Analytical/ML Mixed Approach for Concurrency Regulation in Software Transactional Memory

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

***Abstract*—In this article we exploit a combination of analytical and Machine Learning (ML) techniques in order to build a performance model allowing to dynamically tune the level of concurrency of applications based on Software Transactional Memory (STM). Our mixed approach has the advantage of reducing the training time of pure machine learning methods, and avoiding approximation errors typically affecting pure analytical approaches. Hence it allows very fast construction of highly reliable performance models, which can be promptly and effectively exploited for optimizing actual application runs. We also present a real implementation of a concurrency regulation architecture, based on the mixed modeling approach, which has been integrated with the open source TinySTM package, together with experimental data related to runs of applications taken from the STAMP benchmark suite demonstrating the effectiveness of our proposal.**

I. Introduction

By relying on the notion of *atomic transaction*, Software Transactional Memory (STM) [1] has recently emerged as an attractive programming paradigm for parallel/concurrent applications. It allows code blocks accessing shared-data to be marked as transactions, for which it takes care of managing coherency of data access/manipulation, thus avoiding the need for any handcrafted synchronization scheme to be provided by the application programmer. The relevance of the STM paradigm has significantly grown given that multicore systems have become mainstream platforms. Also, STM is the representative technology for several in-memory Cloud-suited data-platforms (such as VMware vFabric GemFire, Oracle Coherence and Apache Cassandra), where the encapsulation of application code within transactions allows concurrent manipulation of in-memory kept application data according to specific isolation levels, which is done transparently to the programmer.

Even though the STM potential for simplifying the software development process is extremely high, another aspect that is central for the success, and the further diffusion of, the STM paradigm relates to the actual level of performance it can deliver. Particularly, one core issue to cope with is related to exploiting parallelism while also avoiding thrashing phenomena due to excessive transaction rollbacks, caused by excessive contention on logical resources, namely concurrently accessed data portions. We note that this aspect has reflections also on the side of resource provisioning in the Cloud, and associated costs, since thrashing leads to suboptimal usage of resources (including energy) by, e.g., PaaS providers offering STM based platforms to customers (see, e.g., the Cloud-TM platform [2]).

Literature solutions dealing with STM run-time efficiency can be framed within two different sets of orthogonal approaches. On one side we find optimized schemes for transaction conflict detection and management [3], [4], [5], [6], [7]. These include proposals aimed at dynamically determining which threads need to execute specific transactions, so to allow transactions that are expected to access the same data to run along a same thread in order to sequentialize and spare them from incurring the risk of being aborted with high probability. Other proposals rely instead on pro-active transaction scheduling [8], [9] where the reduction of performance degradation due to transaction aborts is achieved by avoiding to schedule (hence delaying the scheduling of) the execution of transactions whose associated conflict probability is estimated to be high.

On the other side we find solutions aimed at supporting performance optimization via the determination of the best suited level of concurrency (number of threads) to be exploited for running the application on top of the STM layer (see, e.g., [10], [11], [12]). These solutions are clearly orthogonal to the aforementioned ones, being potentially usable in combination with them. On the other hand, we can further distinguish these approaches depending on whether they cope with dynamic or static application execution profiles, and on the type of methodology that is used to determine (predict) the well suited level of concurrency for a specific (phase of the execution of the) application. Approaches coping with static workload profiles are not able to predict the optimal level of concurrency for applications where classical parameters expressing proper dynamics of the applications (such as the average number of data-objects touched by a transactional code block) can vary over time. On the other hand, prediction approaches that have been proposed in literature either rely on analytical methods, or on black-box Machine Learning (ML) methodologies. The former ones have the advantage of generally requiring a lightweight application profiling for gathering data to be filled to the prediction model, but provide (slightly) less accurate predictions and in some cases require stringent assumptions to be met by the real STM system in order for its dynamics to be reliably captured by the analytical formulas. On the contrary, ML methods usually require expensive profiling in order to build the knowledge base

that would suffice to instantiate the performance prediction model, which may make the actuation of the optimized concurrency configuration untimely. On the other hand, they typically allow very accurate estimation of the real performance trends of the STM system (see, e.g., [13], [14]).

In this paper we cope with the issue of determining the optimal level of concurrency by presenting an Analytical/ML (AML) mixed approach allowing to chase the best of the two methodologies by tackling the shortcomings intrinsic in each of them. On one side, we allow the training phase required to define the "application specific" performance model to be significantly reduced, compared to pure ML techniques, while also allowing the final AML model to be significantly more precise than pure analytical approaches. In fact, it can ensure the same level of precision as the one provided by pure ML techniques. Further, the AML model we provide is able to cope with cases where the actual execution profile of the application, namely the workload features, can change over time, such as when the (average) size of the data-set accessed by the transactional code in read or write mode changes over time (e.g. according to a phase-behavior). This is not always allowed by pure analytical approaches [10], [12]. Overall, we provide a methodology for fast construction of a highly reliable performance model allowing the determination of the optimal level of concurrency for the specific STM-based application. This is relevant in generic contexts including the Cloud, where the need for deploying new applications (or applications with reshuffling in their execution profile) and promptly determining the system configurations allowing optimized resource usage, is very common.

We also present a real implementation of a concurrency regulation architecture, integrated with the TinySTM open source package [4], which exploits the AML model to dynamically tune the number of threads to be used for running the application. Further, we report experimental results, achieved by running the applications belonging to the STAMP benchmark suite [15] on top of a 16-core HP ProLiant machine, which show the effectiveness of the proposed approach compared to pure analytical or pure ML techniques.

The remainder of this paper is organized as follows. In Section II, related work is discussed. The target system architecture for our performance model is presented in Section III. The AML modeling approach is presented in Section IV. The concurrency-regulation architecture based on the AML model and the experimental analysis of the whole approach are presented in Section V.

