Dynamic Feature Selection for Machine-Learning Based Concurrency Regulation in STM

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

***Abstract*—In this paper we explore machine-learning based approaches for dynamically selecting the well suited amount of concurrent threads in applications relying on Software Transactional Memory (STM). Specifically, we present an approach that dynamically shrinks or enlarges the set of input features to be exploited by the machine-learner. This allows for tuning the concurrency level while also minimizing the overhead for input-features sampling, given that the cardinality of the input-feature set is always tuned to the minimum value that still guarantees reliability of workload characterization. We also present a fully fledged implementation of our proposal within the TinySTM open source framework, and provide the results of an experimental study relying in the STAMP benchmark suite, which show significant reduction of the response time with respect to proposals based on static feature selection.**

I. Introduction

Software Transactional Memory (STM) [1] is recognized as a means for simplifying the development of parallel/concurrent applications by providing a programmer-friendly alternative to traditional lock-based synchronization. On the other hand, one major aspect to cope with is related to the determination of the well suited degree of concurrency (in terms of number of threads), which allows the overlying application to reach optimal speedup values thanks to fruitful parallelism exploitation. Particularly, STM-based applications are prone to thrashing phenomena due to excessive rollbacks of transactions in case the data access pattern tends to exhibit non-negligible conflict among concurrent transactions and the degree of concurrency in the execution is too high. On the other hand, for too low parallelism levels, the achievable speedup may be suboptimal. Recent approaches coping with this issue have been targeted at determining the number of threads which allows for exploiting the available computing resources (namely the available CPU-cores) at the maximum extend still avoiding thrashing phenomena, which leads to optimize the level of parallelism in the execution and the achievable speedup. Along this path we can find solutions ranging from analytical models [2], [4], to heuristic-based schemes [5], to machine learning approaches [8].

In this article we focus on machine learning approaches, which exhibit the advantage of not relying on (strict) assumptions in the workload profile, as instead requested by pure analytical solutions. At the same time they are sufficiently powerful to express/predict the effects of the concurrency degree on performance by accounting for the workload features in an accurate manner.

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

On the other hand, one drawback of machine learning is related to the need for constantly monitoring the set of selected input features to be exploited by the machine learner. This may give rise to non-minimal overhead, especially when considering that STM applications may exhibit fine-grain transactions, natively requiring a (very) reduced amount of CPU cycles for finalizing their task. To cope with this issue, we present an approach where the set of input features exploited by the machine learning based performance model is dynamically shrunk. In other terms, the complexity of both the workload characterization model and the associated performance model is (dynamically) reduced to the minimum that still guarantees reliable performance prediction. This leads to reducing the amount of feature samples to be taken for performance prediction along any wall-clock-time window, hence reducing the actual overhead for performance prediction.

The present work builds on our previous proposal in [8], which presents a Self-Adjusting-Concurrency STM (SAC-STM) architecture, based on neural networks, implemented within TinySTM [9], a popular open-source STM layer written in C language. SAC-STM relies on monitoring a statically identified set of input features which may potentially have an impact on performance (in relation to the degree of concurrency). Hence, it represents a baseline which has been only used as a demonstrator of the viability of machine learning based approaches for concurrency regulation in STM systems. Compared to SAC-STM, the proposal in this paper provides a fully innovative methodology for the dynamic selection of the input features to be monitored and exploited by the performance model. Such a methodology is of general use, hence being not limited to extend the capabilities of SAC-STM. Specifically, it allows for designing/constructing concurrency regulation architectures, suited for integration with generic STM frameworks, which are capable of providing minimal (or very reduced) overhead levels.

We also present a real implementation of our proposal, still integrated with TinySTM, and provide the results of an experimental study based on the STAMP benchmark suite [10]. By the data we show how the proposed approach can reduce the execution time of the benchmark applications, with respect to SAC-STM, by up to 60% when running the applications on top of a 16-core HP Proliant machine

The remainder of this paper is structured as follows. In

Section II we discuss related work. A recap on SAC-STM is provided in Section III. The innovative approach to the dynamic selection of the input features to be sampled within the machine learning scheme is described in Section IV, together with the implementation within TinySTM. Section V reports experimental data.

II. Related Work

We focus in this section on solution explicitly targeted at the identification/selection of the optimal concurrency level in STM systems. The works in [4], [2], [6], [3] present analytical models for the evaluation of the performance of STM applications as a function of the number of concurrent threads and other workload configuration parameters. The approach we study in this paper is related to an orthogonal methodology since is is based on (black-box) machine learning. Also, only a subset of the above works (e.g. [3]) provide models able to predict the performance of STM systems in case of applications exhibiting run-time changes in the execution profile (e.g. in expected number of data objects read/written by transactions). The approach we investigate has this ability.

