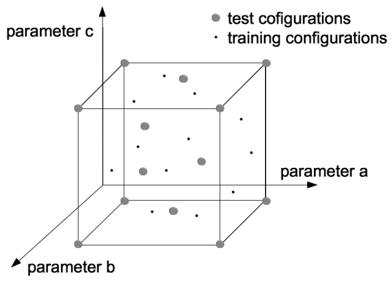

Fig. 2. Example training configurations and test configurations for the case of three workload configuration parameters.

statically configured optimal number of threads remarkably changed for Kmeans (between 4 and 16) and a bit less for Genome (between 4 and 14). Conversely, for Intruder we observed a very low variation (between 4 and 6). Hence, the different variability exhibited by these applications, in terms of optimality of the number of threads when considering the static case, gives rise to a good test-suite for the evaluation of our self-adjusting proposal.

A. Evaluation Methodology

For each test-bed application we performed an off-line training of NN using samples gathered during the execution of a set of runs of the test-bed applications. In order to evaluate the robustness of our proposal, we purposely avoided the generation of training samples that would cover uniformly the sampling space. Conversely, the training runs have been executed by randomly selecting non-equidistant samples for both the values of the workload configuration parameters of the application, each one between its corresponding two extreme values, and the number of concurrent threads. Finally, training samples have been randomly selected across samples gathered during the execution of the runs.

We tested SAC-STM using various workload configurations. Further, to assess its robustness, we also used workload configurations associated with the extreme values of the intervals in which the values of the workload configuration parameters have been selected for the training phase of NN. An example of the type of workload configurations we used in our tests is depicted in Figure 2 for the case of three configuration parameters. The vertices of the cube represent the configurations defined through the extreme values of the intervals. Note that these configurations represent border cases with respect to the configurations used to train NN. Indeed, all the randomly selected configurations used for the training phase are contained within the cube. Since the workload configuration parameters of the application also affect the number of transactions to be executed, in our tests we excluded those configurations for which the number of transactions within the parallel run was so short that it did not allow the completion of at least three sampling intervals.

B. Off-line Training

We trained NN using 800 samples randomly selected over the execution of 64 runs according to the methodology described in the previous section. The number of concurrent threads for each run was randomly selected between 1 and

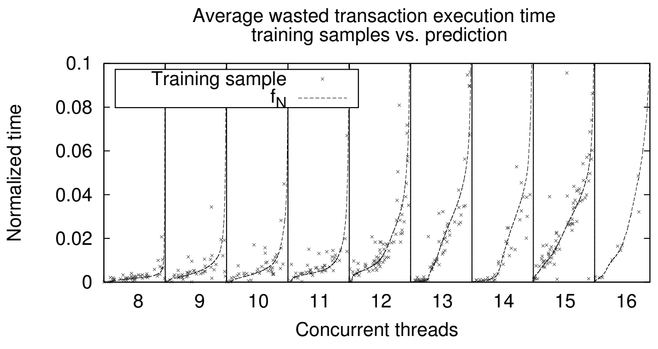

Fig. 3. Average wasted transaction execution time: training set vs. predicted.

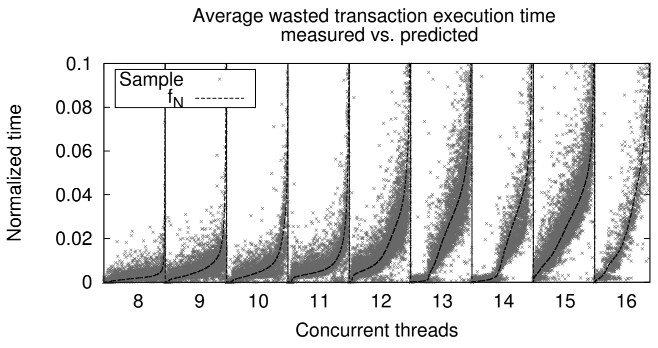

Fig. 4. Average wasted transaction execution time: measured vs. predicted.