II. Related Work

1) Transaction Scheduling: Some literature approaches are based on pro-active transaction scheduling, which relies on the observation of data contention over the recent past of the application. In the approach proposed in [9], incoming transactions are enqueued and sequentialized for execution along a same thread when an indicator, referred to as *contention intensity*, exceeds a pre-established threshold. A variant of such a scheme is provided in [16], where multiple serialization queues (one per active thread) are used, so to better afford partitioned accesses onto the data set. The proposal in [17] sequentializes a transaction when a potential conflict with other running transactions is predicted. Actually, the sequentializing mechanism is activated only when the amount of aborted vs committed transactions exceeds a given threshold. The work in [18] introduces operating system scheduling supports for threads running transactions within an STM environment in order to reduce the likelihood of aborts. This is achieved by, e.g., stretching the time-slice assigned to the thread entering a transactional code block, thus reducing its vulnerability interval (namely, the time-interval along which concurrent operations by other threads may invalidate its work). Compared to our approach, all the above proposals do not directly estimate the wasted time due to aborted transactions (vs the level of concurrency), rather they indirectly attempt to control the wasted time according to heuristics schemes.

2) Concurrency Level Optimization: The (dynamic) identification of the well suited level of concurrency in STM systems, leading to optimized throughput, has been dealt with in literature via differentiated approaches. In [10] an analytical model 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 conflict detection and management scheme used by the target STM is required, which is instead not required by the approach we are proposing.

The work in [12] presents an analytical model taking in input a workload characterization of the application expressed in terms of transaction profiles, contention probability and hardware resources consumption. The model predicts the application execution time as function of the number of concurrent threads sustaining the application. However the prediction only accounts for the average system behavior over the whole lifetime of the application. Hence, differently from our proposal, no ability to capture run-time variations, and the consequent need for dynamic adaptation of the level of concurrency, is provided.

The proposal in [19] is targeted at evaluating scalability aspects of STM systems. It relies on the usage of different types of functions (e.g. polynomial and logarithmic functions) to approximate the application performance 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 that the workload profile of the application is not taken into account, hence the prediction may prove unreliable when the profile changes wrt the one used during measurement and interpolation phases. In [20], an analytical model is used to regulate the level of parallelism of STM-based applications, which is developed through the interpolation of real performance samples using predefined mathematical functions. Differently from [19], this proposal takes into account the profile of the application. We will use this result as the basis for our innovative AML modeling approach, and will also compare the concurrency regulation architecture we provide with the one presented in [20].

In [14], we have provided a neural network based approach to regulate the level of concurrency of STM based applications. A weak point of this approach is that, to achieve good performance prediction capabilities, it is necessary to collect a consistent number of samples of real application runs, hopefully distributed over the whole domain defining the input parameters determining the shape of the performance curve. Our AML model is exactly aimed at bypassing this problem, thus achieving fast construction of a highly reliable performance predictor.

Finally, there are proposals that dynamically adjust the level of concurrency of the STM system on the basis of heuristics. In [8] 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 rate exceeds some threshold while it is increased when the rate is lower than another threshold. In [11] a concurrency regulation approach is provided, based on the hill-climbing heuristic scheme. The approach determines whether the trend of increasing/decresing the concurrency level has positive effects on the STM throughput, in which case the trend is maintained. A variant is also provided, which exploits the performance model of distributed STM systems in [21] in order to accelerate the exploration process when also targeting the selection of the number of STM nodes to be employed within the distributed platform. Differently from our proposal, the heuristics in these works do not directly attempt to capture the relation between the actual transaction profile and the achievable performance (depending on the level of parallelism). This leads them to be mostly suited for static application profiles.

III. Target System Architecture and Performance Model Aim

A. Description of the Target STM System

We consider an STM system where the execution flow of each thread is characterized by the interleaving of transactions and non-transactional code blocks. During the execution of the transaction, the thread can perform read and write operations on a set of shared data objects, and can run code blocks where it does not access shared data objects (e.g. it accesses variables within its own stack). Read (written) data objects 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. A non-transactional code block starts 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.

B. Aim of the AML Modeling Approach

Typical STM oriented concurrency control algorithms [3] rely on approaches where the execution flow of a transaction never traps into operating system blocking services. Rather, spin-locks are exploited to support synchronization activities across the threads. In such a scenario, the primary index having an impact on the throughput achievable by the STM system (and having a reflection on how energy is used for productive work) is the so called *transaction wasted time*, namely the amount of CPU time spent by a thread for executing transaction instances that are eventually aborted.

The ability to predict the transaction wasted time, for a given application profile (namely for a specific data access profile) while varying the degree of parallelism in the execution is the fulcrum of our AML based optimization proposal. More in detail, our AML model is aimed at computing pairs of values $< w_{time,i}, i >$ where i indicates the level of concurrency, namely the number of threads which is supposed to support the execution of the application, and $w_{time,i}$ is the expected transaction wasted time (when running with degree of concurrency equal to the value i), namely the amount of time spent by any thread while running aborted instances of a given transaction. Denoting with t the average transaction execution time (namely the expected CPU time required for running an instance of transaction that is not eventually aborted) and with ntc the average time required for running a non-transactional code block (which is interleaved between two subsequent transactional code blocks in our system model), we can compute the system throughput when running with i threads as

$$thr_i = \frac{i}{w_{time,i} + t + ntc} \tag{1}$$

By exploiting Eq. 1, the objective of the concurrency regulation architecture we present is to identify the value of i, in the interval $[1, max\_threads]$, such that thr_i is maximized.

We will proceed along the following path. We will initially exploit a combination of literature approaches, either analytical or machine learning, for the construction of an AML model evaluating $w_{time,i}$ for the different values of i. Essentially this will be based on introducing an algorithm for the combined usage of the two approaches. As we will show,

$w_{time,i}$ will be expressed as a function of t and ntc. However, these quantities may depend, in their turn, on the value of i due to different thread contention dynamics on system level resources when changing the number of threads. As an example, per-thread cache efficiency may change depending on the number of STM threads operating on a given shared-cache level, thus impacting the CPU time required for a specific code block, either transactional or non-transactional. To cope with this issue, we will provide analytical correction functions allowing, once known the value of t (or ntc) when running with k threads, which we denote as t_k and ntc_k respectively, to predict the corresponding values when supposing a different number of threads. This will lead the final throughput prediction to be actuated via the formula

$$thr_i = \frac{i}{w_{time,i}(t_i, ntc_i) + t_i + ntc_i} \tag{2}$$

where for $w_{time,i}$ we only point out the dependence on t_i and ntc_i, while we intentionally delay to the next section the presentation of the other parameters playing a role in its expression. Overall, the finally achieved performance model in Eq. 2 has the ability to determine the expected transaction wasted time when also considering contention on system level resources (not only logical resources, namely shared-data) while varying the number of threads in the system.