The proposal in [5] presents a black-box approach based on the hill-climbing scheme in order to increase or decrease the level of concurrency within the system. Particularly, the approach determines whether the trend of increasing/decresing the concurrency level has positive effects of the observed throughput, in which case the trend is maintained. However, differently from our proposal, no direct attempt to capture shifts in the transaction profile, and its effects on performance (depending on the level of parallelism) is done.

Finally, one work having close relations with the present proposal is the one in [8], where the machine learning based SAC-STM architecture has been exploited exactly for the purpose of dynamically regulating concurrency within the system. As hinted, this approach represents a baseline, which does not entail solutions for minimizing the cost of the machine learning based optimization process. Hence, it optimizes the level of concurrency, while still not guaranteeing the optimal performance since the housekeeping overhead is not minimized. In this work we exactly tackle the later aspect, hence the present proposal can be considered as orthogonal and complementary to the one in [8].

III. Overview of SAC-STM

SAC-STM relies on the Neural Network (NN) machine learning method [7], which provides the ability to approximate various kinds of functions, including real-valued ones. 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.

The architectural organization of SAC-STM is depicted in Figure 1. The native application layer embeds the STM layer,

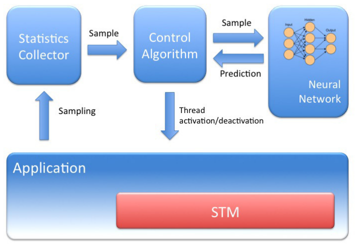

Figure 1. SAC-STM architectural organization

which is in charge of processing STM transactions along any of the active threads. Any transaction starts with a *begin* operation and ends with a *commit* operation. During the execution of the transaction, a thread can perform read/write operations on shared data objects, and can execute code blocks where it does not access shared data objects (e.g. it accesses variables within its own stack). Read (written) shared data objects are included in the transaction read-set (write-set). If a conflict between two concurrent transactions occurs, one of the conflicting transactions is aborted and re-started. The execution flow of each thread is characterized by the interleaving of transactions and non-transactional code (ntc) blocks, as is typical of both real-life applications and STM benchmarks [8].

The following three additional subsystems are included in SAC-STM: A Statistics Collector (SC); A Neural Network (NN); A Control Algorithm (CA). At the end of each sampling interval, whose duration is configurable, CA periodically gets from SC a set of values characterizing the current application workload, which are used as input features to NN. On the basis of the input, NN can provide predictions on the expected value of the transaction wasted time (i.e. the average time spent executing aborted transaction instances), as a function of hypothesized levels of concurrency (number of threads) for the application execution. Particularly, CA queries NN to get these predictions and determines the number of threads that is expected to provide the highest application throughput, and keeps active such a number of threads during the subsequent workload sampling interval.

A. The Selected Input Features

As hinted, the set of input features selected by SAC-STM for characterizing the workload, namely those that are the object of the sampling process, is defined statically. Particularly, the target features have been identified in such a way to cover the set of workload-related parameters that are typically accounted for by performance studies of concurrency control protocols for transactional systems (see, e.g., [11], [12]). In other words, the idea behind the static feature selection process in SAC-STM is to exploit a knowledge base (provided by the literature) related to workload aspects that can, more or less relevantly, impact the performance provided by concurrency control protocols. This approach gave rise to selecting 6 workload-related input features, for which SC provides the corresponding estimated values. We list these features below:

- the average size of the transaction read-set rs_s;
- the average size of the transaction write-set ws_s;
- the average execution time t_t of committed transactions;
- the average execution time ntc_t of ntc code-blocks;
- the read/write affinity rw_a, namely the probability that an object read by a transaction is also written by other transactions;
- the write/write affinity ww_a, namely the probability that an object written by a transaction is also written by other transactions.

B. Performance Prediction and Concurrency Regulation

Performance prediction in SAC-STM is based on the function $w_{time} = f(rs_s, ws_s, rw_a, ww_a, t_t, ntc_t, k)$, where w_{time} is the average transaction wasted time when running with k active threads. The goal of NN is to build a function f_N approximating f. To this end, NN is trained using a set of samples collected by observing the application workload and the transaction wasted time during an initial training phase. Any training sample $(\mathbf{i}, \mathbf{o})$ is such that $\mathbf{i} = (rs_s^t, ws_s^t, rw_a^t, ww_a^t, t_t^t, ntc_t^t, k^t)$ and $\mathbf{o} = (w_{time}^t)$, where we use the superscript t to indicate that they are related to the training phase. At the end of each sampling interval, SC provides to CA the values of rs_s, ws_s, rw_a, ww_a, t_t and ntc_t. Then, NN is exploited to calculate w_{time} for each k such that $1 \leq k \leq max_{thread}$, where max_{thread} is the maximum amount of concurrent threads admitted for executing the application.

Overall, the set $\{(w_{time}, k), 1 \leq k \leq max_{thread}\}$ of predictions is used to estimate the number m of concurrent threads which is expected to maximize the application throughput. Particularly, m is identified as the value of k for which $k/(w_{time} + t_t + ntc_t)$ is maximized, where $(w_{time} + t_t + ntc_t)$ is the predicted average execution time between commit operations of two consecutive transactions along a same thread when there are k active threads.

