# Regulating Concurrency in Software Transactional Memory: An Effective Model-based Approach

Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani, Francesco Quaglia
*DIAG, Sapienza University of Rome*

***Abstract*—Software Transactional Memory (STM) is recognized as an effective programming paradigm for concurrent applications. On the other hand, a core problem to cope with in STM deals with (dynamically) regulating the degree of concurrency, in order to deliver optimal performance. We address this problem by proposing a self-regulation approach of the concurrency level, which relies on a parametric analytical performance model aimed at predicting the scalability of the STM application as a function of the actual workload profile. The regulation scheme allows achieving optimal performance during the whole lifetime of the application via dynamic change of the number of concurrent threads according to the predictions by the model. The latter is customized for a specific application/platform through regression analysis, which is based on a lightweight sampling phase. We also present a real implementation of the model-based concurrency self-regulation architecture integrated within the open source TinySTM framework, and an experimental study based on standard STM benchmark applications.**

## I. Introduction

Software Transactional Memory (STM) [17] has emerged as a promising paradigm aiming at simplifying the development of parallel/concurrent applications. By relying on the concept of atomic transaction, STM represents a friendly alternative to traditional lock-based synchronization. More in detail, code blocks accessing shared data can be marked as transactions, thus demanding coherency of the data access/manipulation to the STM layer, rather than to any handcrafted synchronization scheme. The relevance of the STM paradigm has significantly grown given that multi-core systems have become mainstream platforms, so that even entry-level desktop and laptop machines are nowadays equipped with multiple processors and/or CPU-cores.

Even though one main target for STM is the simplification of the software development process, another aspect that is central for the success of the STM paradigm relates to the actual level of performance it can deliver. As for this aspect, STM needs to be complemented by schemes aimed at allowing the overlying application to reach optimal speedup values thanks to fruitful parallelism exploitation. This issue arises since STM applications are prone to thrashing phenomena (caused by excessive transaction rollbacks) in case the data access pattern tends to exhibit non-negligible conflict among concurrent transactions and the degree of parallelism in the execution is excessively high. On the other hand, for too low parallelism levels, the achievable speedup may still be suboptimal.

Recent approaches coping with this problem have been targeted at selecting/controlling the degree of parallelism by (dynamically) determining the well suited number of concurrent threads to sustain application execution. Along this path we can find solutions ranging from analytical models [13], [4], to heuristic-based schemes [7], to machine learning approaches [14]. On the other hand, all of the proposed approaches exhibit some shortcoming. Classical analytical approaches are in fact know to become unreliable as soon as the assumptions they rely on (e.g. in terms of data access distribution and/or distribution of the CPU time for specific operations) are not met. Further, according to the outcomes in [6], the transaction abort rate can be strongly affected by the order according to which data are accessed along the transaction execution path, which is typically neglected by analytical models. On the other hand, even in case the effects of such an ordering are captured analytically, the actual exploitation of the performance model would require detailed knowledge of the data access pattern for the specific application, which may be unavailable or arduous to build. As for heuristic and/or machine learning approaches, they do not require specific (stringent) assumptions to be met in relation to, e.g., the transactional profile of the application. Hence, they exhibit the potential for high effectiveness in generic application contexts, and for generic computing platforms. On the other hand, these approaches may show limited extrapolation capabilities, thus not being fully suited for forecasting the performance that would be achieved with levels of concurrency not belonging to the already explored domain (e.g. the training domain in case of neural network based approaches). Further, the time required for building the knowledge base to be exploited by the machine learner may be non-minimal, which may make the actuation of the optimized concurrency configuration untimely.

In this article we tackle the issue of regulating the concurrency level in STM via a model-based approach, which differentiates from classical ones in that it avoids the need for the STM system to meet specific assumptions (e.g. in terms of data access pattern). Our proposal relies on a parametric analytical expression capturing the expected

This work has been partially supported by the Cloud-TM project (co-financed by the European Commission through the contract no. 57784) and by COST Action IC1001 EuroTM. The software produced as a result of the research activity presented in this paper can be downloaded at the URL http://www.dis.uniroma1.it/∼hpdcs/CSR-STM.tar

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 within the model exactly aim at capturing execution dynamics and effects that are hard to be expressed through classical (non-parametric) analytical modeling approaches. We derived the parametric expression of the transaction abort probability via combined exploitation of literature results in the field of analytical modeling and a simulation-based analysis. Further, the parametric 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. The latter can be performed by exploiting a set of sampling data gathered through run-time observations of the STM application. However, differently from what happens for the training process in machine learning approaches, the actual sampling phase (needed to provide the knowledge base for regression in our approach) is very light. Specifically, a very limited number of profiling samples, related to a few different concurrency levels for the STM system, likely suffices for successful instantiation of the model parameters via regression. Finally, our approach inherits the extrapolation capabilities proper of pure analytical models (although it does not require their typical stringent assumptions to be met, as already pointed out), hence allowing reliable performance forecast even for concurrency levels standing distant from the ones for which sampling was actuated.

A bunch of experimental results achieved by running the STAMP benchmark suite [2] on top of the TinySTM open source framework [11] are reported for validating the proposed modeling approach. Further, we present the implementation of a concurrency self-regulating STM, exploiting the proposed performance model, still relying on TinySTM as the core STM layer, and we report experimental data for an assessment of this architecture.

The remainder of this paper is organized as follows. In Section II, literature results related to our proposal are discussed. Section III is devoted to describing and validating our STM performance model. The STM architecture entailing self-regulation capabilities of the concurrency level is presented and evaluated in Section IV.