As a final note, in our approach we will consider (and experiment in) scenarios where $max\_threads$ is set to the number of available CPU-cores.

IV. The Actual AML Model

We aim at building a model for $w_{time,i}$ that has the ability to capture changes in the transaction wasted time not only in relation to variations of the number of threads running the application, but also in relation to changes in the run-time behavior of transactional code blocks (such as variations of the amount of shared-data accessed in read/write mode by the transaction). In fact, the latter type of variation may require changing the number of threads to be used in a given phase of the application execution (exhibiting a specific execution profile) in order to re-optimize performance. Our recent results in the field of either analytical or machine learning modeling [20], [14] have pointed out how capturing the combined effects of concurrency degree and execution profile on the transaction wasted time can be achieved in case $w_{time,i}$ is a expressed as a function f depending on a proper set of input parameters, namely

$$w_{time,i} = f(rs, ws, rw, ww, t, ntc, i) \tag{3}$$

where t, ntc and i have the meaning explained above, while the other input parameters are explained in what follows:

- rs is the average read-set size of transactions;
- ws is the average write-set size of transactions;
- rw (read-write conflict affinity) is an index providing an estimation of the likelihood for an object read by some transaction to be also written by some other transaction;
- ww (write-write conflict affinity) is an index providing an estimation of the likelihood for an object written by some transaction to be also written by another transaction.

The objective of the AML model is to provide an approximation f_{AML} of the function f. To this purpose, we combine two different existing estimators, providing two different approximations of f. The first estimator, which we refer to as f_A, is based on an analytical approach, while the second one, which we refer to as f_{ML}, relies on a pure machine learning approach. We briefly recall these two base performance models, and then enter the details of the algorithmic steps used for combining them, namely the algorithm that determines the construction of f_{AML}.

A. Base Analytical Model: f_A

Our base analytical model is built on top of the results in [20]. This work presents a parametric analytical expression of the probability for a transaction to be aborted, namely p_a, which is a function of the parameters appearing in input to Eq. 3. Particularly, the abort probability is expressed as

$$p_a = \beta(rs, ws, rw, ww, t, ntc, i) \tag{4}$$

More precisely

$$p_a = 1 - e^{-\rho \cdot \omega \cdot \phi} \tag{5}$$

where the function ρ is assumed to depend on the input parameters rs, ws, rw and ww, the function ω is assumed to depend on the parameter i (number of concurrent threads), and the function ϕ is assumed to depend on the parameters t and ntc. For the reader's convenience, we report below the final shape of each of these functions as determined in [20]

$$\begin{aligned}\rho =& [c \cdot (ln(b \cdot ws + 1)) \cdot ln(a \cdot ww + 1)]^d \\ &+ [e \cdot (ln(f \cdot rw + 1)) \cdot ln(g \cdot rs + 1) \cdot ws]^z\end{aligned} \tag{6}$$

$$\omega = h \cdot (ln(l \cdot (k - 1) + 1) \tag{7}$$

$$\phi = m \cdot ln(n \cdot \frac{t}{t + ntc} + 1) \tag{8}$$

where m, n, h, l, e, f, g, z, c, b, a, d are all fitting parameters to be instantiated via regression.

We can finally use the abort probability expression, as provided in [20] (see Eq.s 4-8), in order to analytically express the expected transaction wasted time (when running with i threads), namely to instantiate the function f_A, as

$$w_{time,i} = f_A = \frac{p_a}{1 - p_a} \cdot tr \tag{9}$$

where tr is the average CPU time for a single aborted run of the transaction, and $p_a/(1 - p_a)$ is the expected number of aborted runs of the transaction.

B. Base Machine Learning Model: f_{ML}

As for the machine learning predictor of $w_{time,i}$, we rely on the approach we provided in [14], where the function f in Eq. 3 is approximated by relying on a neural network aimed at estimating the shape of the approximating function f_{ML}. We recall that a neural network based model can be trained to approximate an unknown function f via the exploitation of a data set $\{(\mathbf{input}, \mathbf{output})\}$ (training set), which is assumed to be a statistical representation of the function f such that, for each element $(\mathbf{input}, \mathbf{output})$, $\mathbf{ouput} = f(\mathbf{input}) + \delta$, where δ is a random variable (also said *noise*). In the approach in [14], the training set is formed by samples $(\mathbf{input}, \mathbf{output})$, with $\mathbf{input} = \{rs, ws, rw, ww, t, ntc, i\}$ and $\mathbf{output} = w_{time,i}$, which are collected during real executions of the STM application.

C. Combining the Two Base Models: f_{AML}

Both the two base models, namely f_A and f_{ML}, require a training phase to be actuated in order for them to be instantiated. Specifically, f_A requires collecting samples related to the application execution in order to compute the fitting parameters appearing in Eq.s 6-8, and to estimate tr. On the other hand, f_{ML} is constructed by collecting a set of $(\mathbf{input}, \mathbf{output})$ training samples related to the real execution of the STM application. For both the approaches, each sample used to instantiate the model will refer to aggregate statistics (on the values of the parameters $\{rs, ws, rw, ww, t, ntc, i\}$) over multiple committed transactions, typically on the order of several thousands. However, there is a fundamental difference in the training phases to be operated for instantiating the two models.

As discussed and experimentally shown in [20], the f_A model (particularly the expression for p_a) can be instantiated by relying on a (very) limited amount of run-time samples taken during real executions of the application. This implies that, upon deploying the application, a reduced number of configurations, in terms of the concurrency level (expressed by the value of the parameter i), require to be observed (and for a relatively reduce amount of time) in order to build a model having the ability to provide performance predictions in relation to very different levels of concurrency (potentially unexplored in the training phase). In other words, the f_A model offers excellent extrapolation capabilities.

This is not true for the case of f_{ML}, which typically requires to be trained via good coverage of the whole $\mathbf{input}$ domain, also in terms of the degree of concurrency i. This leads to the need for observing the application for longer time, and in differently parameterized operating modes. On the other hand, f_{ML} is expected to be an highly reliable estimator for f (even more reliable than f_A) in case such a good coverage of the $\mathbf{input}$ domain is guaranteed to be achieved during the training phase [14].