IV. Dynamic Feature Selection

A. Rationale

Our rationale for the definition of an innovative approach where, depending on the current execution profile of the application, the set of input features to be sampled can be dynamically shrunk, or enlarged towards the maximum cardinality, is based on noting that:

A: some feature values may show small variance during a given time window, and/or

B: some feature values may be statistically correlated (also including the case of negative correlation) to the values of other features during a given time window.

Particularly, we can expect that (significant) variations of w_{time}, if any, do not depend on any feature exhibiting small variance over the current observation window. On the other hand, in case of existence of correlation across a (sub)set of different features, the impact of variations of the values of these features on w_{time} can be expected to be reliably assessed by observing the variation of any individual feature in that (sub)set. In these scenarios, it can be possible to build an estimating function of w_{time} which, compared to f, relies on a reduced number of input parameters. Consequently, NN has to estimate a simpler f_N function. On the other hand, the relevance of excluding specific input features lies on the potential for largely reducing the run-time overhead associated with application sampling, as we shall demonstrate later on in the paper.

For the reference set $\{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$, it comes out natural to think about the following expectations in relation to the correlation of subsets of the input features:

- the size of the transaction read-set/write-set may be correlated to the transaction execution time. In fact, the number of read (write) operations executed by the transaction directly contributes to the actual transaction execution time. If this reveals true, t_t and one feature, selected between rs_s and ws_s, can be excluded;
- read-write and write-write conflict affinities may exhibit correlation since they are both affected by the distribution of the write operations executed by the transactions. If this reveals true, rw_a or ww_a can be excluded.

Overall, we have some expectation for the actual occurrence of the condition expressed by point **B** for at least a subset of the input features. Further, depending on the actual application logic, generic sets of features could result correlated along some time window. Additionally, still depending on the application logic, any of the features in the set $\{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$ may exhibit small variance along some time window, thus being candidate to be excluded from the relevant set of input features.

To determine at what extent such an expectation materializes, and to observe whether the scenarios in points **A** and **B** can anyhow materialize independently of the initial expectation, we have performed an experimental study relying on the complete suite of STM applications specified by the STAMP benchmark [10]. Particularly, we report in Table I data related to the observed correlation among the different features. All data refer to serial executions of the applications, which have been carried out on a single core of a 16-core HP ProLiant NUMA machine equipped with two 2GHz AMD Opteron 6128 processors and 64GB of RAM. This same platform has been exploited for experiments whose outcomes are reported in the remainder of the paper.

We note that serial execution is adequate for the purpose of this specific experimentation since it is only tailored to determine workload features that are essentially independent of the degree of concurrency in the execution. Specifically, given that correlation and variance are computed over feature-samples, each one representing an average value (over a set of individual samples taken along one observation window entailing 4000 transactions in this experiment), for the only parameters that can be potentially affected by hardware contention (e.g. bus-contention) in case of real parallelization, namely t_t and ntc_t, the corresponding spikes (if any) would be made relatively irrelevant by the aggregation of the individual samples within the window related average.

The data confirm that the rationale behind our proposal can find justification in the actual behavior of STM applications, when considering the STAMP suite as a reliable representation of typical STM applications' dynamics. In fact, by Table I, we can observe that the correlation between rw_a and ww_a is higher than 0.8 for 4 applications (out of the 8 belonging to the STAMP suite), the correlation between rs_s and t_t is higher than 0.8 for 3 (out of 8) applications, and the correlation between ws_s and t_t is higher than 0.8 for 2 (out of 8) applications. Further, although not explicitly reported in tabular form for space constraints, we observed very reduced variance for rw_a and/or ww_a for many of the applications, and reduced variance for rs_s and/or ws_s in a few cases.

B. Pragmatic Relevance: Run-time Sampling Costs

The pragmatic relevance of an approach where the sampled input features to be provided to the NN gets shrunk to a minimal set, which is anyhow sufficient to capture the workload characteristics, is clearly related to the possibility to reduce the sampling overhead. An alternative approach to reducing such an overhead would be to make an individual thread (over a set of m concurrent threads) to take samples and to provide statistical data while the application is running. This would delay the critical path execution of 1 out of m threads. However, this approach exhibits two drawbacks making it unsuitable for generic settings:

- The production frequency for the samples gets reduced. Hence, catching any variation in the execution profile of the application may occur untimely.
- STM applications may devote specific threads to run specific transactions (e.g., for locality along the thread execution and cache efficiency improvement [14]). Hence, taking samples along a single thread does not provide a complete picture of the application workload, even in case the sampling thread is dynamically changed over time (e.g. in round-robin fashion).