## II. Related Work

Our proposal has relations with literature results in the field of analytical modeling of concurrency control protocols for transactional systems. These include performance models for traditional database systems and related concurrency control mechanisms (see, e.g., [18], [22]) and approaches specifically targeting STM (see, e.g., [4]). Some of the literature analytical models rely on (stringent) assumptions on the transaction data access pattern, such as uniformly distributed accesses (e.g. [22], [10]) or the *b-c* access model (e.g. [18], [19]). Differently from all these works, our proposal does not assume any specific distribution for the data accesses, thus being more general and exploitable in generic application contexts. Other literature models are able to capture more complex data access patterns by assuming Zipf-distributed accesses [5] or phase-based accesses [6]. Compared to these solutions, our proposal avoids the need for any detailed characterization of the data access distribution. As a reflection, the instantiation of the parameters appearing in our model requires a lighter application sampling process than what required to instantiate the actual data access distribution.

In [8] the authors propose a technique to approximate the performance of the STM application when considering different amounts of concurrent threads. The technique is based on the usage of different types of functions, such as polynomial, rational and logarithmic functions. The approximation process relies on sampling the speed-up of the application over a set of runs, each one executed with a different number of concurrent threads. After, the speed-up forecasting function is instantiated by interpolating the measurements. Compared to our proposal, a limitation of this approach lies on that the workload profile of the application is not taken into account while instantiating the performance forecasting function. This may lead to reduced reliability of the forecasting outcome, especially when the workload profile of the application changes.

As for machine learning, it has been used in [20] for selecting the best performing conflict detection and management algorithm. Conversely, it has been used in [3] to select suitable mappings of threads to CPU-cores, allowing performance improvements thanks to increased effectiveness of the caching system. The goal of both these works is different and orthogonal with respect to our one since we focus on the regulation of the overall concurrency level in the STM system. To the best of our knowledge, the only machine learning based approach targeting this same problem has been presented in [14]. Compared to this solution, our proposal relies on a sampling process that is lighter than the one required for building the machine learning based performance model via training.

In [7] a black-box approach is proposed, based on the hill-climbing heuristic scheme, which dynamically increases or decreases the level of concurrency. Particularly, 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. Differently from our proposal, no direct attempt to capture the relation between the actual transaction profile and the achievable performance (depending on the level of parallelism) is done.

Given that our model-based approach is ultimately aimed at regulating concurrency so to avoid thrashing phenomena, our proposal is related to pro-active transaction scheduling schemes, which cope with the issue of performance degradation due to excessive data contention [1], [21], [9]. These solutions avoid scheduling the execution of transactions whose associated conflict probability is estimated to be high. The work in [1] presents a control algorithm that dynamically changes the number of threads concurrently executing transactions on the basis of the observed transaction conflict rate (by decreasing/increasing the level of concurrency when the conflict rate exceeds/undergoes some threshold). In [21], incoming transactions are enqueued and sequentialized when an indicator, referred to as *contention-intensity* (calculated as a dynamic average depending on the number of aborted vs committed transactions), exceeds a pre-determined threshold. In [9], a transaction is sequentialized when a potential conflict with other running transactions is predicted. The prediction relies on the estimation of the expected transaction read-set and write-set. The sequentializing mechanism is activated only when the amount of aborted vs committed transactions exceeds a given threshold. Compared to our model-based approach, all the above proposals do not directly estimate the likelihood of transaction aborts as a function of the level of concurrency. Rather, they attempt to control the wasted time in an indirect manner via heuristics.

## III. The Parametric Performance Model

As already hinted, we decided to exploit a model relying on a parametric analytical expression which captures the expected trend of the transaction abort probability as a function of (1) a set of features characterizing the current workload profile, and (2) the number of concurrent threads sustaining the STM application. The parameters in the analytical expression aim at capturing effects that are hard to express through a classical (non-parametric) analytical modeling approach. Further, they are exploited to customize the model for a specific STM application through regression analysis, which is done by exploiting a set of sampling data gathered through run-time observations of the application. In the remainder of this section we provide the basic assumptions on the behavior of the STM application, which are exploited while building the parametric analytical model. Then the actual construction of the model is presented, together with a model validation study.

### A. Basic Assumptions

The STM application is assumed to be run with a number $k$ of concurrent threads. The execution flow of each thread is characterized by the interleaving of transactions and non-transactional code ($ntc$) blocks. This is the typical structure for common STM applications, which also reflects the one of widely diffused STM benchmarks (see, e.g., [2]). The transaction read-set (write-set) is the set of shared data-objects that are read (written) by the thread while running a transaction. If a conflict between two concurrent transactions occurs, then one of the conflicting transactions is aborted and re-started (which leads to a new transaction run). After the thread commits a transaction, it executes a $ntc$ block, which ends before the execution of the begin operation of the subsequent transaction along the same thread.

### B. Model Construction

The set $P$ of features exploited for the construction of the parametric analytical model, which are used to capture the workload profile, consists of:

- the average size of the transaction read-set $rs_s$;
- the average size of the transaction write-set $ws_s$;
- the average execution time $t_t$ of committed transaction runs (i.e. the average duration of transaction runs that are not aborted);
- the average execution time $t_{ntc}$ of $ntc$ blocks;
- the read/write affinity $rw_a$, namely the probability that an object read by a transaction is also written by other transactions;
- the write/write affinity $ww_a$, namely the probability that an object written by a transaction is also written by other transactions.

Operatively, $rw_a$ 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, $ww_a$ can be calculated as the dot product between the distribution of write operations and itself.