We decided to combine the usage of the two modeling approaches by exploiting f_A in order to definitely shorten the length of the training phase required to instantiate f_{ML}. Overall, in our mixed modeling methodology the analytical component is used as a support to improve some aspect (namely the learning latency) of the machine learning component.

A core aspect in our combination of analytical and machine learning models is the introduction of a new type of training set for the machine learning component, which we refer to as Virtual Training Set (denoted as VTS). Particularly, VTS is a set of virtual $(\mathbf{input^v}, \mathbf{output^v})$ training samples where:

- $\mathbf{input^v}$ is the set $\{rs^v, rs^v, rw^v, ww^v, t^v, ntc^v, i^v\}$ formed by stochastically selecting the value of each individual parameter belonging to the set;
- $\mathbf{output^v}$ is the output value computed as $f_A(\mathbf{input^v})$, namely the estimation of w_{time,i^v} actuated by f_A on the basis of the stochastically selected input values.

In other word, VTS becomes a representation of how the STM system behaves, in terms of the relation between the expected transaction wasted time and the value of configuration or behavioral parameters (such as the degree of concurrency), which is built without the need for actually sampling the real system behavior. Rather, the representation provided by VTS is built by sampling Eq. 9, namely f_A. We note that the latency of such sampling process is independent of the actual speed of execution of the STM application, which determines in its turn the speed according to which individual $(\mathbf{input}, \mathbf{output})$ samples, referring to real executions of the application, would be taken. Particularly, the sampling process of f_A is expected to be much faster, especially because the stochastic computation (e.g. the random computation) of any of its input parameters, which needs to be actuated at each sampling-step of f_A, is a trivial operation with negligible CPU requirements. On the other hand, the possibility to build the VTS is conditioned to the previous instantiation of the f_A model. However, as said before, this can be achieved via a very short profiling phase, requiring the collection of a few samples of the actual behavior of the STM application. Overall, we list below the algorithmic steps required for building the application specific VTS, to be used for finalizing the construction of the f_{AML} model:

Step-A. We randomly select Z different values of i in the domain $[1, max\_threads]$, and for each selected value of i we observe the application run-time behavior by taking δ real-samples, each one including the set of parameters $\{rs, ws, rw, ww, t, ntc, i, tr\}$.

Step-B. Via regression we instantiate all the fitting parameters requested by Eq.s 6-8. Hence, at this stage we have an instantiation of Eq. 5, namely the model instance for p_a.

Step-C. We fill the instantiated model for p_a in Eq. 9, together with the average value of tr sampled in **Step-A**, and then we generate the VTS. This is done by generating δ' virtual samples $(\mathbf{input^v}, \mathbf{output^v})$

where $\mathbf{input^v} = \{rs^v, ws^v, rw^v, ww^v, t^v, ntc^v, i^v\}$ and $\mathbf{output^v} = w_{time,i^v}$ as computed by the model in Eq. 9. Each $\mathbf{input^v}$ sample is instantiated by randomly selecting the values of the parameters that compose it ([1]). For the parameter i the random selection is in the interval $[1, max\_threads]$, while for the other parameters the randomization needs to take into account a plausible domain, as determined by observing the actual application behavior in **Step-A**. Particularly, for each of these parameters, its randomization domain is defined by setting the lower extreme of the domain to the minimum value that was observed while sampling that same parameter in **Step-A**. On the other hand, the upper extreme for the randomization domain is calculated as the value guaranteeing the 90-percentile coverage of the whole set of values sampled for that parameter in **Step-A**, which is done in order to reduce the effects due to spikes.

After having generated the VTS in **Step-C**, we use it in order to train the machine learning component f_{ML} of the modelling approach. However, training f_{ML} by only relying on VTS would give rise to a final f_{ML} estimator identical to f_A given that the curve learned by f_{ML} would exactly correspond to the one modelled by f_A. Hence, in order to improve the quality of the machine learning based estimator, our combination of analytical and machine learning methods relies on additional algorithmic steps where we use VTS as the base for the construction of an additional training set called Virtual-Real Mixed Training Set (denoted as VRMTS). This set represents a variation of VTS where some virtual samples are replaced with real samples taken by observing the real behavior of the STM application, still for a relatively limited amount of time. More in detail, the following two additional algorithmic steps are used for constructing the VRMTS:

Step-D. We select Z' different values for i (in the interval $[1, max\_threads]$), and for each selected value we observe the application run-time behavior by taking δ'' real training samples $(\mathbf{input^r}, \mathbf{output^r})$.

Step-E. We initially set VRMTS equal to VTS. Then we generate the final VRMTS image via an iterative procedure that substitutes at each iteration one element in VRMTS with one $(\mathbf{input^r}, \mathbf{output^r})$ sample from the sequence of samples taken in **Step-D**, until this sequence ends.

The rationale behind the construction of VRMTS is to improve the quality of the final training set to be used to build the machine learning model by complementing the virtual samples originally appearing in VTS with real data related to the execution of the application. Two things need to be considered in this process: (1) the actual length of **Step-D** could be further reduced by reusing (all or part of the) real samples of the application execution taken in **Step-A**, which were exploited in **Step-B** for computing the fitting parameters for the f_A model; (2) the substitution in **Step-E** could be actuated according to differentiated policies.

As for the latter aspect, we have decided to use a policy based on Euclidean distance, in order to avoid clustering phenomena leading the final VRMTS image to contain training samples whose distribution within the whole domain significantly differs from the original distribution determined by the random selection process used in **Step-C** for the construction of VTS. More in detail, the victim selection policy we have adopted to replace iteratively any sample while generating the final VRMTS works as follows:

- given a collected real sample of the application execution $s^r = (rs^r, ws^r, rw^r, ww^r, t^r, ntc^r, i^r)$, the subset $S_{i^r} = \{(rs, ws, rw, ww, t, ntc, i) | i = i^r\}$ of VRMTS is computed. Actually, S_{i^r} is the subset of samples for which the level of parallelism i they refer to is the same as the level of parallelism characterizing the real sample to be used for replacement in the current iterative step;
- the actual sample in VRMTS to be replaced with s^r is identified inside the subset S_{i^r} using the Euclidean distance as computed on all the parameters characterizing the sample except i (namely rs,ws, rw, ww, t and ntc). Particularly, the victim is the sample s^* belonging to S_{i^r} which is closest to s^r.