Further, significant overhead reduction might be achieved via the above approach only for values of m which are larger than the optimal degree of concurrency for the specific application. Overall, the typical scenario for reliable sampling and workload characterization (and timely determination of shifts in the workload) consists of taking samples for evaluating the input features to be provided to NN along the execution of all the active concurrent threads.

For this reference scenario, we have experimentally evaluated the sampling overhead for STAMP applications while varying (a) the number of concurrent threads (between 1 and 16), and (b) the set of selected input features. The overhead value has been computed as the percentage of additional time required to complete the execution of the benchmark application in case sampling is activated, compared to the time required for executing the application in case sampling is not activated. The platform used for the experiments is the same 16-core HP ProLiant machine exploited in the study presented in Section IV-A. In Figure 2 we report the overhead values for the case of the **intruder** application. For the other STAMP applications we got results with similar trends, hence for conciseness they are not explicitly reported.

One observation we can make when analyzing the results is that, when considering the case of the maximum set of sampled input features, the overhead tends to scale down while the number of concurrent threads gets increased. However, the most significant reduction of the overhead is observed exactly for the cases where the set of input features for which sampling is active gets shrunk. Particularly, when shrinking the monitored features from 6 to 4 or 2, we get

ssca2						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,259	1		-	-	-
rs_s	-0,166	0,190	1	-	-	-
ws_s	-0,166	0,190	1	1	-	-
rw_a	-0,024	-0,638	-0,136	-0,136	1	-
ww_a	-0,001	-0,629	-0,210	-0,210	0,992	1

intruder						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,781	1	-	-	-	-
rs_s	0,914	0,940	1	-	-	-
ws_s	0,577	0,924	0,848	1	-	-
rw_a	0,516	-0,377	-0,540	-0,330	1	-
ww_a	0,023	-0,350	-0,269	-0,559	0,322	1

genome						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,012	1	-	-	-	-
rs_s	0,352	0,902	1	-	-	-
ws_s	-0,742	0,492	0,158	1	-	-
rw_a	-0,584	-0,202	-0,397	-0,422	1	-
ww_a	0,040	-0,027	-0,009	-0,049	0,064	1

kmeans						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,141	1	-	-	-	-
rs_s	0,434	0,194	1	-	-	-
ws_s	-0,524	0,106	0,481	1	-	-
rw_a	-0,245	-0,729	0,072	0,177	1	-
ww_a	-0,072	-0,723	0,090	0,008	0,968	1

yada						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,705	1	-	-	-	-
rs_s	0,860	0,619	1	-	-	-
ws_s	0,828	0,617	0,946	1	-	-
rw_a	-0,417	-0,183	-0,508	-0,552	1	-
ww_a	-0,400	-0,173	-0,491	-0,542	0,999	1

vacation						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,989	1	-	-	-	-
rs_s	0,507	0,520	1	-	-	-
ws_s	0,345	0,315	-0,487	1	-	-
rw_a	-0,167	-0,179	0,811	0,657	1	-
ww_a	-0,572	-0,535	0,262	-0,954	-0,483	1

labyrinth						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,993	1	-	-	-	-
rs_s	0,992	0,991	1	-	-	-
ws_s	0,992	0,992	0,999	1	-	-
rw_a	-0,521	-0,500	-0,495	-0,492	1	-
ww_a	-0,332	-0,277	-0,273	-0,267	0,714	1

bayes						
	t_t	ntc_t	rs_s	ws_s	rw_a	ww_a
t_t	1	-	-	-	-	-
ntc_t	0,141	1	-	-	-	-
rs_s	0,434	0,194	1	-	-	-
ws_s	-0,524	0,106	0,481	1	-	-
rw_a	-0,245	-0,729	0,072	0,177	1	-
ww_a	-0,072	-0,723	0,090	0,007	0,968	1

Table I
INPUT FEATURES CORRELATION FOR STAMP APPLICATIONS

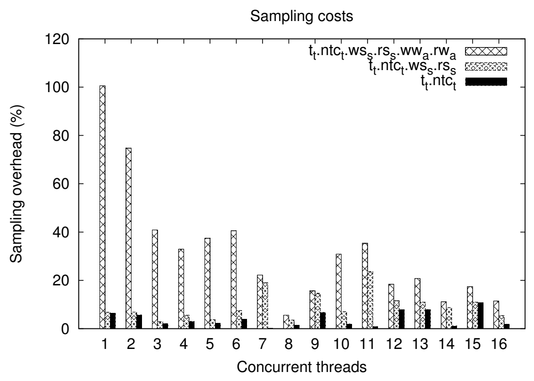

Figure 2. Sampling overhead for the **Intruder** benchmark while varying the set of sampled features

up to 90% reduction of the overhead for smaller numbers of concurrent threads (namely up to 6). This is highly significative when considering that the optimal degree of concurrency for **intruder** has been shown to be around 5/6 when running on the same hardware platform used in this study [8]. In other words, the optimal parallelism degree is achieved for a number of concurrent threads that does not allow to afford the overhead due to sampling in case no optimized scheme for shrinking the set of features to be sampled is provided, which is exactly the target of this paper.

