Tuning the Level of Concurrency in Software Transactional Memory: an Overview of Recent Analytical, Machine Learning and Mixed Approaches

D. Rughetti, P. Di Sanzo, A. Pellegrini, B. Ciciani, F. Quaglia

Abstract Synchronization transparency offered by Software Transactional Memory (STM) must not come at the expense of run-time efficiency, thus demanding from the STM-designer the inclusion of mechanisms properly oriented to performance and other quality indexes. Particularly, one core issue to cope with in STM is related to exploiting parallelism while also avoiding thrashing phenomena due to excessive transaction rollbacks, caused by excessively high levels of contention on logical resources, namely concurrently accessed data portions. A means to address run-time efficiency consists in dynamically determining the best-suited level of concurrency (number of threads) to be employed for running the application (or specific application phases) on top of the STM layer. For too low levels of concurrency, parallelism can be hampered. Conversely, over-dimensioning the concurrency level may give rise to the aforementioned thrashing phenomena caused by excessive data contention—an aspect which has reflections also on the side of reduced energy-efficiency. In this chapter we overview a set of recent techniques aimed at building "application-specific" performance models that can be exploited to dynamically tune the level of concurrency to the best-suited value. Although they share some base concepts while modeling the system performance vs the degree of concurrency, these techniques rely on disparate methods, such as machine learning or

Diego Rughetti
e-mail: `rughetti@dis.uniroma1.it`

Pierangelo Di Sanzo
e-mail: `disanzo@dis.uniroma1.it`

Alessandro Pellegrini
e-mail: `pellegrini@dis.uniroma1.it`

Bruno Ciciani
e-mail: `ciciani@dis.uniroma1.it`

Francesco Quaglia
e-mail: `quaglia@dis.uniroma1.it`

DIAG — Sapienza, University of Rome

analytic methods (or combinations of the two), and achieve different tradeoffs in terms of the relation between the precision of the performance model and the latency for model instantiation. Implications of the different tradeoffs in real-life scenarios are also discussed.

1 Introduction

As mentioned earlier in this book, the TM paradigm has been conceived to ease the burden of developing concurrent applications, which is a major achievement when considering that, nowadays, even entry-level computing platforms rely on hardware parallelism, in the form of, e.g., multi-core chips. By simply encapsulating code that is known to access shared data within transactions, the programmer can produce a parallel application which is guaranteed to be correct, without incurring the complexities related to, e.g., lock-based programming.

The achievement of optimized run-time efficiency is clearly another core objective, given that the TM paradigm is not meant to achieve synchronization transparency while (excessively) sacrificing, e.g., performance. For STM systems, synchronization is demanded to an STM-library whose (run-time) configuration is crucial to achieve efficient runs of the overlying applications. This requires proper techniques to be put in place in order to effectively exploit the computing power offered by modern parallel architectures. Particularly, the central problem to be addressed by these techniques 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., [1]).

In order to cope with this issue, a plethora of solutions have been proposed, which can be framed into two different sets of orthogonal approaches. On one side, we find optimized schemes for transaction conflict detection and management [7, 12, 16, 17, 25]. These include proposals aimed at dynamically determining which threads need to execute specific transactions, so as 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 [2, 26] where the reduction of performance degradation due to transaction aborts is achieved by avoiding to schedule (hence delaying the scheduling 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 (i.e., number of threads) to be exploited for running the application on top of the STM layer (see, e.g., [5, 9, 15]). These solutions are clearly orthogonal to the aforementioned ones, being potentially usable in combination with them. 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 predict the well-suited level of concurrency for a specific application (or application phase). Approaches coping with static workload profiles are not able to predict the optimal level of concurrency for applications where typical 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.

The focus of this chapter is exactly on approaches for the (dynamic) tuning of the level of concurrency. Particularly, we will overview the STM-suited solutions we recently provided in [6, 21, 22]. The reason for selecting and focusing on these works in this comparative overview is twofold:

- They share the same basic model describing the level of performance as a function of the level of concurrency, which leads them to exhibit some kind of homogeneity; this will help drawing reliable conclusions while comparing them, which are likely to generalize. Also, the exploited basic model is able to capture scenarios where the application profile can vary over time, hence they appear as solutions whose usage is not limited to contexts with static profiles.
- They rely on alternative techniques to instantiate "application-specific" performance models, which range from analytical approaches to machine learning to a mix of the two. However, all of them are based on model-instantiation schemes exploiting training samples coming from the observation of the real application behavior during an (early) phase of deploy, which do not require stringent assumptions to be met by the real STM application in order for its dynamics to be reliably captured by the model. This further widens their usability in real life contexts.

Nonetheless, we will also provide a comparative discussion with literature approaches that stand as valuable alternatives for predicting the level of performance vs the degree of concurrency and/or for dynamically regulating the concurrency level to suited values.

We will initially start by discussing common points to all the overseen approaches, then we will enter details of each of them. Successively, we will provide hints on the organization of associated concurrency regulation architectures and present experimental data for an assessment of the different alternatives. A comparative discussion with literature alternatives ends the chapter.

2 The Common Base-ground

We overview concurrency-regulation approaches targeted at STM systems where the execution flow of each thread is characterized by the interleaving of transactional and non-transactional code blocks. During the execution of a transaction, the thread can perform read/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.

Typical STM-oriented concurrency-control algorithms [8] 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 (which also impacts 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 the concurrency regulation techniques presented in [6, 21, 22], which we are overseeing in this chapter.