We note that the above Euclidean distance based policy may lead in intermediate steps to evict from VRMTS some previously inserted real sample. This may happen in case the closest sample to the one currently being inserted in VRMTS is a real sample (which was inserted in a previous iteration). This is not a drawback of our victim selection policy, rather it is the reflection of the fact that we prevent clustering effects of the elements included in the final image of VRMTS, which may lead some portions of the domain not to be sufficiently represented within the set. As a final note, the current proposal does not account for substituting virtual training samples by explicitly having the real ones (taken in **Step-D**) evenly distributed across different execution phases of the applications, if any (possibly leading to different actual profiles). This aspect will be the objective of future investigations.

Once achieved the final VRMTS image, we use it to train f_{ML} in order to determine the final AML model. Overall, f_{AML} is defined as the instance of f_{ML} trained via VRMTS.

D. Correcting Factors

As pointed out, the instantiation of the f_{AML} model for the prediction of $w_{time,i}$ needs to be complemented with a predictor of how t and ntc are expected to vary vs the degree of parallelism i. In fact, $w_{time,i}$, as expressed by the instance of machine learning predictor trained via VRMTS depends on t and ntc. Also, the final equation establishing the system throughput, namely Eq. 2, which is used for evaluating the optimal concurrency level, also relies on the

[1]Generally speaking, this step could take advantage from a selection algorithm providing minimal chances of collision.

ability to determine how t and ntc change when changing the level of parallelism (due to contention on hardware resources). To cope with this issue, we rely on correcting functions aimed at determining (predicting) the values t_i and ntc_i once known the values of these same parameters when running with parallelism level $k \neq i$. To achieve this goal, the samples taken in **Step-A** are used to build, via regression, the function expressing the variation of the number of clock-cycles the CPU-core spends waiting for data or instructions to come-in from the RAM storage system. We recall that the collection of training samples in **Step-A** should be made very short, hence referring to a limited number of values of the concurrency level i. However, the expectation is that the number of clock-cycles spent in waiting phases should scale (almost) linearly vs the number of concurrent threads used for running the application. Hence, regression on a limited number of samples should suffice for reliable instantiation of the correction functions. To support our claim, we report in Figure 1 and in Figure 2 the variation of the clock-cycles spent while waiting data to come from the RAM storage system for two different STM applications of the STAMP benchmark suite [15], namely *Intruder* and *Vacation*, while varying the number of threads running the benchmarks between 1 and 16. These data have been gathered on top of a 16-core HP ProLiant machine, equipped with 2 AMD Opteron™6128 Series Processor, each one having eight hardware cores, and 32 GB RAM, running a Linux Debian distribution with kernel version 2.6.32-5-amd64. This is the same machine we exploited for the experimental assessment of the whole AML methodology presented in Section V. The reported statistics have been collected via the `perf` tool, which marks the stall cycles while gathering data from RAM storage as `Stalled-Cycles-Backend`. By the curves, the close-to-linear scaling is fairly evident, hence, once determined the scaling curve via regression, which we denote as sc,

$$t_i = t_k \times \frac{sc(i)}{sc(k)} \qquad ntc_i = ntc_k \times \frac{sc(i)}{sc(k)} \tag{10}$$

where:

- t_i is the estimated expected CPU time (once known/estimated t_k) for a committed transaction in case the application runs with level of concurrency i;
- ntc_i is the estimated expected CPU time (once known/estimated ntc_k) for a non-transactional code block in case the application runs with level of concurrency i;
- $sc(i)$ (resp. $sc(k)$) is the value of the correction function for level of concurrency i (resp. k).

V. Experimental Evaluation

A. The AML Based Concurrency Regulation Architecture

We have implemented a fully featured STM concurrency regulation architecture based on AML, which we refer to as AML-STM ([2]), whose organization is presented in Figure 3. The core STM layer exploited in our implementation is the open source TinySTM [4]. AML-STM is made up by three building blocks, namely: A Statistics Collector (SC); A Model Instantiation Component (MIC); and A Concurrency Regulator (CR). The MIC module initially interacts with CR in order to induce variations of the number of running-threads i so that the SC module is allowed to perform the sampling process requested to support **Step-A** of the instantiation of the AML model ([3]). After the initial sampling phase, the MIC module instantiates f_A (and the correction function sc) and computes VTS. It then interacts again with CR in order to induce variations of the concurrency level i that are requested to support the sampling process (still actuated via SC) used for building VRMTS (see **Step-D** and **Step-E**). It then instantiates f_{AML} by relying on a neural network implementation of the f_{ML} predictor, which is trained via VRMTS. Once the f_{AML} model is built, MIC continues to gather statistical data from SC, and depending on the values of $w_{time,i}$ that are predicted by f_{AML} (as a function of the average values of the sampled parameters rs, ws, rw, ww, t_i, and ntc_i), it determines the value of i providing the optimal throughput by relying on Eq. 2. This value is filled in input to CR (via queries by CR to MIC), which in its turn switches off or activates threads depending on whether the level of concurrency needs to be decreased or increased for the next observation period.

We note that the length of the phases requested for eventually instantiating f_{AML} depend on the amount of samples that are planned to be taken in **Step-A** and in **Step-D** of the model construction (see the parameters Z, δ, Z' and δ'' in the detailed description of these steps). We will evaluate the effectiveness of our AML modeling approach, and compare this approach with pure analytical or machine learning based methods, while varying the length of these sampling phases. We note that the shorter such a length, the more promptly the final performance model to be used for concurrency regulation is available. Hence, reduction of the length of these phases, while still guaranteing accuracy of the finally built performance model, will allow more prompt optimization of the run-time behavior of the STM based application. As hinted, this is relevant in scenarios where applications are dynamically deployed, and need to be promptly optimized in terms of their run-time behavior in order to improve the fruitful usage of resources and to also improve the system energy efficiency (via reduction of wasted CPU time), such as when applications are hosted by PaaS providers on top of STM-based platforms.

[2]The source code is freely available at the URL `http://www.dis.uniroma1.it/~hpdcs/AML-STM.zip`

[3]As for the parameters to be monitored via SC, rw can be calculated as the dot product between the distribution of read operations and the distribution of write operations (both expressed in terms of relative frequency of accesses to shared data objects). Similarly, ww can be calculated as the dot product between the distribution of write operations and itself. This can be achieved by relying on histograms of relative read/write access frequencies.