C. Shrinking vs Enlarging the Feature Set

As pointed out in previous sections, shrinking the set of features to be provided in input to the neural network based performance model can rely on run-time analysis of variance and correlation. However, the application execution profile may vary over time such in a way that excluded features become again relevant. As an example, two generic features x and y, which exhibited correlation in the past, may successively start to behave in an uncorrelated manner. Hence, the excluded feature (x or y), if any, should be re-included within the input set, since both of them are again relevant for reliably characterizing the workload.

Detecting this type of scenarios, in order to support the dynamic enlarging of the feature-set cannot be based on run-time input features analysis (e.g. analysis of the correlation), since the feature that was excluded from the input set (for overhead reductio purposes) has been no more sampled. Hence, no fresh information for that feature is available to detect whether variance and/or correlation with other features have changed.

To overcome this problem, our proposal relies on evaluating whether the current wasted-time prediction by NN is of good quality or not (compared to the real one observed at run-time during the successive observation window). In case the quality is detected to be low, the input feature set can be enlarged towards the maximum in order to improve again workload characterization. In other words, low quality prediction by NN is imputable in our approach to the reliance on an input feature-set currently expressing a wrong/not-complete characterization of the workload.

The index we have selected for determining the quality of the prediction is the *weighted root mean square error* (WRMS) of the NN wasted time prediction vs the corresponding real (measured) value. To provide quantitative data showing that WRMS can be considered as a reliable metric, we have performed additional experiments where the effects of concurrency regulation performed by the original version of SAC-STM have been compared with the observed values of WRMS. This experimentation has been carried out by varying both the number of hidden nodes within the NN used by SAC-STM and the number of iterations of the used NN training algorithm. Variations of these parameters allowed us to generate differentiated configurations where the NN may exhibit differentiated prediction qualities.

The results by this experimentation are reported in Figure 3, still for the case of the **intruder** benchmark application. However, also in this case, the data obtained with different benchmarks show very similar trends. From the results, we see how, when the execution time achieved by regulating concurrency with SAC-STM is reduced, the corresponding values of WRMS look very reduced. This tendency is noted independently of the amount of hidden nodes, as soon as at least a minimum amount of iterations of the learning algorithm are carried out. Also, the value of WRMS tends to decrease vs the number of iterations of the learning algorithm. An exception is noted for the case of 32 hidden-nodes, where some spikes are observed for both the benchmark execution time and WRMS, in correspondence to some non-minimal values for the number of iterations. This may be attributed to over-fitting phenomena that may arise when the number of hidden nodes in NN is excessively large. On the other hand, with too low values of the number of hidden-nodes, such as with 4 hidden nodes, the value of WRMS tends to decrease slowly, which leads concurrency regulation to become less effective in reducing the actual execution time for the benchmark. However, the data show that for configurations with reasonable amounts of hidden nodes, the reliability of WRMS as the means for expressing the relation between the quality of the waste of time prediction by NN and the final performance achieved while regulating concurrency on the basis of that prediction is actually assessed.

D. The Actual Dynamic Feature Selection Architecture

To support dynamic selection of relevant features to be sampled and exploited for concurrency regulation, we need to rely on a set of NN instances (not a unique instance as instead it occurs in SAC-STM), each one able to manage different feature-sets and properly trained on that set. These NN instances can be trained in parallel during the early phase of application processing. Then a so called Parameter-Scaling-Algorithm (PSA), implemented within an additional module integrated in SAC-STM, can be exploited for dynamically scaling-up/down the set of features (also referred to as parameters in the final architecture) to be taken into

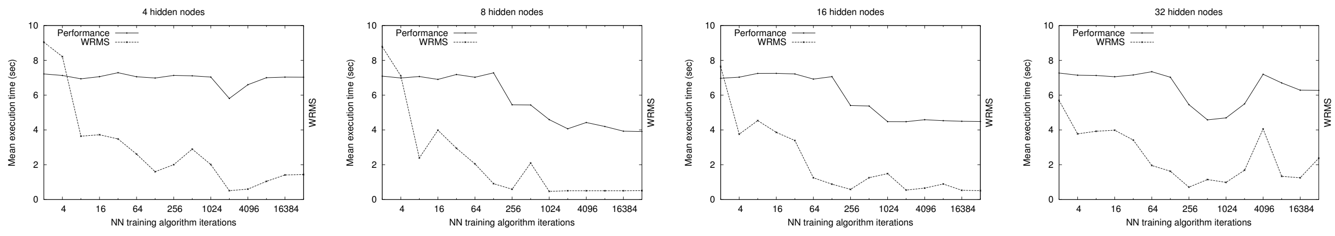

Figure 3. Relation between the final achieved performance and WRMS for **intruder**

account for concurrency regulation along the sub-sequent execution window. Thus PSA is aimed at determining the NN instance to be used in relation to the selected parameters' set. The schematization of the architecture entailing dynamic feature selection capabilities is shown in Figure 4.