In more details, these proposals aim at computing pairs of values $\langle w_{time,i}, i\rangle$ where i indicates the level of concurrency, namely the number of threads which are supposed to be used for executing the application, and $w_{time,i}$ is the expected transaction wasted time (when running with degree of concurrency equal to the value i). 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 the target system model), the system throughput when running with i threads can be computed as

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

By exploiting Equation (1), the objective of the concurrency regulation proposals in [6, 21, 22] is to identify the value of i, in the reference interval $[1, max\_threads]$, such that thr_i is maximized[1].

As we will see, $w_{time,i}$ is expressed in the different considered approaches 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

[1] Approaches to regulate concurrency typically rely on setting *max_threads* to the maximum number of CPU-cores available for hosting the STM application. This choice is motivated by the fact that using more threads than the available CPU-cores is typically unfavorable since the overhead caused by context-switches among the threads may become predominant [11]. Also, thread-reschedule latencies may further unfavor performance due to secondary effects related to increasing the so-called transaction vulnerability-window, namely the interval of time along which actions by concurrent transactions can ultimately lead to the abort of some ongoing transaction [18].

cache level, thus impacting the CPU time required for a specific code block, either transactional or non-transactional. To cope with this issue, once the value of t (or ntc) when running with k threads—which we denote as t_k and ntc_k respectively—is known, analytic correction functions are typically employed to predict the corresponding values when supposing a different number of threads. This yields the final throughput prediction (vs the concurrency level) to be expressed as:

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

where for $w_{time,i}$ we only point out the dependence on t_i and ntc_i, while we intentionally delay to the next sections the presentation of the other parameters playing a role in its expression. Overall, the finally achieved performance model in Equation (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 already pointed out, one core objective of the concurrency-regulation proposals that we are overseeing consists in modeling the system performance so as to capture the effects of variations of the application execution profile. This has been achieved by relying on a model of $w_{time,i}$ 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. The proposals in [6, 21, 22] all share the common view that capturing the combined effects of concurrency degree and execution profile on the transaction wasted time can be achieved in case $w_{time,i}$ is 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.

We note that the above parameters cover the set of workload-characterizing parameters that have been typically accounted for by performance studies of concurrency control protocols for traditional transactional systems, such as database systems (see, e.g., [24, 27]). In other words, the idea behind the above model 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.

The objective of the modeling approaches in [6, 21, 22] is to provide approximations of the function f via proper estimators. The first estimator we discuss, which we refer to as f_A, has been presented in [6] and is based on an analytic approach. The second one, which we refer to as f_{ML}, has been presented in [21] and relies on a pure Machine Learning (ML) approach. Finally, the third estimator, which we refer to as f_{AML}, has been presented in [22] and is based on a mixed approach combining analytic and ML techniques.

We refer the reader to the technical articles in [6, 21, 22] for all the details related to the derivation of these estimators, so that the following presentation is intended as an overview of each of the approaches, and as a means to discuss virtues and limitations of each individual solution. The discussion will be then backed by experimental data we shall report later on in this chapter.

3 The f_A Estimator

The solution presented in [6] tackles the issue of predicting the optimal concurrency level (and hence regulating concurrency) in STM via an analytic approach that differentiates from classical ones. Particularly, it relies on a *parametric analytic expression* capturing the expected trend in the transaction abort probability (versus the degree of concurrency) as a function of a set of features associated with the actual workload profile. The parameters appearing in the model exactly aim at capturing execution dynamics and effects that are hard to be expressed through classical (non-parametric) analytic modeling approaches (such as [5]), which typically make the latter reliable only in case the modeled system conforms the specific assumptions that underlie the analytic expressions.

Further, the parametric analytic model is thought to be easily customizable for a specific STM system by calculating the values to be assigned to the parameters (hence by instantiating the parameters) via regression analysis. One relevant virtue of this kind of solution is that the actual sampling phase, needed to provide the knowledge base for regression, can be very lightweight. Specifically, a very limited number of profiling samples, related to few different concurrency levels for the STM system, likely suffices for successful instantiation of the model parameters via regression.

The core analytical expression provided by the study in [6] is the one encapsulating the probability for a transaction to be aborted, namely p_a, which is built as a function of the parameters appearing in input to Equation (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 [6]:

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

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

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

where m, n, h, l, e, f, g, z, c, b, a, d are all fitting parameters to be instantiated via regression. In more details, regression analysis is performed by exploiting a set of sampling data gathered through run-time observations of the STM application. Each sample includes the average values of all the input parameters (independent variables) and of the abort probability (dependent variable) in Equation (4), measured over different time slices. Hence, Equation (5) is used as regression function, whose fitting parameters' values are estimated to be the ones that minimize the sum of squared residuals [3].

The abort probability expression, as provided by relying on Equations (4)–(8), has been exploited 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 transaction runs (per successful transaction commit).

4 The f_{ML} Estimator

The solution presented in [21] addresses the issue of concurrency regulation by a perspective that stands as different from the one in [6]. Particularly, this solution is based on a *pure* ML approach, whose general virtue is to provide an extremely precise representation of the target system behavior, provided that the training process is based on a sufficiently wide set of configurations, spanning many of the parameters potentially impacting this behavior. Generally speaking, good coverage of the domain typically guarantees higher accuracy of ML based models when compared to their analytic counterpart [20].

The exploited ML method in [21] is a Neural Network (NN) [20], which provides the ability to approximate various kinds of functions, including real-valued ones. Inspired by the neural structure of the human brain, a NN consists of a set of interconnected processing elements which cooperate to compute a specific function, so that, provided a given input, the NN can be used to calculate the output of the function. By relying on a learning algorithm, the NN can be trained to approximate an unknown function f exploiting a data set $\{(\mathbf{i},\mathbf{o})\}$ (training set), which is assumed to be a statistical representation of the function f such that, for each element $(\mathbf{i},\mathbf{o})$, $\mathbf{o} = f\{\mathbf{i}\} + \delta$, where δ is a random variable (also said *noise*). In [21], 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.