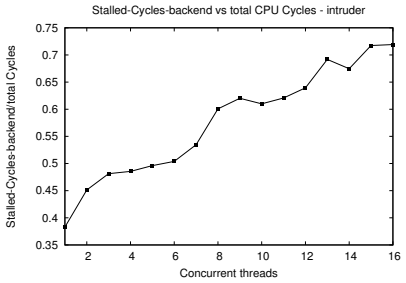

Figure 1. Stalled cycles back-end for the Intruder benchmark

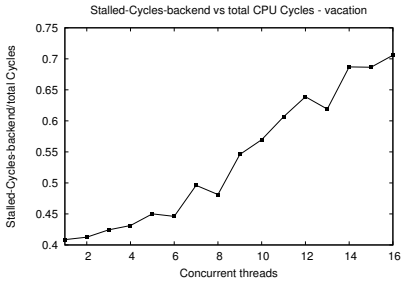

Figure 2. Stalled cycles back-end for the Vacation benchmark

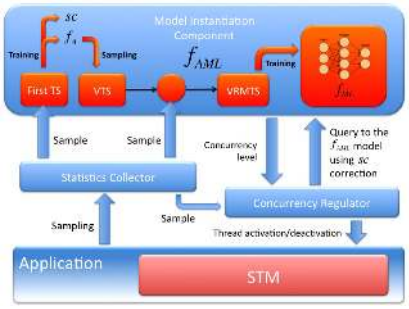

Figure 3. System architecture

B. Experimental Data

The data we report in this section refer to the execution of applications belonging to the STAMP benchmark suite [15]. These applications have been run on top of the aforementioned 16-core HP ProLiant machine. This section is divided in two parts. In the first one we provide an experimental support of the feasibility of our AML approach. Specifically, we provide data related to how the prediction error of $w_{time,i}$ changes over time (namely vs the length of the sampling phase used to gather data to instantiate the performance model) when comparatively considering our AML model and the two base models, pure analytical and pure machine learning, exploited for building AML.

Successively, we provide experimental data related to how the concurrency regulation architecture based on AML, namely AML-STM, allows more prompt achievement of optimized run-time performance and optimized energy usage, when compared to the concurrency regulation architectures we have presented in [14], [20], where concurrency regulation takes place by exclusively relying on an analytical performance model or on a pure machine learning approach. We will refer to these two architectures as A-STM and ML-STM, respectively. We note that both these architectures have been implemented by relying on TinySTM as the core STM layer, hence our study provides a fair comparison of the different performance modeling and optimization approaches, when considering the same STM technology and implementation. Also, we feel that comparing our AML approach with literature approaches addressing the very same problem (namely the dynamic selection of the optimal value of the number of threads in scenarios where the application execution profile can change over time) is the more reliable way of assessing the present proposal ([4]). In fact, comparing AML with approaches based on different rationales (like the ones based on transaction scheduling, see [16]) would lead to compare solutions that can be integrated and make synergy, thus not representing alternatives excluding each other.

[4]The proposals in, e.g. [11], [10], [12], are suited for selecting and/or regulating concurrency with static execution profiles, where, e.g., read and write set size does not change over time. We exclude therefore these solutions in our comparative study.

1) Part A - Model Accuracy: To determine how the estimation accuracy of $w_{time,i}$ provided by the AML approach varies vs the length of the sampling phase used to gather profiling data on top of which the performance model is built, and to compare such accuracy with the one provided by pure analytical (f_A) or pure machine learning (f_{ML} trained on real samples) methods, we have performed the following experiments. We have profiled STAMP applications by running them with different levels of concurrency, which have been varied between 1 and the maximum amount of available CPU-cores, namely 16. All the samples collected up to a point in time have been used either to instantiate f_A via regression, or to train f_{ML} in the pure machine learning approach. On the other hand, for the case of f_{AML} they are used according to the following rule. The 10% of the initially taken samples in the observation interval are used to instantiate f_A (see **Step-A** and **Step-B** in Section IV), which is then used to build VTS, while the remaining 90% are used to derive VRMTS (see **Step-D** and **Step-E** in Section IV). In this scheme the cardinality of the VTS, from which VRMTS is build, has been fixed at 1500 elements. Also, each real sample taken during the execution of the application aggregates the statistics related to 4000 committed transactions, and the samples are taken in all the scenarios along a single thread, thus leading to similar rate of production of profiling data independently of the actual level of concurrency while running the application. Hence, the knowledge base on top of which the models are instantiated is populated with similar rates in all the scenarios.

Then for different lengths of the initial sampling phase (namely for different amounts of samples coming from the real execution of the application), we instantiated the three different models and compared the errors they provide in predicting $w_{time,i}$. These error values are reported in Figures 4-7, and refer to the average error while comparing predicted values with real execution values achieved while varying the number of threads running STAMP applications between 1 and the maximum value 16. Hence, they are average values over the different possible configurations of the concurrency degree for which predictions are carried out. Also, we have normalized the number of real samples used

in each approach in such a way that the x-axis expresses the actual latency for model instantiation (not only for real samples collection), hence including the latency (namely the overhead) for VTS and VRMTS generation and actual training of f_{ML} over VRMTS in case of the AML approach. This allowed us to compare the accuracy of the different models when considering the same identical amount of time for instantiating them (since for models requiring more processing activities in order for them to be instantiated, we recover that time by reducing the actual observation interval, and hence the number of real samples provided for model construction).

By the data we can see how the AML modeling approach always provides the minimal error independently of the length of the application profiling phase. Also, with the exception of *Vacation* and *Intruder*, AML allows achieving minimal errors (on the order of 2-3%) in about half of the time requested by the best of the other two models for achieving the same level of precision. On the other hand, for *Intruder*, AML significantly outperforms the other two prediction models for different lengths of the application sampling period. As for *Vacation*, AML provides close-to asymptotically minimal prediction error even with a very reduced amount of available profiling samples. These data support the claim of high accuracy of the predictions by AML, guaranteed via very reduced time for instantiating the application specific performance model.