To select the best suited NN instance, and hence the subset of features that can be considered as reliably representative of the workload actual behavior, PSA performs the following tasks: (1) It periodically (e.g. at the end of the observation window) evaluates the quality of the prediction by the currently in use instance of NN (representative of the currently in use set of features) via estimation of WRMS; (2) It periodically analyzes the statistics related to the currently monitored features, to determine variance and correlation.

If the calculation of WRMS in point 2 gives rise to a value exceeding a specific threshold, then PSA enlarges the set of input features, to be exploited for concurrency regulation in the subsequent observation window, to the maximum set formed by the 6 features originally used in SAC-STM, namely $maxSet = \{rs_s, ws_s, rw_a, ww_a, t_t, ntc_t\}$. It then issues configuration commands to SC, CA and NN in order to trigger their internal reconfiguration, leading to work with $maxSet$. This means that any query from CA while performing concurrency regulation during the subsequent observation window needs to be answered by relying on the NN instance trained over $maxSet$.

On the other hand, in case the WRMS value computed in point 1 does not exceed the threshold value, the analysis in point 2 is exploited to determine whether the currently in use set of features can be shrunk (and hence to determine whether a scale-down of the set of sampled parameters can be actuated). Particularly, if the variance observed for a given feature is lower than a given threshold, the feature is discarded from the relevant feature-set to be exploited in the next observation window. Then for each couple of not yet discarded features, PSA calculates their correlation and, if another threshold is exceeded, one of them is discarded too. The non-discarded features form the optimized (shrunk) set of parameters to be exploited for concurrency regulation in the subsequent observation window, which we refer to as $minSet$. Then, similarly to what done before, PSA issues configuration commands to SC, CA and NN in order to trigger them for operating with $minSet$.

The values that have been used for configuring threshold parameters and the length of the observation window for the actual experimentation of the final architecture, whose outcomes are reported in the next section, have been selected on the basis of empirical observations. Procedures for automatizing the configuration are planned as future work along this same research path. Finally, in the new architecture in Figure 4, all the tasks associated with concurrency control regulation, which are performed by the modified versions of SC, CA and NN, and by the added PSA module, are carried out off the application critical path. Specifically, they are executed along different, low priority threads, which spend most of their time in the waiting state. Hence, intrusiveness of these threads is extremely limited, as we will also show in Section V via experimental data.

V. Experimental Evaluation

In this section we present the results of an experimental study aimed at evaluating the effectiveness of our proposal. The provided data are related to experiments carried out by still running applications from the STAMP benchmark suite on top of the 16-core HP ProLiant machine that has been exploited for the previous reported experiments. As for STAMP, we selected three applications, namely **intruder**, **ssca2** and **vacation**, since they exhibit quite different execution profiles, hence being representative test cases. Further, we also provide experimental data in relation to a modified version of **vacation**, properly configured to stress (and thus further evaluate) the innovative capabilities by the architecture deriving from our proposal. In Figure 5 we list the statically configured values for the platform parameters, which have been used for the experiments.

In Figure 7 we show the results achieved with the **intruder** benchmark. In particular, we report the benchmark execution time while varying the number of CPU-cores allowed to be used for application execution. This is reflected into a maximum value for the number of threads running the application. In fact, in our study we adhere to the common practice of avoiding the usage of more application threads than the available CPU-cores, which is done in order to avoid suboptimal execution scenarios for STM systems characterized by excessive context-switch overhead [13]. We note that the original version of TinySTM always exploits all the allowed to be used CPU-cores, since it does not entail any concurrency regulation scheme. On the other hand, both SAC-STM (which is taken as a reference together with TinySTM) and the new architecture we have provided, which we refer to as Dynamic-Feature-Selection STM (DFS-STM)

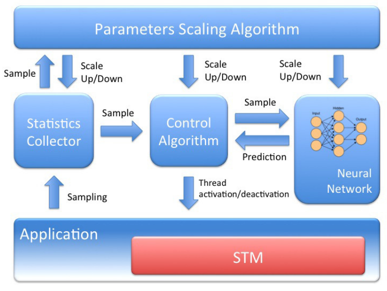

Figure 4. Extended SAC-STM architecture

variance thresholds				
	ssca2	**intruder**	**vacation**	**mod. vacation**
rs_s	4	50	600	340
ws_s	2	3	16	3
rw_a	$5 \cdot 10^{-5}$	0,018	$6 \cdot 10^{-6}$	10^{-4}
ww_a	$5 \cdot 10^{-5}$	$2,5 \cdot 10^{-5}$	$1,2 \cdot 10^{-4}$	10^{-4}
correlation thresholds				
	ssca2	**intruder**	**vacation**	**mod. vacation**
all param.	0,85	0,85	0,85	0,85
variance and correlation analysis window				
	ssca2	**intruder**	**vacation**	**mod. vacation**
#transactions	$4 \cdot 10^5$	$4 \cdot 10^5$	$2 \cdot 10^5$	$4 \cdot 10^5$
concurrency regulation interval				
	ssca2	**intruder**	**vacation**	**mod. vacation**
#transactions	4000	4000	4000	4000