32 ([1]). Each sample contains the values calculated over a sampling interval whose duration was determined by the execution of 2000 subsequent committed transactions along any thread. To limit the number of outliers, we discarded (filtered out) samples stemming from sampling intervals in which more than 99% of the transaction runs have been aborted. Note that, when moving towards such an abort probability, the transaction response time grows very fast and exhibits high variability. We assume that in this situation the system throughput is never optimal. Hence, for higher abort probability it suffices that NN approximates the transaction response time by relying on the closest samples which have not been filtered out.

To train NN we used a back-propagation algorithm [19], [20], [21]. We observed that a number of hidden layers equal to one was a good trade-off between prediction accuracy and learning time. In this case, the number of hidden nodes for which NN provided the best approximations was between 4 and 16, depending on the application. Further, we observed that 0.1 and 0.0 were good values for the learning coefficient and the momentum, respectively. In the worst cases, the iterations of the back-propagation algorithm have been no more than 25000, and the algorithm execution time was less

[1]Although, as hinted in the Introduction, it is generally not convenient to use more threads than the available cores, for completeness of the analysis (and for compliance with what done in other studies), we also report some performance data related to the case where max_{thread} is varied up to 32, thus doubling the 16 CPU-cores available on the used hardware platform. This is the reason why we considered up to 32 threads in the off-line learning phase.

than 20 seconds on a desktop machine equipped with an Intel®Core^{TM}2 Duo P8700 and 8 GB RAM. On the other hand, the on-line computation latency by NN was on the order of less than one microsecond.

To provide some graphical details about off-line training, in Figure 3 we plot the dispersion of the training set values of the wasted transaction execution time, namely w^t_{time}, together with the function f_N learned by NN as result of the off-line training process. These data refer to the Intruder benchmark. The plots refer to different values of the number of concurrent threads, that has been varied between 8 and 16. Note that, fixed the number of concurrent threads, the values of f_N depend on other six parameters which have been projected on a two-dimensional space, where the f_N function is plotted according to an increasing ordering of its values. In order to assess the quality of the prediction by NN, in Figure 4 we plot the estimated function f_N and the wasted transaction execution time associated with a larger set of training samples that have not been used for the aforementioned training phase. As we can note, also in cases where very few training samples are used (see, e.g., Figure 3 in correspondence to 16 concurrent threads), NN is able to reliably predict the wasted transaction execution time (see Figure 4 in correspondence to the same number of concurrent threads) thanks to its interpolation/extrapolation ability.

C. Results

For all the tests we present in this section, we plot the application execution time (expressed in sec) achieved with SAC-STM and with the original TinySTM, which we use as a baseline, while varying max_{thread}. We initially consider test cases where max_{thread} is less than or equal to the value 16, which corresponds to the amount of CPU-cores available on the used hardware platform. After (as already hinted, for completeness of the analysis) we present the results for a test where we consider values for max_{thread} up to 32. When considering values of max_{thread} less than or equal to 16, the selected mechanism for managing thread activation/deactivation within SAC-STM is busy-waiting, which for the specific configuration conditions does not give rise to delays in the progress of individual threads (since each thread runs on an exclusively dedicated CPU-core) and avoids sleep-ready thread transitions at the operating system level.

For each test-bed application, we present the results achieved with three different workload configurations. These include the two configurations corresponding to the vertices of the cube (see Section VII-A) where SAC-STM achieved the worst and the best performance with respect to the best case observed when running on top of TinySTM while manually varying the number of threads.

In Figure 5 we present the results for the Intruder benchmark. As we can see, for all the considered configurations, the application execution time achieved with TinySTM decreases when increasing the number of used threads up to 4-6, while for greater values it drastically increases. Conversely, for all the tests, SAC-STM achieves very good results independently of the maximum amount of allowed concurrent threads. In fact, in scenarios where max_{thread} is less than the amount of threads giving rise to the optimum case for TinySTM, the results achieved with SAC-STM and TinySTM are comparable.