On the other hand, significant coverage of the domain of values for the above **input** parameters may require long training phases, imposing a delay in the optimization of the actual run-time behavior of the STM application. Overall, this ML based scheme might not fully fit scenarios where fast construction of application-specific performance models needs to be actuated in order to promptly optimize performance and resource usage (including energy). An example case is the one of dynamic deploy of applications in Cloud Computing environments.

5 The f_{AML} Estimator

The proposal in [22] is based on mixing analytic and ML techniques (hence AML) according to a scheme aimed at providing a performance prediction model f_{AML} showing the same capabilities (in terms of precision) as the ones offered by the ML approach, namely f_{ML}, but offering a reduced training latency, comparable to the one allowed by the pure parametric-analytic based approach f_A. In other words, the attempt in this proposal is to get the best of the two worlds, which is operatively achieved by a sequence of algorithmic steps performing the combination of f_A and f_{ML}.

A core aspect in this combination is the introduction of a new type of training set for the machine learning component f_{ML}, which has been referred 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 Equation (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 (**input**, **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 building the VTS requires 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:

(A) A number Z of different values of i are randomly selected in the domain $[1, max\_threads]$, and for each selected value of i, the application run-time behavior is observed by taking δ real-samples, each one including the set of parameters $\{rs, ws, rw, ww, t, ntc, i\} \cup \{tr\}$.

(B) Via regression all the fitting parameters requested by Equations (6)–(8) are instantiated. Hence, at this stage an instantiation of Equation (5), namely the model instance for p_a, has been achieved.

(C) The instantiated model for p_a is filled in input to Equation (9), together with the average value of tr sampled in step **A**, and then the VTS is generated. This is done by generating δ' virtual samples ($\mathbf{input^v}, \mathbf{output^v}$) where, for each of these samples, $\mathbf{input^v} = \{rs^v, ws^v, rw^v, ww^v, t^v, ntc^v, i^v\}$ and $\mathbf{output^v} = w_{time,i^v}$ as computed by the model in Equation (9). Each $\mathbf{input^v}$ sample is instantiated by randomly selecting the values of the parameters that compose it[2]. 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** (recall that all these parameters have anyhow non-negative values). 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, the proposal in [22] uses 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 the scenario where the curve learned by f_{ML} would correspond to the one modelled by f_A. Hence, in order to improve the quality of the machine learning based estimator, the actual combination of the

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

analytical and machine learning methods presented in [6, 21] relies on additional algorithmic steps where VTS is used 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 (according to proper rules aimed at avoiding 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 for the construction of VTS). 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.

Once achieved the final VRMTS image, it is used to train f_{ML} in order to determine the final AML estimator. Overall, f_{AML} is defined in [22] as the instance of f_{ML} trained via VRMTS.

6 Correcting Functions

As pointed out, the instantiation of the different estimators of the function f in Equation (3), which are ultimately aimed at predicting $w_{time,i}$, needs to be complemented with a predictor of how t and ntc are expected to vary vs the degree of parallelism i. In fact, $w_{time,i}$ is expressed in the various modeling approaches as a function of t and ntc. Further, the final equation establishing the system throughput, namely Equation (2), which is used for evaluating the optimal concurrency level by all the overseen proposals, also relies on the 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, one can 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 early samples taken in all the approaches for instantiating the performance models can be 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. The expectation is that the number of clock-cycles spent in waiting phases scales (almost) linearly vs the number of concurrent threads used for running the application. Hence, even if applied on a very limited number of samples, regression should suffice for reliable instantiation of the correction functions. To support this claim, we report in Figure 1 and in Figure 2 the variation of the clock-cycles spent while waiting for data to come from RAM for two different STM applications of the STAMP benchmark suite [19], namely intruder and vacation[3], 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

[3] The description of these (and other) STAMP benchmarks exploited in this chapter is postponed to Section 8.

Opteron^{TM}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. By the curves, the close-to-linear scaling is fairly evident, hence, once determined the scaling curve via regression, which we denote as *sc*, we let:

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

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).

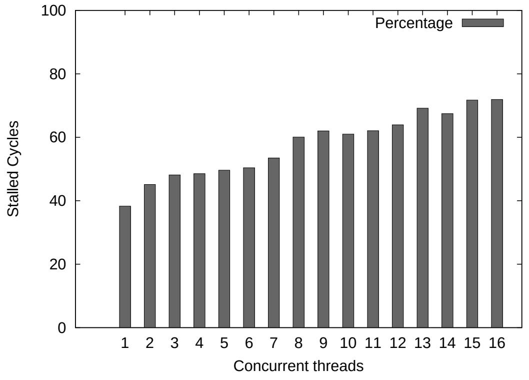

Fig. 1 Stalled cycles for the intruder benchmark

7 The Concurrency Regulation Architecture

Beyond providing the performance models and the concurrency regulation schemes, the works in [6, 21, 22] also provide guidelines for integrating concurrency regulation capabilities within operating STM environments. In this section we provide an overview of how the concurrency regulation architecture based on f_{AML}, selected as a reference instance, has been integrated with a native STM layer. Given that f_{AML} is the combination of the other two approaches, the architectures relying on the corre-

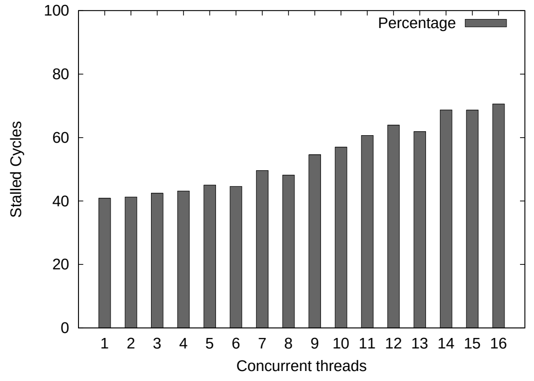

Fig. 2 Stalled cycles for the vacation benchmark

sponding two estimators f_A and f_{ML} can be simply derived by removing functional blocks from the one presented here.