2) Part B - Performance and Energy Efficiency: To demonstrate the effectiveness of AML in allowing prompt deliver of optimized performance (and prompt improvement of energy usage), once instantiated the performance models at some point in time according to the settings presented in Section V-B1, we evaluated both: (A) the transaction throughput, given that the concurrency level is dynamically regulated according to the predictions by the instantiated model and (B) the average energy consumption (joule) per committed transaction. For both the parameters, we also report the values achieved by running the application sequentially on top of a single thread and fixing the number of threads to the maximum value of 16 (we refer to this configuration as TinySTM in the plots), which allows us to establish baseline values for the assessment of both speedups and energy usage variations by the runs where the degree of concurrency is dynamically changed on the basis of the performance model predictions.

By the throughput data in Figures 8-11, we see how dynamic concurrency regulation based on AML allows the achievement of improved or even the peak observable throughput values much earlier in time, when compared to what happens with the pure analytical and the pure machine learning approaches. Also, the pure analytical approach is typically not able to provide the peak observed throughput, independently of the length of the sampling period during which the knowledge base for instantiating the model is being constructed. Also, for some benchmark, such as *Kmeans*, the time requested by the pure machine learning based approach in order to instantiate a model guaranteeing the peak observed performance is one order of magnitude longer than what required for the instantiation of the AML model. For other benchmarks, such as *Yada*, the AML approach requires on the order of 40% less model-instantiation time to achieve a model providing the peak performance. We also note that for most of the benchmarks, the TinySTM configuration where all the available 16 CPU-cores are used to run a fixed number of 16 concurrent threads, typically leads to a speed-down wrt the sequential run. This indicates how the execution profiles of STAMP applications are not prone to exploitation of uncontrolled parallelism, which leads the observed speedup values, promptly achievable via AML, to be representative of a significant performance boost.

As for data related to energy efficiency, reported in Figures 12-15, we see how both the pure analytical and the AML approaches allow prompt achievement of reduction of the energy requested per transaction commit. This is not guaranteed by the pure machine learning approach. Also, the AML approach allows the achievement of optimized tradeoffs between execution speed and energy consumption. In fact, even though the pure analytical approach allows reducing the energy consumption for the *Yada* benchmark when considering longer time for model instantiation, this is achieved by clearly penalizing the system throughput.

To provide more insights into the relation between speed and usage of energy, we report in Figure 16 the curves showing the variation of the ratio between the speedup provided by any specific configuration (again while varying the performance model instantiation time) and the energy scaling per committed transaction (namely the ratio between the energy used in a given configuration and the one used in the sequential run of the application). For space constraints we report these curves limited to the *Kmeans* benchmark, however the corresponding curves for the other benchmark applications could be derived by combining the previously presented curves. Essentially, the curves in Figure 16 express the speedup per unit of energy, when considering that the unit of energy for committing a transaction is the one employed by the sequential run. Hence they express a kind of iso-energy speedup. Clearly, for the sequential run this curve has constant value equal to 1. By the data we see how the AML approach achieves the peak observed iso-energy speedup for a significant reduction of the performance model instantiation time. On the other hand, the pure analytical approach does not achieve such a peak value even in case of significantly stretched application sampling phases, used to build the model knowledge-base. Also, the configuration with concurrency degree set to 16, namely TinySTM, further shows how not relying on smart and promptly optimized concurrency regulation, as the one provided by AML, degrades both performance and energy efficiency.