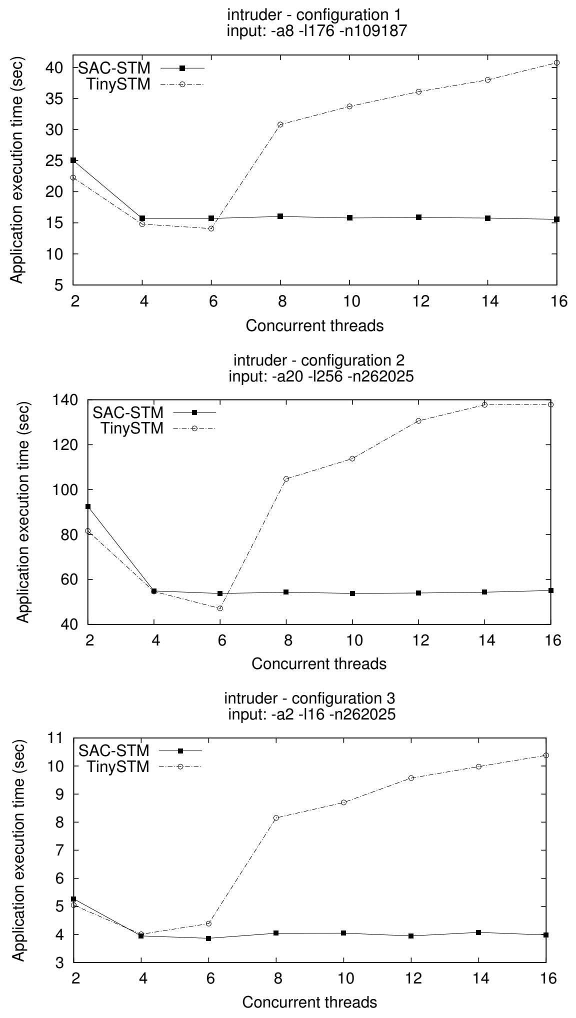

Fig. 5. Application execution time with SAC-STM and TinySTM with Intruder.

With more threads, SAC-STM is able to constantly ensure an application execution time very close to the best one achieved with TinySTM. In particular, also for the two configurations corresponding to the vertices of the cube, i.e., *configuration 2* and *configuration 3*, the performance results are good. In the most adverse case to SAC-STM, which corresponds to *configuration 2*, SAC-STM constantly achieves an execution time no more than 12.5% worse compared to the best case provided by TinySTM, i.e. when it runs with a fixed number of 6 threads. However we note that as soon as 8 or more threads are used by TinySTM, its performance rapidly degrades up to a factor 2.8x, thus exhibiting a clear scalability problem with this workload. This phenomenon is avoided at all by SAC-STM thanks to its proper thread activation/deactivation functionalities, which provide a means to control the negative effects associated with data contention.

The results of the tests with the Genome benchmark are shown in Figure 6. For *configuration 1* and *configuration 2* (the latter is a vertex configuration of the cube), the execution times with SAC-STM and TinySTM are comparable up to 4 threads. With more threads, while the performance with

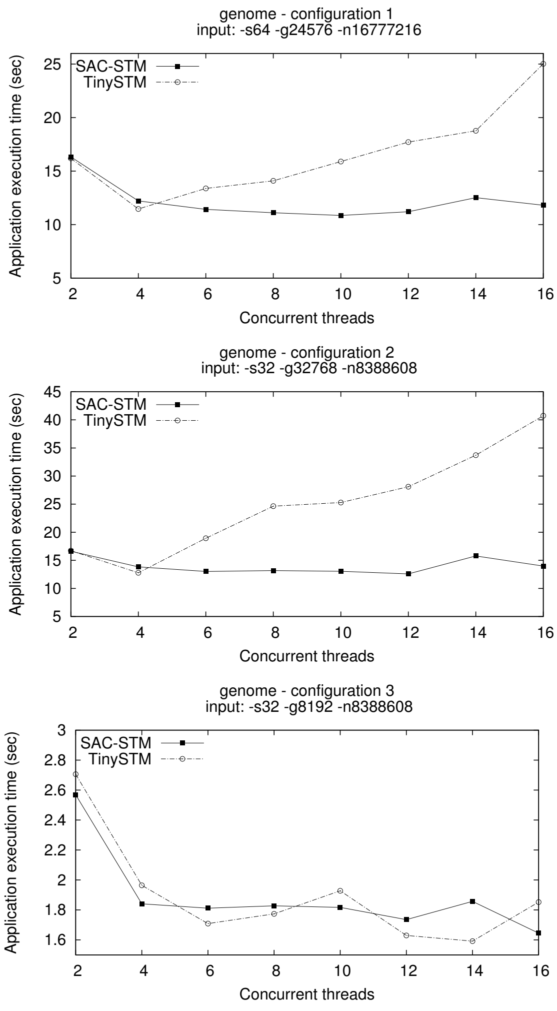

Fig. 6. Application execution time with SAC-STM and TinySTM with Genome.