Our parametric analytical model expresses the transaction abort probability $p_a$ as a function of the features belonging to the set $P$, and the number $k$ of concurrent treads supposed to run the STM application. Specifically, it instantiates (in a parametric manner) the function

$$p_a = f(rs_s, ws_s, rw_a, ww_a, t_t, t_{ntc}, k) \tag{1}$$

Leveraging literature models proposing approximated performance analysis for transaction processing systems (see [22], [16]), we express the transaction abort probability $p_a$ through the function

$$p_a = 1 - e^{-\alpha} \tag{2}$$

However, while in literature results the parameter $\alpha$ is expressed as the multiplication of parameters directly representing, e.g., the data access pattern and the workload intensity (such as the transaction arrival rate $\lambda$ for the case of open systems), in our approach we express $\alpha$ as the multiplication of different functions that depend on the set of features appearing in equation (1). Overall, our expression for $p_a$ is structured as follows

$$p_a = 1 - e^{-\rho \cdot \omega \cdot \phi} \tag{3}$$

where the function $\rho$ is assumed to depend on the input parameters $rs_s$, $ws_s$, $rw_a$ and $ww_a$, the function $\omega$ is assumed to depend on the parameter $k$, and the function $\phi$ is assumed to depend on the parameters $t_t$ and $t_{ntc}$.

We note that equation (2) has been derived in literature while modeling the abort probability for the case optimistic concurrency control schemes, where transactions are aborted (and restarted) right upon conflict detection. Consequently, this expression for $p_a$ and the variation we propose in equation (3) are expected to well match the STM context, where pessimistic concurrency control schemes (where transactions can experience lengthy lock-wait phases upon conflicting) are not used since they would limit the exploitation of parallelism in the underlying architecture. More specifically, in typical STM implementations (see, e.g., [11]), transactions are immediately aborted right upon executing an invalid read operation. Further, they are aborted on write-lock conflicts either immediately or after a very short wait-time.

The model we propose in equation (3) is parametric thanks to expressing $\alpha$ as the multiplication of parametric functions that depend on a simple and concise representation of the workload profile (via the features in the set $P$) and on the level of parallelism. This provides it with the ability to capture variations of the abort probability (e.g. vs the degree of parallelism) for differentiated application profiles. Particularly, different applications may exhibit similar values for the featuring parameters in the set $P$, but may anyhow exhibit different dynamics, leading to a different curve for $p_a$ while varying the degree of parallelism. This is catchable by our model via application-specific instantiation of the parameters characterizing the functions $\rho$, $\omega$ and $\phi$, which can be done through regression analysis. In the next section we discuss how we have derived the actual $\rho$, $\omega$ and $\phi$ functions, hence the actual function expressing $\alpha$.

### *C. Instantiating $\rho$, $\omega$ and $\phi$*

The shape of the functions $\rho$, $\omega$ and $\phi$ determining $\alpha$ is derived in our approach by exploiting the results of a simulation study. We decided to rely on simulation, rather than using measurements from real systems, since our model is aimed at capturing the effects associated with data contention on the abort probability, while it is not targeted at capturing the effects of thread-contention on hardware resources. Consequently, the instantiation of the functions appearing within the model has been based on an "ideal hardware" simulation model showing no contention effects. Anyway, when exploiting our data contention model for concurrency regulation in a real system, a hardware scalability model (e.g. a queuing network-based model) can be used to estimate variations of the processing time, due to contention effects on shared hardware resources, as a function of the number of the concurrent threads. In the final part of this paper, we provide some results that have been achieved by exactly using our data contention model and a hardware scalability model in a joint fashion.

The simulation framework we have exploited in this study is the same used in [4] for validating an analytical performance model for STM. It relies on the discrete-event paradigm, and the implemented model simulates a closed system with $k$ concurrent threads, each one alternating the execution of transactions and *ntc* blocks. The simulated concurrency control algorithm is the default algorithm of TinySTM (encounter time locking for write operations and timestamp-based read validation). A transaction starts with a *begin* operation, then it interleaves the execution of read/write operations (accessing a set of shared data objects) and local computation phases, and, finally, executes a *commit* operation. The durations of *ntc* blocks, transactional operations and local computation phases are exponentially distributed.

In the simulation runs we performed to derive and validate the expression of $\alpha$, we varied $rs_s$ and $ws_s$ between 0 and 200, $rw_a$ and $ww_a$ between $25 \cdot 10^{-6}$ and 0.01, $t_t$ between 10 and 150 $\mu$sec, and $t_{ntc}$ between 0 and $15{\cdot}10^4$ $\mu$sec. These intervals include values that are typical for the execution of STM benchmarks such as [2], hence being representative of workload features that can be expected in real execution contexts. Further, we varied $k$ between 2 and 64 in the simulations. Due to space constraints, we omit to explicitly show all the achieved simulation results. However, the shown results are a significative, although concise, representation of the whole set of achieved results.

The construction of the analytical expressions for $\rho$, $\omega$ and $\phi$ has been based on an incremental approach. Particulary, we first derive the expression of $\rho$ analyzing simulation results while varying workload configuration parameters affecting it, i.e. $rs_s$, $ws_s$, $rw_a$, $ww_a$, and keeping fixed other parameters. After, we calculate the values of $\rho$ from the ones achieved for $p_a$ via simulation, which is done by using the inverse function $\rho = f^{-1}(p_a)$, once set $\omega = 1$ and $\phi = 1$. After having identified a parametric fitting function for $\rho$, we derive the expression of $\omega$ via the analysis of the simulation results achieved while also varying $k$. Hence, we calculate $\omega = f^{-1}(p_a)$, where we use for $\rho$ the previously identified expression, and where we set $\phi = 1$. Therefore, we select a parametric fitting function for $\omega$. Finally, we use the same approach to derive the expression of $\phi$, which is done by exploiting the simulation results achieved while varying all the workload profile parameters and the level of concurrency $k$, thus calculating $\phi = f^{-1}(p_a)$, where we use for $\rho$ and $\omega$ the previously chosen expressions.