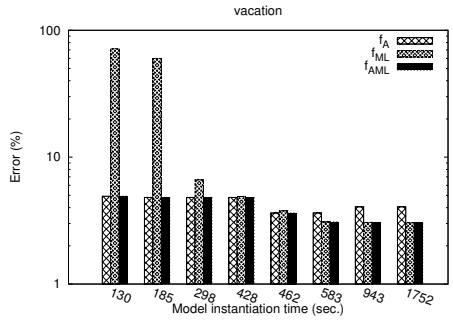

Figure 4. Prediction error comparison - vacation

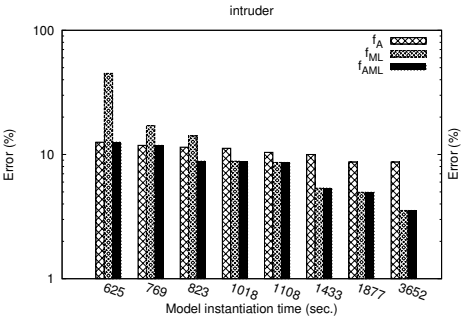

Figure 5. Prediction error comparison - intruder

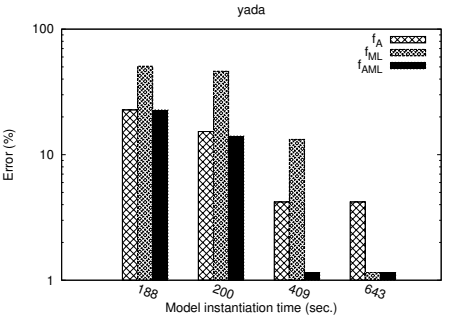

Figure 6. Prediction error comparison - yada

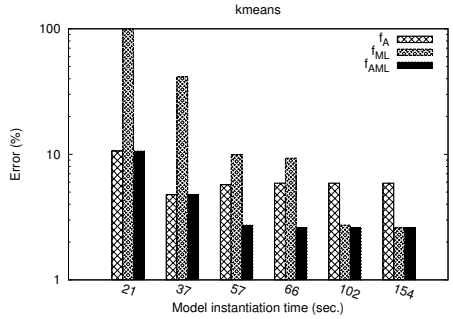

Figure 7. Prediction error comparison - kmeans

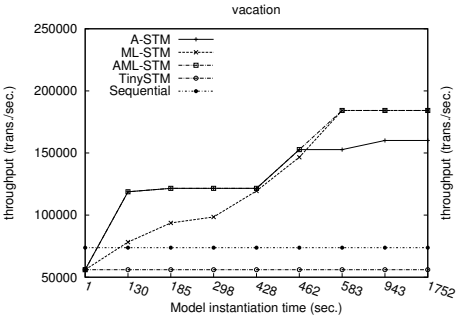

Figure 8. Throughput - vacation

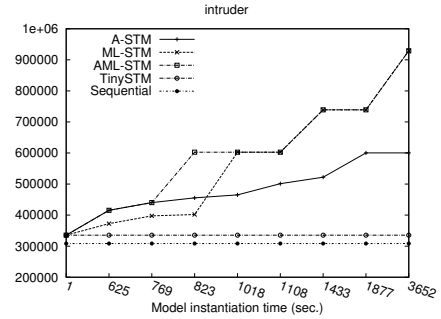

Figure 9. Throughput - intruder

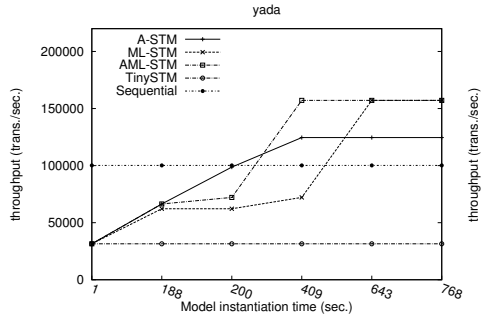

Figure 10. Throughput - yada

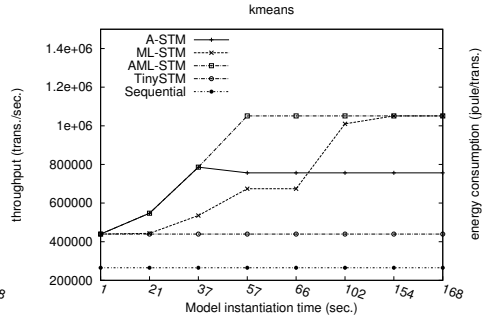

Figure 11. Throughput - kmeans

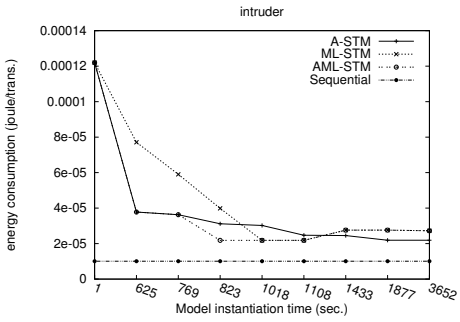

Figure 12. Energy consumption per committed transaction - intruder

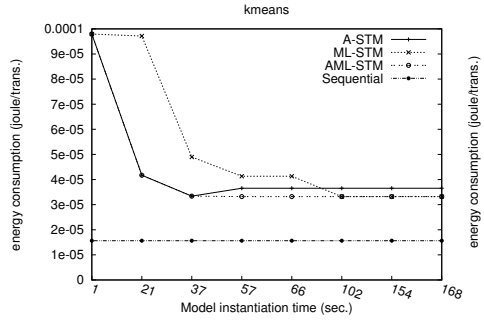

Figure 13. Energy consumption per committed transaction - kmeans

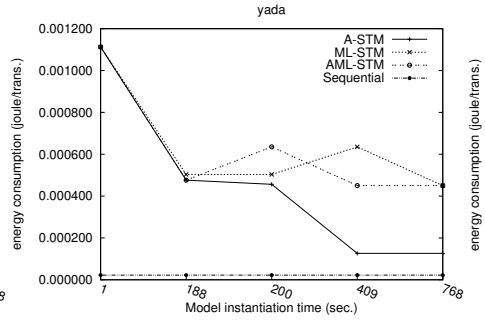

Figure 14. Energy consumption per committed transaction - yada

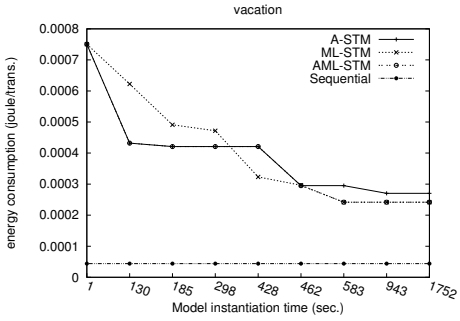

Figure 15. Energy consumption per committed transaction - vacation

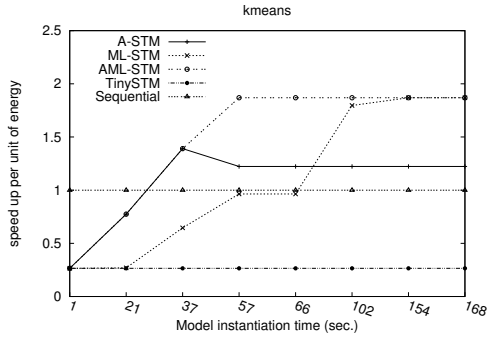

Figure 16. Iso-energy speedup curves

VI. Conclusions

We have presented an innovative approach for dynamically controlling the level of concurrency of STM applications, which is based on a performance model built by combining analytical and machine learning methodologies. The core advantage by this mixed method lies in its ability to definitely shorten the learning phase needed to instantiate the performance model (as compared to pure machine learning) and to improve the level of accuracy of pure analytical methods. We have also quantified these advantages experimentally, by studying the above tradeoffs for the case of the STAMP benchmark suite run on top of a 16-core HP ProLiant machine. The presented method well fits scenarios where fast construction of application specific performance models needs to be actuated in order to promptly optimize performance and also resource usage (including energy), given that unsuited concurrency levels in STM might lead on one side not to exploit parallelism, and on the other side to thrashing phenomena, due to excessive transaction rollbacks. Deployment of STM based applications on top of Cloud platforms is an example of this kind of scenarios.

References

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

[2] Cloud-TM: a Novel Programming Paradigm for the Cloud. http://http://www.cloudtm.eu/.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[4] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, pages 237–246, 2008.

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

[6] 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 Symposium on Principles and Practice of Parallel Programming*, pages 141–150, 2009.

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

[8] 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, 2008.

[9] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, 2008.

[10] 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.

[11] 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*, 2013.

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

[13] Maria Couceiro, Pedro Ruivo, Paolo Romano, and Luis Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2013.

[14] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 278–285, 2012.

[15] 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.

[16] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, pages 125–134, 2008.

[17] 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, 2009.

[18] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 79–90, 2010.

[19] Aleksandar Dragojević and Rachid Guerraoui. Predicting the scalability of an stm: A pragmatic approach. In *Proceedings of the 5th ACM Workshop on Transactional Computing*, 2010.

[20] 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 *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing System*, pages 31–40, 2013.

[21] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th ACM International Conference on Autonomic Computing*, pages 125–134, 2012.