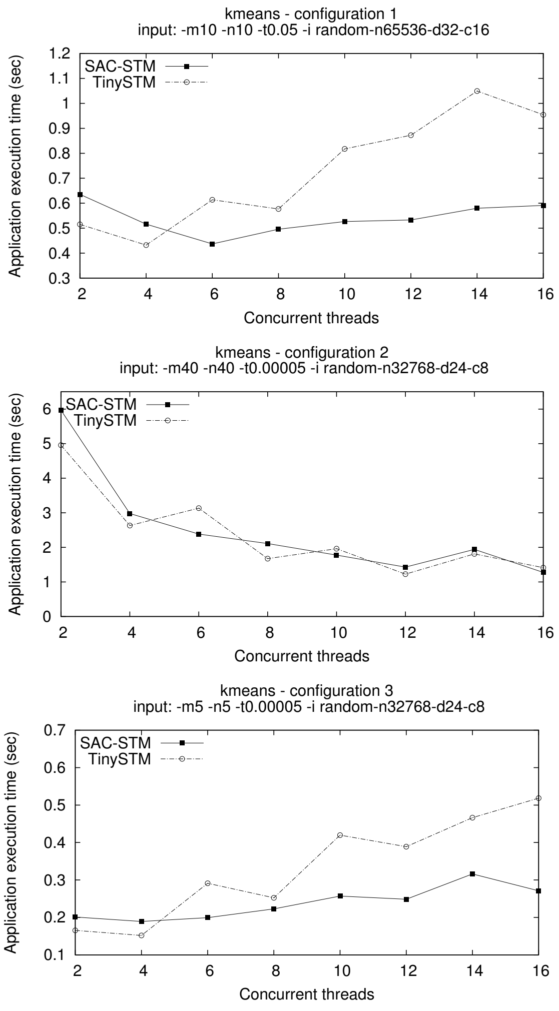

Fig. 7. Application execution time with SAC-STM and TinySTM with Kmeans.

TinySTM degrades, SAC-STM ensures, again independently of the number of available threads, an execution time comparable to, or lower than, the best one provided by TinySTM (i.e. with 4 threads). With *configuration 3* (which is the other vertex configuration of the cube) the best execution time with TinySTM is achieved with 14 threads, after which the performance slightly decreases. With this configuration, the results achieved with SAC-STM are, on average, comparable with those by TinySTM.

Finally, the results plotted in Figure 7 refer to the tests with the Kmeans benchmark. Also in this case, SAC-STM provides performance benefits in all the scenarios when max_{thread} is set to a larger value than the one giving rise to the best case for TinySTM. For *configuration 2*, namely a vertex configuration of the cube, the execution times are comparable while varying the amount of available threads. For the other vertex configuration, namely *configuration 3*, the best execution time achieved by TinySTM (i.e. with 4 threads) is about 22% lower than the execution time achieved with SAC-STM. But with more available threads, SAC-STM constantly achieves a better execution time, hence outlining again how TinySTM may suffer from selection of an oversized degree of concurrency, which is instead not the case for SAC-STM.

In order to show the different effects on performance determined by the specific thread activation/deactivation mechanisms, in Figure 8 we plot the application execution time obtained with SAC-STM in a test with the Intruder benchmark, where we used busy-waiting or semaphores. For the latter case, we also report data related to both round-robin and non-round robin policies for the selection of the threads to be kept active during the subsequent observation period. Additionally, we plot the application execution time obtained with TinySTM. This time we consider values of max_{thread} up to 32. We note that the estimated optimal number of threads by NN in this test was always around 5.