To derive the expression of $\rho$, we initially analyzed via simulation the relation between the values of $p_a$ and the values of the parameters $ws_s$ and $ww_a$. In Figure 1 we provide some results showing the values of $\rho$ as calculated through the $f^{-1}(p_a)$ inverse function (like depicted above) by relying on simulation data as the input. The data refer to variations of $ww_a$ and to 3 different values of $ws_s$, while all the other parameters have been kept fixed. We note that $\rho$ appears to have a logarithmic shape. Additionally, in order to chose a parametric function fitting the calculated values of $\rho$, we need to consider that if $ww_a = 0$ then $p_a = 0$. In fact, no data contention ever arises in case of no write operations within the transactional profile (which implies $\rho = 0$). Thus, we approximated the dependency of $\rho$ on $ww_a$ through the following parametric logarithmic function

$$c \cdot ln(a \cdot ww_a + 1) \tag{4}$$

where $a$ and $c$ are the fitting parameters. The presence of the $+1$ term in expression (4) is due to the above-mentioned constraint according to which $ww_a = 0$ implies $\rho = 0$.

After, we also considered the effects of the parameter $ws_s$ on $\rho$. To this aim, in Figure 2 we report the values of $\rho$, derived from the simulation results, while varying $ws_s$ and for 3 different values of $ww_a$. We remark the presence of a flex point. Therefore, in this case, we approximated the dependency of $\rho$ on $ws_s$ by using the function

$$e \cdot (ln(b \cdot ws_s + 1))^d \tag{5}$$

where $b$, $d$ and $e$ are fitting parameters, $d$ being the one capturing the flex. Assuming that the effects on the transaction abort probability are multiplicative with respect to $ww_a$ and $ws_s$ (which is aligned to what literature models state in term of the proportionality of the abort probability wrt the multiplication of the conflict probability and the number of operations, see, e.g., [22]), we achieved the following parametric expression of $\rho$ (vs $ww_a$ and $ws_s$), where $d$ has been used as the exponent also for expression (4) in order to capture the effects of shifts of the flex point caused by variations of $ww_a$ (as shown by the plots in Figure 2 relying on simulation)

$$[c \cdot (ln(b \cdot ws_s + 1)) \cdot ln(a \cdot ww_a + 1)]^d \tag{6}$$

where we collapsed the original parameters $c$ and $e$ within one single parameters $c$. We validated the accuracy of the expression (6) via comparison with values achieved through a set of simulations, where we used different workload profiles. The parameters appearing in expression (6) have been calculated through regression analysis. Specifically, for each test, we based the regression analysis on 40 randomly selected workload profiles achieved while varying $ww_a$ and $ws_s$. Then, we measured the average error between the transaction abort probability evaluated via simulation and the one predicted using for $\rho$ the function in expression (6) for a set of 80 randomly selected workload profiles. As an example, in Figure 6, we depict results for the case with $k = 8$. Along the x-axis, workload profiles are identify by integer numbers and are ordered by the values of $ws_s$ and $ww_a$. The measured average error in all the tests was 5.3%.

Successively, we considered the effects on the transaction abort probability caused by read operations. Thus, we analyzed the relation between $p_a$ and the parameters $rs_s$, $rw_a$ and $ws_s$. The parameter $ws_s$ is included since contention on transactional read operations is affected by the amount of write operations by concurrent transactions. In Figure 4 we report simulation results showing the values of $\rho$ while varying $rs_s$ and for 3 different values of $rw_a$. In Figure 5, we report values of $\rho$ achieved while varying $rw_a$ and for 3 different values of $rs_s$. We note that the shape of the curves are similar to the above cases, where we analyzed the relation between $p_a$ and the parameters $ww_a$ and $ws_s$. Thus, using a similar approach, and considering that $p_a$ is also proportional to $ws_s$, we approximate the dependency of $\rho$ on $rw_a$, $ws_s$ and $ww_a$ using the following function

$$[e \cdot (ln(f \cdot rw_a + 1)) \cdot ln(g \cdot rs_s + 1) \cdot ws_s]^z \tag{7}$$

where $e$, $f$, $g$ and $z$ are the fitting parameters. The final expression for $\rho$ is then derived summing expressions (6) and (7). The intuitive motivation is that adding read operations within a transaction, the likelihood of abort due to conflicts on original write operations does not change. However, the added operations lead to an increase of the overall abort probability, which we capture summing the two expressions. Also in this case, we validated the final expression for $\rho$ via comparison with the values achieved through a set of simulations, where we varied the workload profile. Similarly to what done before, the regression analysis has been based on 40 workload profiles, while the comparison has been based on 80 workload profiles, all selected by randomly varying $ww_a$, $ws_s$, $rw_a$, $rs_s$. The results for $k = 8$ are reported in Figure 6. Along the x-axis, workload profiles are ordered by values of $rs_s$, $rw_a$, $ws_s$ and $ww_a$. The average error we measured in all the tests was 2.7%.