The organization of the reference instance[4], which we name AML-STM, is shown in Figure 3. AML-STM is composed of the following three building blocks:

- A Statistics Collector (SC);
- A Model Instantiation Component (MIC);
- 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 the instantiation of the AML model[5]. 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. 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 Equation (2). This value is filled in input to CR (via queries by CR to MIC), which in its turn switches

[4] The source code of the actual implementation is freely available at `http://www.dis.uniroma1.it/~hpdcs/AML-STM.zip`. It exploits TinySTM [13] as the core STM layer.

[5] 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.

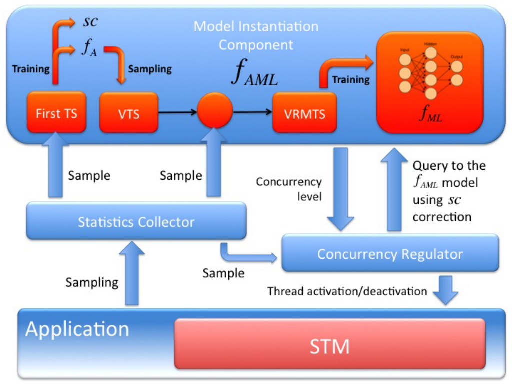

Fig. 3 System architecture

off or activates threads depending on whether the level of concurrency needs to be decreased or increased for the next observation period.

As noted above, in case the concurrency regulation architecture would have been based on f_A or f_{ML}, then the initial training set TS would have been directly used to instantiate f_A (as already shown in the picture), or f_{ML} (which could be achieved by simply collapsing VRMTS onto TS in the architectural organization). On the other hand, the training phase, namely the phase along which real samples of the application behavior are collected in order to instantiate the different estimators, would be of different length. We will provide data for a quantitative assessment of this aspect in the next section. We recall again that the shorter such a length, the more promptly the final performance model (based on a given estimator) to be used for concurrency regulation is available. Hence, a reduction in the length of this phase, 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.

8 Experimental Assessment

In this section we provide experimental data for a comparative assessment of the concurrency regulation techniques (and of the associated performance prediction models) we have overseen in this chapter. The experimentation has been based on applications belonging to the STAMP benchmark suite [19], which have been run on top of the aforementioned 16-cores HP ProLiant machine. Particularly, we focus the discussion on the results achieved with kmeans, yada, vacation, and intruder, which have been selected from the STAMP suite as representatives of a mix of applications with very different transactional profiles, as we shall describe below.

kmeans is a transactional implementation of a partition-based clustering algorithm [4]. A cluster is represented by the mean value of all the objects it contains, and during the execution of this benchmark the mean points are updated by assigning each object to its nearest cluster center, based on Euclidean distance. This benchmark relies on threads working on separate subsets of the data and uses transactions in order to assign portions of the workload and to store final results concerning the new centroid updates. The peculiarity of this benchmark lies in a very reduced amount of shared data structures being updated by transactions.

yada implements Ruppert's algorithm for Delaunay mesh refinement [23], which is a key step used for rendering graphics or to solve partial differential equations using the finite-element method. This benchmark discretizes a given domain of interest using triangles or thetraedra, by iteratively refining a coarse initial mesh. In particular, elements not satisfying quality constraints are identified, and replaced with new ones, which in turn might not satisfy the constraints as well, so that a new replacement phase must be undertaken. This benchmark shows a high level of intrinsic parallelism, due to the fact that elements which are distant in the mesh do not interfere with each other, and operations enclosed by transactions involve only updates of the shared mesh representation and cavity expansion. Also, transactions are relatively long.

intruder is an application which implements a signature-based network intrusion detection systems (NIDS) that scans network packets for matches against a known set of intrusion signatures. In particular, it emulates Design 5 of the NIDS described in [14]. Three analysis phases are carried on in parallel: *capture*, *reassembly*, and *detection*. The capture and reassembly phases are both enclosed by transactions, which are relatively short and show a contention level which is either moderate or high, depending on how often the reassembly phase re-balances its tree.

vacation implements a travel reservation system supported by a single-instance database, where tables are implemented as red-black trees. In the database, there are four different tables, each one representing cars, rooms, flights, and customers, respectively. The customers' table is used to keep track of the reservations made by each customer, along with the total price of the reservations they made. The other tables have relations with fields representing, e.g., reserved quantity, total available quantity, and price. In this benchmark several clients (concurrently) interact with the database, making actual reservations. Each client session is enclosed in a coarse-grain transaction to ensure validity of the database. Additionally, the amount of shared data touched by transactions is (on average) non-negligible.