We can see by the plots that differences between the execution times are low when considering up to 6 threads. After, the execution time with TinySTM quickly increases. With busy-waiting the execution time by SAC-STM remains low up to 16 threads, then it quickly increases, as expected by the fact that only 16 CPU-cores are available (thus leading busy-waiting threads to interfere with the advancement of the threads actively supporting the application). With semaphores, the execution time increases with 6 to 10 threads. Then it

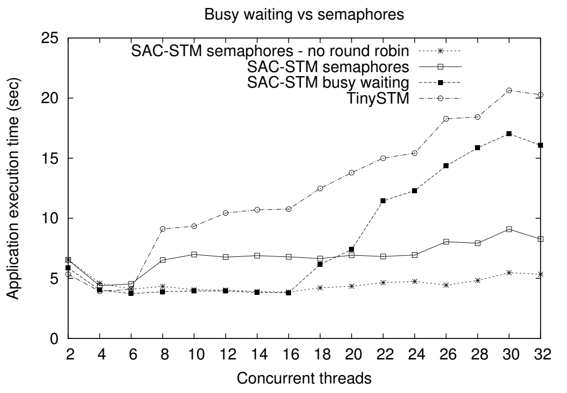

Fig. 8. Application execution time with SAC-STM and TinySTM with Intruder up to 32 threads.

tends to remain quite constant, even after 16 threads. This is due to the fact that, up to 5 threads, none or a few threads, for each sampling interval, are context-switched due to round-robin selection, giving way to other threads which are resumed. With more than 5 threads, and up to 10 threads, the number of threads that are context-switched progressively increases, causing an increase of the execution time. With more than 10 threads, we get that 5 threads, on average, are context-switched and 5 threads are resumed for each sampling period, then the cost of these operations tends to remain constant when further increasing the number of threads of the application. Finally, using semaphores without round-robin selection, the execution time remains quite close to the optimum case independently of the number of threads. In fact, avoiding round-robin selection, running threads tend to remain the same, so reducing costs associated with wait-ready transitions (namely sleeping-thread resume operations) and context-switching.

VIII. Conclusions and Future Work

In this paper we proposed a novel machine learning-based solution addressing the problem of the dynamic selection of the optimal concurrency level in the context of STM systems. We experimented our approach by implementing the system architecture we depicted, thus building an STM which can self-adjust the concurrency level by activating and deactivating concurrent threads on the basis of the profile of the current workload. In our evaluation experiments, we used applications selected from the STAMP benchmark suite. The results we got are very promising, as they shown that, in most of the cases, the performance achieved is, independently of the maximum number of concurrent threads of the application, close to the best case when using a fixed (optimal) number of running threads. In particular, we observed that when an application is executed with an overestimated number of concurrent threads, our self-adjusting STM proves to be able to reduce the concurrency level so to avoid the typical performance degradation experienced with traditional (non-self adjusting) STM systems.

References

[1] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[2] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[3] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[4] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[5] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, New York, NY, USA, 2009. ACM.

[6] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT09*, 2009.

[7] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[8] Chi C. Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008.

[9] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 719–728, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

[11] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[12] Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Performance Evaluation*, 69(5):187 – 205, 2012.

[13] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2010 IEEE International Symposium on*, pages 307 –315, aug. 2010.

[14] Aleksandar Dragojević and Rachid Guerraoui. Predicting the Scalability of an STM: A Pragmatic Approach, 2010.

[15] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 7–16, New York, NY, USA, 2009. ACM.

[16] Qingping Wang, Sameer Kulkarni, John V. Cavazos, and Michael Spear. Towards applying machine learning to adaptive transactional memory. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[17] C.P Ribeiro M. Cole M. Cintra M. Castro, L.F. Wanderley-Goes and J. Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 18th Annual International Conference on High Performance Computing*, 2011.

[18] http://leenissen.dk/fann/wp.

[19] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[20] A.E. Bryson and Y.C. Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Halsted Press book'. Taylor & Francis, 1975.

[21] David E. Rumelhart and Ronald J. Williams Geoffrey E. Hinton. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.