Successively, in order to build the expression for $\omega$, we considered the effects of the number of concurrent threads, namely the parameter $k$, on the abort probability. On the basis of simulation results, some of which are reported in Figure 7, we decided also in this case to use a parametric logarithmic function as the approximation curve of $\omega$ vs $k$. Clearly, the constraint needs to be accounted for that if $k = 1$ then $\omega = 0$ (since the absence of concurrency cannot give rise to transaction aborts). Thus, we approximate $\omega$ as

$$h \cdot (ln(l \cdot (k - 1) + 1), \tag{8}$$

where $h$ and $l$ are the fitting parameters. Again, we validated the out-coming function for $p_a$, depending on $\omega$ (and hence depending on modeled effects of the variation of $k$), using the same amount of workload profiles as in the previous studies, still selected by randomly varying $ww_a$, $ws_s$, $rw_a$, $rs_s$ and $k$. Some results are depicted in Figure 8 for variations of $k$ between 1 and 64. The average error we measured in all the tests was 2.1%.

Finally, we built the expression of $\phi$, which depends on $t_t$ and $t_{ntc}$. To this aim, we note that if $t_t = 0$ (which represent the unrealistic case where transactions are executed instantaneously) then $\phi$ must be equal to 0 (given that the likelihood of concurrent transactions is zero). Additionally, we note that $t_t$ can be seen as the duration of a *vulnerability window* during which the transaction is subject to be aborted. For longer fractions of time during which transactions are vulnerable, higher probability of actual transaction aborts

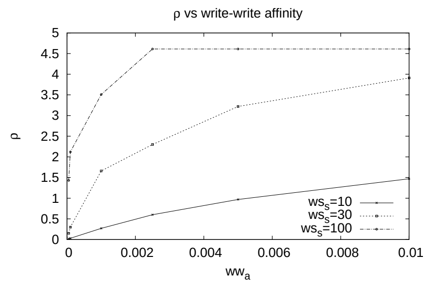

Figure 1: Variation of $\rho$ with respect to the write-write affinity

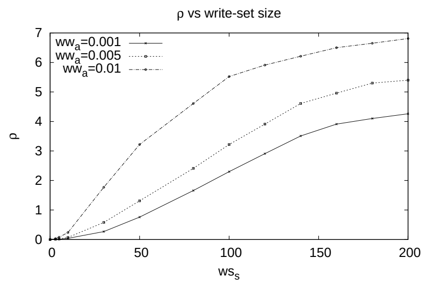

Figure 2: Variation of $\rho$ with respect to the write-set size

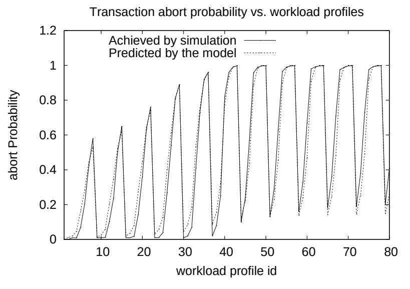

Figure 3: Simulated vs predicted abort probability while varying $ww_a$ and $ws_s$

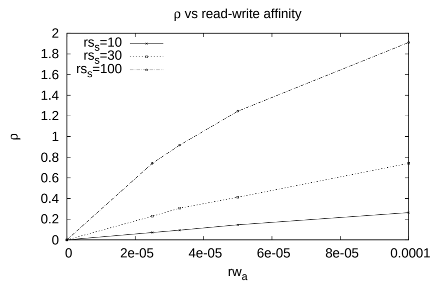

Figure 4: Variation of $\rho$ with respect to the read-write affinity

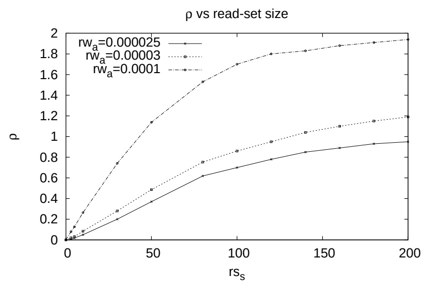

Figure 5: Variation of $\rho$ with respect to the read-set size

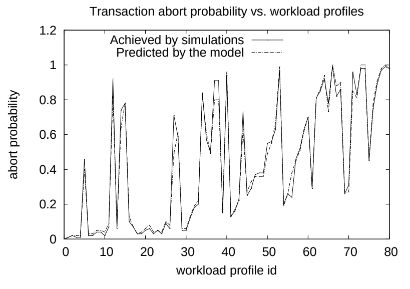

Figure 6: Simulated vs predicted abort probability vs $rw_a$, $rs_s$, $ww_a$, $ws_s$

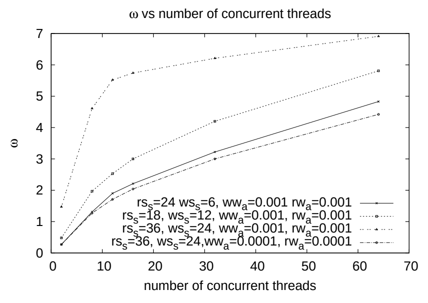

Figure 7: Variation of $\omega$ vs the number of concurrent threads

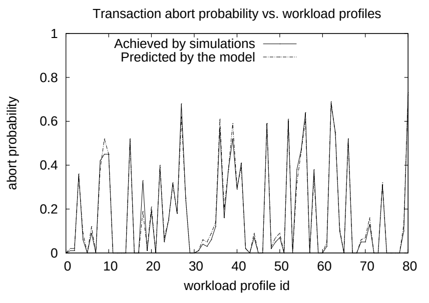

Figure 8: Simulated vs predicted abort probability vs $k$, $rw_a$, $rs_s$, $ww_a$, $ws_s$

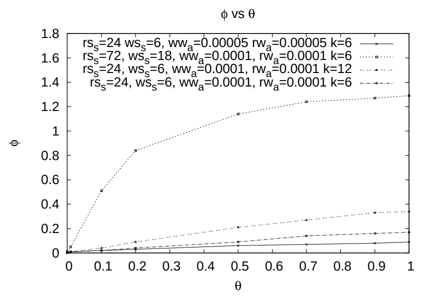

Figure 9: Variation of $\phi$ with respect to $\theta$

can be expected. Thus we assume $\phi$ to be proportional to

$$\theta = \frac{t_t}{t_t + t_{ntc}} \qquad (9)$$

We analyzed through simulation the relation between $\phi$ and $\theta$. Some results are shown in Figure 9, on the basis of which we decided to approximate $\phi$ using the function:

$$m \cdot ln(n \cdot \theta + 1) \qquad (10)$$

where $m$ and $n$ are the fitting parameters.