Fixed the above applications as the test-bed, we initially focus on assessing the quality of the different performance prediction models we have overseen, hence of the different estimators of the function f in Equation (3). This is done by reporting how the error in predicting $w_{time,i}$ changes for the different estimators (f_A, f_{ML} and f_{AML}) with respect to the length of the sampling phase used to gather training data to instantiate each individual performance model. In other words, the focus is initially on determining how fast we can build a "reliable" model for performance estimation vs the level of concurrency in STM systems when considering the three different target methodologies (analytical, machine learning and mixed) in comparison with each other. To this end, 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 in the underlying computing platform, 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 have been used according to the following rule. The 10% of the initially taken samples in the observation interval are used to instantiate f_A (see steps **A** and **B** in Section 5), which is then used to build VTS, while the remaining 90% are used to derive VRMTS. 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.

Table 1 Comparison of error by different predictors/sampling times

		A	ML	AML
1 min	intruder	15.79%	80.04%	15.91%
	kmeans	5.82%	9.63%	2.66%
	vacation	6.08%	99.43%	6.19%
	yada	41.25%	99,82%	41.48%
5 mins	intruder	15.79%	80.04%	15.85%
	kmeans	5.90%	2.66%	2.59%
	vacation	4.93%	71.58%	5.01%
	yada	4.20%	13.24%	1.15%
10 mins	intruder	12.57%	45.01%	12.45%
	vacation	3.77%	3.31%	3.26%
	yada	4.20%	1.15%	1.16%
15 mins	intruder	11.46%	14.13%	8.84%
25 mins	intruder	10.00%	5.36%	5.35%

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 performance models and compared the errors they provide in predicting $w_{time,i}$. These error values are reported in Table 1, and refer to the average error while comparing predicted values with real execution values achieved while varying the number of threads running the 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.

By the data we can draw the following main conclusions. We cannot avoid relying on machine learning if extremely precise predictions of the level of performance vs the degree of concurrency are required. In fact, considering the asymptotic variation of the prediction error of $w_{time,i}$ (while increasing the length of the sampling phase used to build the knowledge base for instantiating the performance prediction models), the f_A estimator gives rise to an error which is on the order of 100% (or more) greater than the one provided by the other two estimators f_{ML} and f_{AML}. The machine learning technique would therefore look adequate for scenarios where the error in predicting the level of performance may have a severe impact on, e.g., some business process built on top of the STM system, such as when the need for guaranteeing predetermined Quality-of-Service levels by the transactional applications arises. However, we note that the sampling times reported in Table 1 for instantiating performance models offering specific levels of reliability have all been achieved for the case of pre-specified transactional profiles (e.g. a pre-specified mix of transactional operations), for which the domain of values for the parameters characterizing the actual workload are essentially known (or easily determinable). This has led to building adequate training sets allowing, e.g., good coverage of the whole domain along the sampling period, which would lead to kinds of best-case latencies for instantiating machine learning based schemes. On the other hand, in case the transactional profile of the application is not predetermined (as it may occur when deploying new applications, whose actual profile can be determined a-posteriori of the real usage by its clients), the length of the sampling phase for building the reliable machine learning based model can be significantly stretched, which may also negatively impact the overlying business process (e.g. because the application can be forced to run with sub-optimal concurrency levels for longer time due to the need for longer latencies for materializing good approximation and coverage of the actual domain during some on-line operated sampling phase). The role of the analytical component in coping with the reduction of the number of samples (hence the reduction of the coverage of the domain of values for the parameters determining the actual application workload) for the achievement of reliable predictions is clearly evident by the reported data. In particular, the f_{AML} estimator provides non-asymptotic results which outperform both the analytic approach and the pure machine learning approach (see, e.g., kmeans—5 minutes, yada–5 minutes, vacation–10 minutes, or intruder–15 minutes). This is exactly related to the fact that f_{AML} is able to get benefits from both prediction methods, and is therefore able to provide a faster convergence to the "optimal" estimator.

As a second assessment, we provide experimental data related to the runtime performance that can be achieved when relying on concurrency regulation architectures based on the different performance models we are comparing (which we refer to as A-STM, ML-STM and AML-STM). As a matter of fact, this part of the assessment provides hints on whether (and to what extent) concurrency regulation, operated according to each of the discussed approaches, can be effective. Also, we study the actual performance delivered by the different solutions while again varying the length of the sampling phase along which the knowledge base for instantiating the different performance models is built, which we refer to as *model instantiation time* in the reported graphs. The concurrency regulation architectures here considered adhere to the architectural organization depicted in Section 7 and all rely on TinySTM as their core STM layer. The experimental data we provide refer again to the four STAMP benchmark applications as before, namely intruder, kmeans, vacation, and yada. In Figures 4–7 we report plots showing how the throughput provided by the different solutions (which is expressed in terms of committed transactions/second, on the average run) varies vs the model instantiation time. We also report the throughput values obtained when running with plain TinySTM (i.e. with no concurrency regulation scheme) or sequentially, which will be used as baselines in the discussion. Clearly, these data appear as flat curves, given that they do not depend on any performance model to be instantiated along time via application sampling.

By the data we can draw the following main conclusions. First, (dynamically) controlling the level of concurrency is a first class approach to achieve speedup as compared to the case where all the operations are processed sequentially along a single thread. In fact, settings where the level of concurrency is simply determined by the number of available CPU-cores (namely by deploying a single thread per CPU-core), as for the case of plain TinySTM, do not provide significant speedup, and may even give rise to significant slow down in the execution speed (of committed work), as for the case of yada (see Figure 7). Further, a machine learning based performance model gives rise to the asymptotically optimal approach for concurrency regulation, while analytical techniques provide the orthogonal advantage of allowing faster instantiation of an "adequate" performance model to be employed for concurrency regulation purposes. However, the additional information convoyed by the reported plots is the quantification of the final (asymptotic) performance gain achievable thanks to the increased precision by machine learning based approaches (such as ML-STM or AML-STM), which is on the order of up to 30% as compared to the analytical approach (say A-STM).