Figure 5. Parameters configuration for the performance tests

	ssca2	**intruder**
DFS-STM	25%	18%
TinySTM	25%	09%
	genome	**kmeans**
DFS-STM	54%	23%
TinySTM	32%	18%
	yada	**vacation**
DFS-STM	23%	40%
TinySTM	19%	7%
	labyrinth	**bayes**
DFS-STM	32%	51%
TinySTM	23%	33%

Figure 6. Minimum achieved percentage of ideal speedup

in the rest of this study, actuate concurrency regulation. Hence, they both lead the application to use a variable amount of threads over time, which ranges from 1 to the maximum value admitted for the specific experimentation point. By the data, the TinySTM curve shows that the benchmark reaches its minimum execution time with static concurrency level set to 5. Beyond this value, data contention brings the application to pay large penalties caused by excessive transaction rollback. Thus, the performance delivered by TinySTM rapidly decreases when running the application assuming more than 5 CPU-cores. Conversely, SAC-STM and DFS-STM provide the same performance achievable with the optimal degree of concurrency for any value of the maximum number of threads in the interval [5-16]. Hence, they correctly regulate concurrency to the optimal level, even when more CPU-cores are available. However, by dynamically shrinking the set of input features to be sampled, DFS-STM allows up to 30% reduction of the benchmark execution time. Hence, it reveals effective in significantly reducing the overhead associated with the static feature-selection approach used by SAC-STM.

The performance data for **ssca2**, reported in Figure 8, look somehow different. Particularly, the TinySTM curve reveals that no thrashing occurs, even when running the application by relying on all the 16 available CPU-cores. This means that, for this benchmark, concurrency regulation cannot be expected to improve performance significantly. However, the results show SAC-STM pays a relevant sampling cost (with no particular revenue from the concurrency regulation process, as hinted above), which leads it to deliver performance from 87% to 60% worse than the one delivered by TinySTM, depending on the maximum allowed number of concurrent threads. Such an overhead is fully removed by DFS-STM, which allows delivering the same identical performance as TinySTM (still with no advantage from concurrency regulation, for the reasons explained above). We note that the overhead reduction, beyond indicating the effectiveness of dynamically shrinking the set of features to be sampled, also indicates null intrusiveness of the additional tasks performed by DFS-STM, such as the execution of PSA.

Figure 9 shows the results for the standard version of **vacation**. Also in this case we reach the optimum performance with 5 threads. Beyond this value, TinySTM exhibits rapidly decreasing performance, just for the reasons explained above. Again, SAC-STM and DFS-STM allow regulating the concurrency level to the best suited value. However, SAC-STM shows significant overhead. Hence, DFS-STM reduces the execution time by about 30%.

All the benchmark configurations that have been considered so far are characterized by phase-based execution profiles, with very few changes along the executon. In order to evaluate the DFS-STM with highly dynamic workloads, the modified version of **vacation** has been exploited. Essentially, **vacation** emulates a travel reservation system, where customers can reserve flights, rooms and cars. The fraction of transactions accessing each one of the three types of items is fixed over time. This is representative of scenarios where the popularity of the different types of items does not change over time. We modified this feature in order to emulate scenarios where the item popularity can show significant changes according to a periodic basis. We note that this kind of scenarios are prone to take place in relation to real-life events (e.g. associated with relevant promotional sales or new product launches). In the modified version of **vacation**, the fractions of transactions accessing the three types of items periodically changes. Specifically, the fraction of transactions accessing car-items changes over time according to the curve depicted in Figure 10. The remaining fraction is equally split into transactions accessing flight and room items.

For this workload, we show in Figure 10 how the number of input features, selected by DFS-STM as relevant, changes over time. These results refer to an execution where we allow a maximum number of concurrent threads equal to 8. We note that, whenever the mix of transactions remains quite constant over time (e.g. up to 17 seconds of the execution, or in the interval between 22 and 27 seconds of the execution), only two parameters (specifically t_t and ntc_t) are selected. Conversely, whenever the mix of transactions rapidly changes (e.g. in the interval between 17 and 22 seconds of the execution, or between 27 and 32 seconds), which leads to increase the variance and/or un-correlation of some workload features, the number of parameters grows to 4 (specifically including t_t, ntc_t, ws_s, rs_s). The throughput achieved with both SAC-STM and DFS-STM is shown on the right of Figure 10. Also in this case DFS-STM achieves a remarkable performance improvement with respect to SAC-STM. For completeness, the benchmark execution time