The expression of $p_a$ in equation (3) is now fully defined. To validate it, we used the same approach that has been adopted for the validation of each of the aforementioned incremental steps. Some results, where we randomly selected workload profiles, are shown in Figure 10. In all our tests, we measured an average relative error of 4.8%.

### D. Model Validation with Respect to a Real System

As a further validation step we compared the output by the proposed model with real measurements taken by running applications belonging to the STAMP benchmark suite [2] on top of the open source TinySTM framework [11]. Additionally, we evaluated the model ability to provide accurate predictions while varying the amount of samples used to perform the regression analysis, gathered through observations of the behavior of the real system. Particularly, we evaluated the extrapolation capability of the model, namely its ability to forecast the transaction abort probability

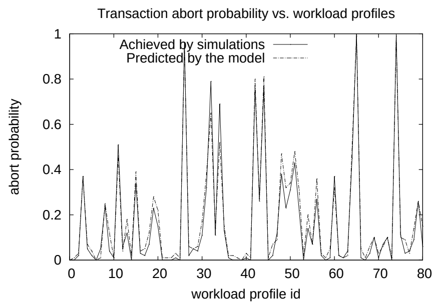

Figure 10: Simulated vs predicted abort probability while varying all the workload profile parameters

that would be achieved when running the STM application with concurrency levels (number of threads) not included in the observed domain where regression samples were taken.

The presented results refer to three different benchmark applications of the STAMP suite, namely Kmeans, Yada and Vacation. As shown in [2], these applications are characterized by quite different workload profiles. This allowed us to evaluate the model accuracy with respect to a relatively wide workload configuration domain. All the tests have been performed on top of an HP ProLiant server equipped with two AMD OpteronTM6128 Series Processor, each one having eight CPU-cores (for a total of 16 cores), and 32 GB RAM, running Linux (kernel version 2.7.32-5-amd64).

For each application, we performed regression analysis to calculate three different sets of values for the model parameters, hence instantiating three models relying on the proposed parametric analysis. Any regression has been performed using one of three different sets of measurements, each set including 80 samples. The first set included samples gathered observing the application running with 2 and 4 concurrent threads. The second one included samples gathered observing the application running with 2, 4 and 8 concurrent threads. Finally, the third one included samples gathered observing the application running with 2, 4, 8 and 16 concurrent threads. This allowed us to evaluate the extrapolation ability of the model, with respect to the number of concurrent threads, while observing the application for limited amounts of concurrency levels (say for 2, 3 or 4 different levels of concurrency). We performed, for each application, the following tests. After setting up the model instances, we executed a set of runs of the application using different values for the application input parameters (leading the same application to run with somehow different workload profiles) and with a number of concurrent threads spanning from 2 to 16. During each run, we measured the average values of the workload profile features included in the set $P$ along different observation intervals having a pre-established length, and we used them as the input to the three instantiated models in order to compute the expected abort probability for each observation interval. After, for

| | **Observed concurrency levels for the regression analysis** | | |
|---|---|---|---|
| **application** | **2/4 threads** | **2/4/8 threads** | **2/4/8/16 threads** |
| *Vacation* | 2.166% (0,00089) | 1.323% (0,00028) | 1.505% (0,00032) |
| *Kmeans* | 18.938% (0,09961) | 2.086% (0,00100) | 2.591% (0,00109) |
| *Yada* | 2.385% (0,00029) | 2.086% (0,00016) | 2.083% (0,00022) |

Table I: Abort probability prediction error (and its variance)

each instantiated model, we compared the predicted value with the real one observed during the runs.

In Table I, we report the average value of the prediction error (and its variance) for all the target benchmark applications, and for the three model instances, while considering variations of the actual level of concurrency between 2 and 16. By the results, we note that, for the cases of Yada and Vacation, it has been sufficient to execute regression analysis with samples gathered observing the application running with only 2 and 4 threads in order to achieve an average prediction error bounded by 2.4% for any level of concurrency between 2 and 16. When enlarging the observation domain for the gathering of samples to be used by regression, i.e. when observing the application running also with 8 concurrent threads, we achieved for Yada a slight error reduction. With Vacation, the reduction is more accentuated. On the other hand, the prediction error achieved for Kmeans with observations of the application running with 2 and 4 concurrent threads was greater. However, such an error drastically drops down when including samples gathered with 8 concurrent threads in the data set for regression. As for regression based on samples gathered with 2, 4, 8 and 16 threads, we note that the error marginally increases in all the cases. We believe that this is due to the high variance of the values of the transaction abort probability we measured for executions with 16 concurrent threads, which gives rise to variability of the results of the regression analysis depending on the set of used observations. Overall, by the results, we achieved good accuracy and effectiveness by the model since it can provide low prediction error, for a relatively wide range of hypothesized thread concurrency levels (namely between 2 and 16) by just relying on observing the application running with 2, 4 and (at worst also) 8 concurrent threads.