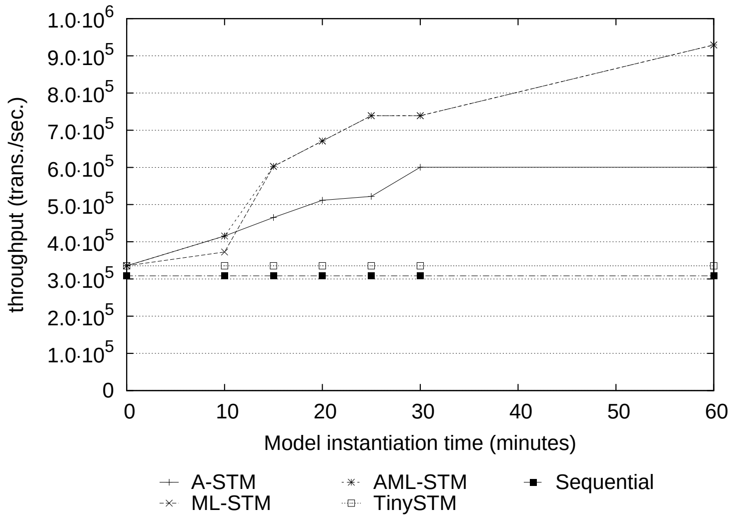

Fig. 4 Throughput – intruder

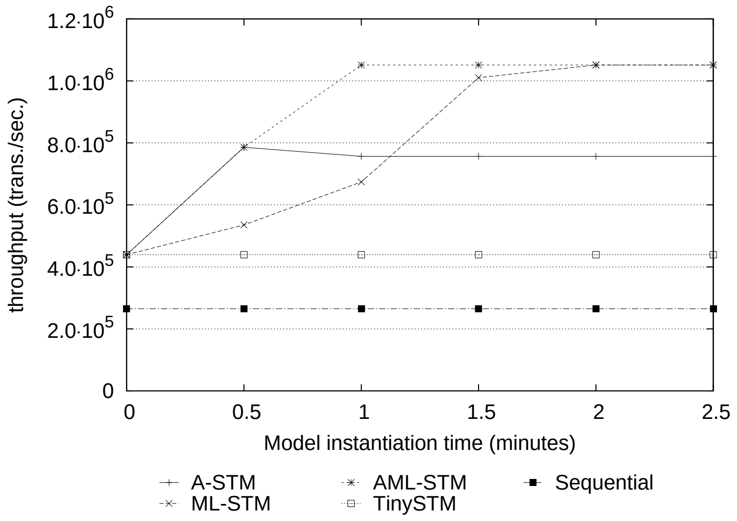

Fig. 5 Throughput – kmeans

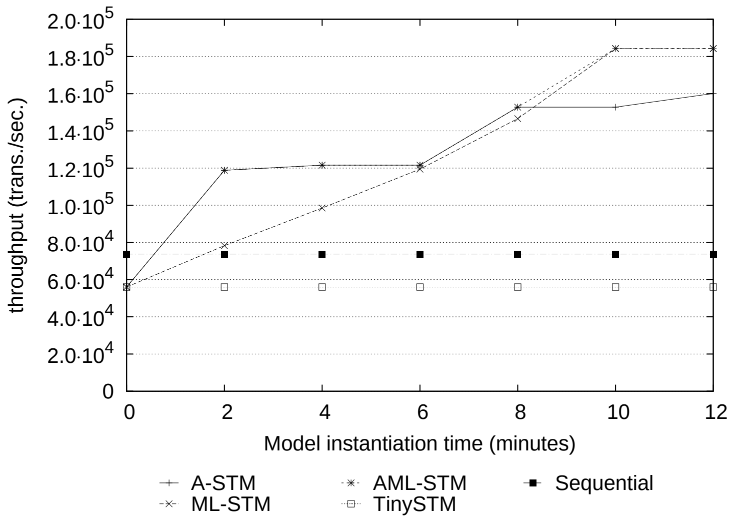

Fig. 6 Throughput – vacation

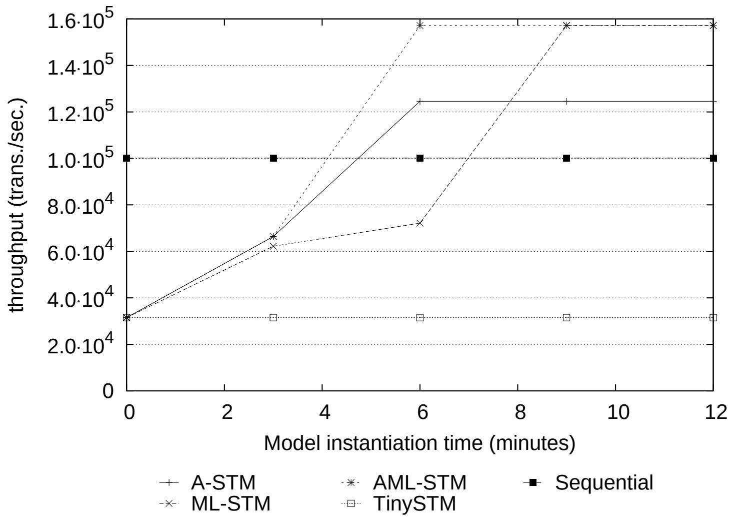

Fig. 7 Throughput – yada

The last aspect we would like to point in this experimental assessment relates to energy efficiency, and its improvement thanks to concurrency regulation. As for this aspect, we focus on kmeans given that it is more likely to incur logical contention (hence transaction aborts and unfruitful usage of energy for rolled back work) when a larger number of threads is used. Hence, the energy saving via concurrency regulation (e.g. vs the TinySTM baseline) with this benchmark likely represents a kind of lower bound on the saving that we may expect with the other benchmarks.