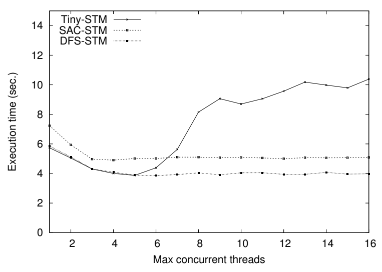
Figure 7. Results for **intruder**

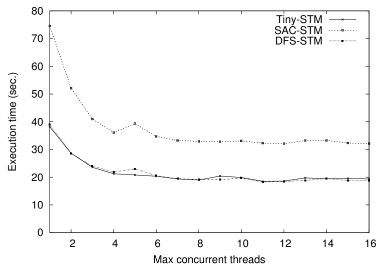
Figure 8. Results for **scca2**

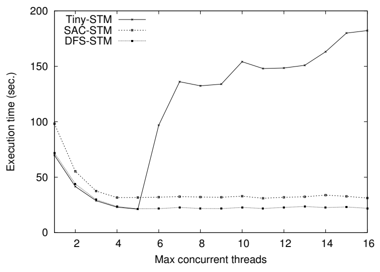
Figure 9. Results for **vacation**

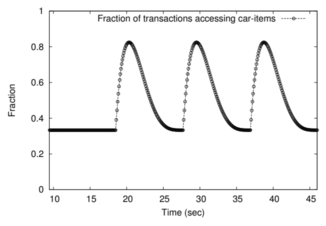
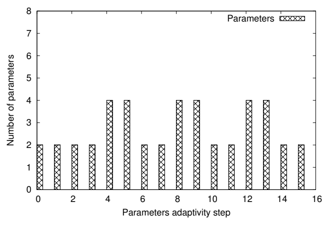
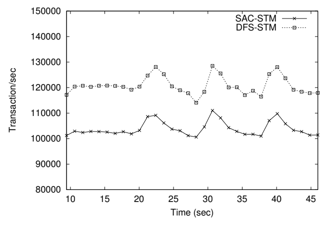
Figure 10. Parameters and throughput variation over time for the modified **vacation** benchmark

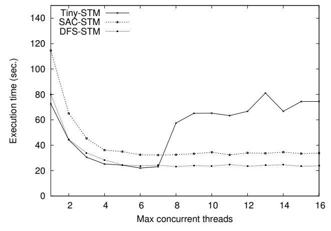
Figure 11. Execution time for the modified **vacation** benchmark

while varying the maximum number of concurrent threads for the case of the modified **vacation** benchmark is depicted in Figure 11, which again shows the relevant gain that can be achieved by DFS-STM over SCA-STM thanks to the reduction of the overhead for supporting concurrency regulation. Finally, in Figure 6 we report the minimum percentage of the ideal speedup (over serial execution of the same application on a single CPU-core) which is achieved by DFS-STM and by TinySTM when considering variations of the number of CPU-cores between 1 and 8. For all the applications of the STAMP suite DFS-STM generally guarantees much higher percentage values of the ideal speedup, which again indicates its ability to efficiently control the parallelism degree by both avoiding thrashing phenomena and inducing very reduced feature sampling overhead.

VI. Summary

We presented an innovative approach for dynamically selecting the input features to be exploited by machine learning based performance models aimed at supporting concurrency regulation in STM systems. The approach relies on runtime analysis of variance and correlation of workload characterization parameters, and on feedback control on the quality of performance prediction achieved with shrunk sets of features. The final target is the reduction of the overhead for sampling the features. A real implementation of the proposal within the open source TinySTM framework has been presented and evaluated.

References

[1] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[2] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Proc. of Int. Symp. on Model. Analysis and Simul. of Comp. and Telecomm. Systems*, pages 307 –315, 2010.

[3] Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Regulating concurrency in software transactional memory: An effective model-based approach. In *Proc. of the 7th Int. Conf. on Self-Adaptive and Self-Organizing Systems*, 2013.

[4] 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. *Perf. Eval.*, 69(5):187 – 205, 2012.

[5] Diego Didona, Pascal Felber, Diego Harmanci, Paolo Romano, and Joerg Schenker. Identifying the optimal level of parallelism in transactional memory systems. In *Proceedings of the International Conference on Networked Systems*, NETYS. Springer, 2013.

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

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

[8] Diego Rughetti, Pierangelo di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proc. of Int. Symp. on Modeling, Analysis and Simulation of Comp. and Telecomm. Systems*, pages 278–285, 2012.

[9] 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, 2008.

[10] 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, 2008.

[11] Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. On the analytical modeling of database concurrency control. *J. ACM*, 40(4):831–872, 1993.

[12] Pierangelo di Sanzo, Bruno Ciciani, Francesco Quaglia, and Paolo Romano. A performance model of multi-version concurrency control. In *Proc. of Int. Symp. on Modeling, Analysis and Simulation of Comp. and Telecomm. Systems*, pages 41–50, 2008.

[13] Robert Ennals and Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, 2006.

[14] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 125–134, 2008.