We conclude this section comparing the extrapolation ability of our model with respect the neural network-based model proposed in [14], which, similarly to ours, has been targeted at the estimation of the STM performance (vs the level of concurrency). To perform fair comparison, a same set of observations has been provided in input to both the models. Particularly, the reported results refer to the Yada benchmark application, for which we provided a set of 80 observations (the same used for validating the model, as shown above), related to executions with 2 and 4 concurrent threads, in input to both our parametric model and the neural network based model in [14]. As for the neural network approach, we used a back-propagation algorithm

[15], and we selected the best trained network, in terms of prediction accuracy, among a set of networks having a number of hidden nodes spanning from 2 to 16, using a number of algorithm iterations spanning from 50 to 1600. In Figure 11, we show two dispersion charts, each one representing the correlation between the measured values of the transaction abort probability and the ones predicted using the model (left chart) and the neural network (right chart). These refer to concurrency levels spanning in the whole interval 2-16. We remark that a lower prediction error corresponds to a higher concentration of points along the diagonal straight line evidenced in the graphs. We can see that, in the case of the neural network, there is a significantly wider dispersion of points compared to the model we are proposing. In fact, the average prediction error for the neural network is equal to 17.3%, while for the model it is equal to 2.385%. This is a clear indication of higher ability to extrapolate the abort probability by the model when targeting concurrency levels for which no real execution sample is available (and/or that are far from the concurrency levels for which sampling has been actuated). As a reflection, the parametric model we present provides highly reliable estimations, even with a few profiling data available for the instantiation of its parameters. Hence it is suited for the construction of concurrency regulation systems inducing low overhead and providing timely selection of the best suited parallelism configuration (just because the model needs a few samples related to a limited set of configurations in order to deliver its reliable prediction on the optimal concurrency level to be adopted). A concurrency self-regulation architecture exploiting the parametric model is presented and experimentally assessed in the next section.

## IV. Concurrency Self-Regulating STM

### A. The Architecture

The architecture of the Concurrency Self-Regulating STM (CSR-STM) is depicted in Figure 12. A Statistic Collector (SC) provides a Control Algorithm (CA) with the average values of workload profile parameters, i.e. $rs_s$, $ws_s$, $rw_a$, $ww_a$, $t_t$ and $t_{ntc}$, measured by observing the application on a periodic basis. Then, the CA exploits these values to calculate, through the parametric model, the transaction abort probability $p_{a,k}$ as predicted when using $k$ concurrent threads, for each $k$ such that $1 \leq k \leq max_{thread}$. The value $max_{thread}$ represents the maximum amount of concurrent threads admitted for executing the application. We remark that a number of concurrent threads larger then the number of available CPU-cores typically penalizes STM performance (e.g. due to costs related to context-switches among the threads). Hence, it is generally convenient to bound $max_{thread}$ to the maximum number of available CPU-cores. The set $\{(p_{a,k}), 1 \leq k \leq max_{thread}\}$ of predictions is used by CA to estimate the number $m$ of concurrent threads which is expected to maximize the application throughput. Particularly, $m$ is identified as the value of $k$ for which

$$\frac{k}{w_{t,k} + t_{t,k} + t_{ntc,k}} \tag{11}$$

is maximized. In the above expression: $w_{t,k}$ is the average transaction wasted time (i.e. the average execution time spent for all the aborted runs of a transaction); $t_{t,k}$ is the average execution time of committed transaction runs; $t_{ntc,k}$ is the average execution time of $ntc$ blocks. All these parameters refer to the scenario where the application is supposed to run with $k$ concurrent threads.

We note that $w_{t,k} + t_{t,k} + t_{ntc,k}$ is the average execution time between commit operations of two consecutive transactions executed by the same thread when there are $k$ active threads. Hence, expression (11) represents the system throughput. Now we discuss how $w_{t,k}$, $t_{t,k}$ and $t_{ntc,k}$ are estimated. We note that $w_{t,k}$ can be evaluated by multiplying the average number of aborted runs of a transaction and the average duration of an aborted transaction run when the application is executed with $k$ concurrent threads. Thus, the average number of aborted transaction runs with $k$ concurrent threads can be estimated as $p_{a,k}/(1-p_{a,k})$, where $p_{a,k}$ is calculated through the presented model.

To calculate the average duration of an aborted transaction run, and to estimate $t_{t,k}$ and $t_{ntc,k}$, while varying $k$, an hardware scalability model has to be used. In the presented version of CSR-STM, we exploited the model proposed in [12], where the function modeling hardware scalability is

$$C(k) = 1 + p \cdot (k-1) + q \cdot k \cdot (k-1) \tag{12}$$

where $p$ and $q$ are fitting parameters, and $C(k)$ is the scaling factor when the application runs with $k$ concurrent threads. The values of $p$ and $q$ are again calculated through regression analysis. Thus, assuming that, e.g., during the last observation interval there were $x$ concurrent threads and the measured average transaction execution time was $t_{t,x}$, CA can calculate $t_{t,k}$ for each value of $k$ through the formula $t_{t,k} = C(k)/C(x) \cdot t_{t,x}$.

Once estimated the number $m$ of concurrent threads which is expected to maximize the application throughput, exactly $m$ threads are kept active by CA during the subsequent workload sampling interval.