In Figure 8 we report measurements related to per-transaction energy consumption (in Joule/Transaction)—which is an index of how much power is required by the application to successfully complete the execution of a single transaction—again while varying the model instantiation time. By the results we note first of all that the configuration exhibiting the lowest energy consumption is the sequential one. This is clearly due to the fact that in a sequential execution no operation is aborted, and therefore the amount of energy used on average per each operation is exactly the one strictly required for carrying on the associated work. Nevertheless, this configuration exploits no parallelism at all. On the other hand, AML-STM and ML-STM asymptotically show the same energy consumption. At the same time, we note that AML-STM and A-STM give rise to comparable (but non-minimal) energy consumption in case of very reduced model instantiation times (say on the order of 20 secs).

To provide more insights into the relation between speed and usage of energy, we report in Figure 9 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

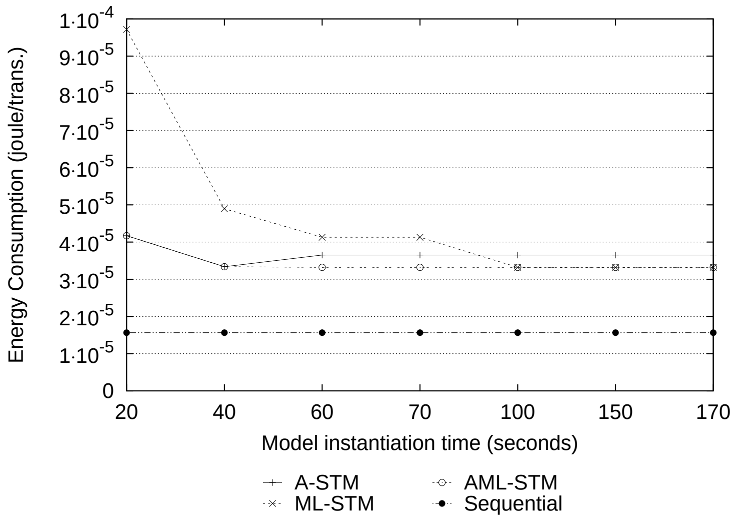

Fig. 8 Energy consumption per committed transaction – kmeans

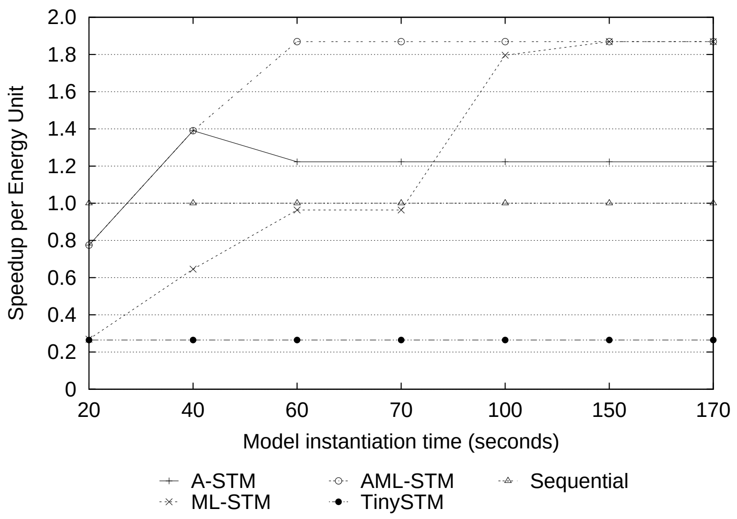

Fig. 9 Iso-energy speedup – kmeans

sequential run of the application). Essentially, the curves in Figure 9 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 AML-STM 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 may degrade both performance and energy efficiency.

9 A Look at Literature Alternatives

Other studies exist in literature coping with predicting/identifying the optimal level of concurrency in STM systems and (possibly) dynamically regulating this level while the application is in progress. We can classify them in two categories, for each of which recent achievements are described in what follows.

Model-Based Approaches. In this category we include all the solutions where the prediction of how the STM system performance scales vs the level of concurrency (and thus the identification of the optimal level of concurrency) is based on the a-priori construction of a performance model. Along this path we find the work in [5], where 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. The actual target of this proposal is to build mathematical tools allowing the analysis of the effects of the contention management scheme on performance while the concurrency level varies. For this reason a detailed knowledge of the specific conflict detection and management scheme used by the target STM is required, and needs to be dealt with by a specialized modeling scheme capturing its dynamics. The proposed analytical model is in fact build up by coupling two building block sub-models: one independent of the actual concurrency control scheme, and another one which is instead specific to a given concurrency control algorithm. The latter has been instantiated in the work in [5] for the case of the Commit-Time-Locking (CTL) algorithm, and cannot be directly reused for algorithms based on different rules. Further, the model globally relies on assumptions to be met by the real STM system (e.g. in terms of data access pattern) in order for it to provide reliable predictions. In other words, this solution stands as kind of *scenario specific* approach.

The work in [15] 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. This model is able to predict 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 (as expressed by the workload characterization). In other words, given an application, a unique "optimal" concurrency level can be identified via this approach, the most suited one for coping with situations where the application would behave according to expected values of the parameters determining the actual workload. In case of employment of this model in a real concurrency regulation architecture, the binding to the average system behavior would reduce the ability to capture the need for readapting the concurrency level on the basis of run-time variations of the application transactional profile in the different phases of its execution.

The proposal in [10] 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 numbers 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 (namely the performance model) used to predict the speed-up of the application vs the number of threads. In this approach the workload profile of the application is not taken into account, hence the prediction may prove unreliable when the profile changes wrt the one characterizing the behavior of the application during measurement and interpolation phases. Variance, or shifts, in the profile due to changes in the data-set content (possibly giving rise to, e.g., changes in the read/write set size) are therefore not captured by this kind of approach, and hence cannot be dealt with in terms of dynamic re-tuning of the level of concurrency in case of their materialization.

Heuristic Methods. In this category we find solutions that do not rely on a-priori constructing any model expressing the variation of performance vs the level of concurrency. The idea underlying these proposals is to try to push the system to its "optimal" performance level without building/relying on any knowledge base on how the level of performance would actually vary when chancing the number of threads. In this category we find the proposal in [2], which presents a control algorithm that dynamically changes the number of threads concurrently executing transactions on the basis of the observed transaction conflict rate. It is decreased when the rate exceeds some threshold value while it is increased when the rate is lower than another threshold. Another proposal along this direction can be found in [9], where 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. 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.

We also report in Table 2 a summary comparison of the approaches we have overseen in this chapter with literature alternatives. It is based on five indexes we identify as relevant, which are related to either the extent to which each approach is widely applicable, or its operating mode.

Table 2 Comparison of the different approaches

Approach	**Suitable for any conflict manager**	**Bound to a given Tx profile**	**Explicitly captures variations of Tx profiles**	**Initial training required**	**Reduced training latency**
f_A / A-STM	✓	✗	✓	✓	✓
f_{ML} / ML-STM	✓	✗	✓	✓	✗
f_{AML} / AML-STM	✓	✗	✓	✓	✓
[5]	✗	✓	✗	✓	✗
[15]	✓	✗	✗	✓	✓
[10]	✓	✗	✗	✓	✓
[2]	✓	✗	✗	✗	–
[9]	✓	✗	✗	✗	–

References

1. Cloud-TM: a Novel Programming Paradigm for the Cloud. http://http://www.cloudtm.eu/
2. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Proceedings of the 14th international Euro-Par conference on Parallel Processing. pp. 719–728. Euro-Par, Springer-Verlag (2008)
3. Bates, D., Watts, D.: Nonlinear regression analysis and its applications. Wiley series in probability and mathematical statistics, Wiley, New York [u.a.] (1988)
4. Bezdek, J.C.: Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell, MA, USA (1981)
5. Di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., Romano, P.: 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)
6. Di Sanzo, P., Del Re, F., Rughetti, D., Ciciani, B., Quaglia, F.: Regulating concurrency in software transactional memory: An effective model-based approach. In: Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems. SASO, IEEE Computer Society (Sep 2013)
7. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proceedings of the 20th International Symposium on Distributed Computing. pp. 194–208. ACM, New York, NY, USA (2006)
8. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proceedings of the 20th International Symposium on Distributed Computing. pp. 194–208. ACM, New York, NY, USA (2006)

9. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Identifying the optimal level of parallelism in transactional memory applications. In: NETYS. pp. 233–247. Lecture Notes in Computer Science, Springer (2013)
10. Dragojević, A., Guerraoui, R.: Predicting the scalability of an STM: A pragmatic approach. In: Presented at: 5th ACM SIGPLAN Workshop on Transactional Computing (2010)
11. Ennals, R.: Software transactional memory should not be obstruction-free. Tech. rep., Intel Research Cambridge Tech Report (Jan 2006)
12. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 237–246. PPoPP, ACM (2008)
13. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming. pp. 237–246. ACM, New York, NY, USA (2008)
14. Haagdorens, B., Vermeiren, T., Goossens, M.: Improving the performance of signature-based network intrusion detection sensors by multi-threading. In: Proceedings of the 5th International Conference on Information Security Applications. pp. 188–203. WISA, Springer-Verlag (2005)
15. He, Z., Hong, B.: 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. pp. 307–315. IEEE Computer Society, Washington, DC, USA (2010)
16. Herlihy, M.P., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. ACM SIGARCH Computer Architecture News 21(2), 289–300 (May 1993)
17. Lev, Y., Luchangco, V., Marathe, V.J., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing. TRANSACT, ACM (2009)
18. Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J.L., Muller, G.: Scheduling support for transactional memory contention management. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 79–90. PPOPP (2010)
19. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Proceedings of the IEEE International Symposium on Workload Characterization. pp. 35–46. IEEE Computer Society, Washington, DC, USA (2008)
20. Mitchell, T.M.: Machine Learning. McGraw-Hill, 1 edn. (1997)
21. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: 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. pp. 278–285. MASCOTS, IEEE Comp. Soc. (Aug 2012)
22. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: Analytical/ML mixed approach for concurrency regulation in software transactional memory. In: Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. CCGrid, IEEE Comp. Soc. (Aug 2014)
23. Ruppert, J.: A delaunay refinement algorithm for quality 2-dimensional mesh generation. Journal of Algorithms 18(3), 548–585 (1995)
24. di Sanzo, P., Ciciani, B., Quaglia, F., Romano, P.: A performance model of multi-version concurrency control. In: Proceedings of the 16th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. pp. 41–50. MASCOTS (2008)
25. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming. pp. 141–150. ACM, New York, NY, USA (Feb 2009)
26. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. pp. 169–178. SPAA, ACM (2008)

27. Yu, P.S., Dias, D.M., Lavenberg, S.S.: On the analytical modeling of database concurrency control. Journal of the ACM pp. 831–872 (1993)