### B. Evaluation Study

In this section we present an experimental assessment of CSR-STM, where we used Vacation, Kmeans and Yada, which have been run on top of the same 16-core HP ProLiant server exploited for previous experiments. All the tests we present focus on the comparison of the execution time achieved by running the applications on top of CSR-STM and on top of the original version of TinySTM. Specifically, in each test, we measured, for both CSR-STM and TinySTM, the delivered application execution times while varying

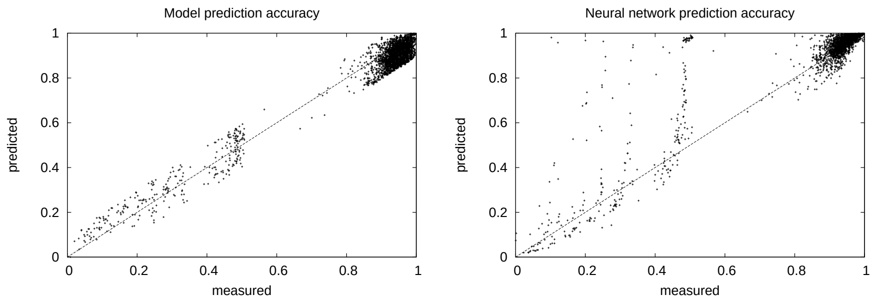

Figure 11: Model and neural-network prediction accuracy

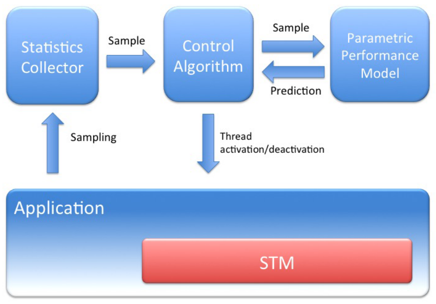

Figure 12: CSR-STM architecture

$max_{thread}$ between 2 and 16. For TinySTM, $max_{thread}$ corresponds to the (fixed) number of concurrent threads exploited by the application. While, in the case of CSR-STM, the application starts its execution with a number of concurrent threads equal to $max_{thread}$. However, CSR-STM may lead to changes of the number of concurrent threads setting it to any value between 1 and $max_{thread}$.

For each application, we calculated the values of the model parameters through regression analysis, using samples gathered observing the application running with 2 and 4 concurrent threads for the cases of Vacation and Yada, and including also observations with 8 concurrent threads for the case of Intruder. As for the parameters appearing in the hardware scalability model expressed in (12), regression analysis has been performed by using, for each application, the measured average values of the committed runs of transactions, observed with 2, 4 and 8 concurrent threads.

We performed a number of runs using, for each application, different values for the input parameters. Due to space constraints, we only report results achieved with two different workload profiles for each application, which are shown in Figures 13, 14 and 15 for Vacation, Kmeans and Yada, respectively. We explicitly report, according to the input-string syntax established by STAMP, the values of the input parameters used to run the applications.

Observing the results, the advantages of CSR-STM with respect to TinySTM can be easily appreciated. For system configurations where CSR-STM is allowed to use a maximum number of threads ($max_{thread}$) greater then the optimal concurrency level (as identified by the peak performance delivered by TinySTM), it always tunes the concurrency level to suited values. Thus it avoids the performance loss experienced by TinySTM when making available a number of CPU-cores exceeding the optimal parallelism level. Particularly, the performance by TinySTM tends to constantly degrade while incrementing the parallelism level. Conversely, CSR-STM prevents this performance loss, providing a performance level which is, for the majority of the cases, near to the best value, independently of the actual number of available CPU-cores for running the application. Obviously, when $max_{thread}$ is lower then the optimum concurrency level, CSR-STM can not activate the well suited number of concurrent threads, which equals the optimal level of parallelism. Thus, for these configurations, the performance of CSR-STM is, in some cases, slightly reduced with respect to TinySTM due to the overhead associated with the components/tasks proper of the concurrency self-regulation mechanism. As for the latter aspect, all the components except SC (for which we measured a negligible overhead), require a single (non-CPU-bound) thread. Thus, resource demand is reduced, wrt the total application demand, of a factor bounded by $1/k$ (when $k$ CPU-cores are available). Accordingly, the cases where CSR-STM provides lower performance than TinySTM (e.g. when $max_{thread}$ is less than 4 for Vacation and Kmeans), the advantage by TinySTM progressively decreases vs $max_{thread}$.

## V. Summary

In this article we have presented a parametric analytical model for determining the optimal level of concurrency in STM applications. Instantiation of the parameters can be actuated via a light regression process based on a few samples related to the run-time behavior of the application. Also, the model does not rely on any strong assumption in relation to the application profile, hence being usable in generic application contexts. It has been validated via comparison with real data traced by running applications from the STAMP benchmark suite on top of a 16-core HP ProLiant machine. We also presented a concurrency self-regulation architecture based on the model, which has been integrated in the TinySTM open source framework, and reported experimental data showing its ability to regulate the concurrency level to well suited values.

## References

[1] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proc. 14th Int. Euro-Par Conference*, pages 719–728. 2008.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. 4th IEEE Int. Symposium on Workload Characterization*, pages 35-46. 2008.

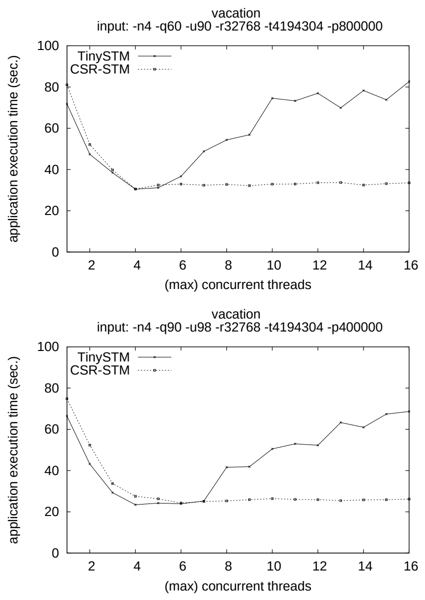

Figure 13: Execution time for Vacation with CSR-STM and TinySTM

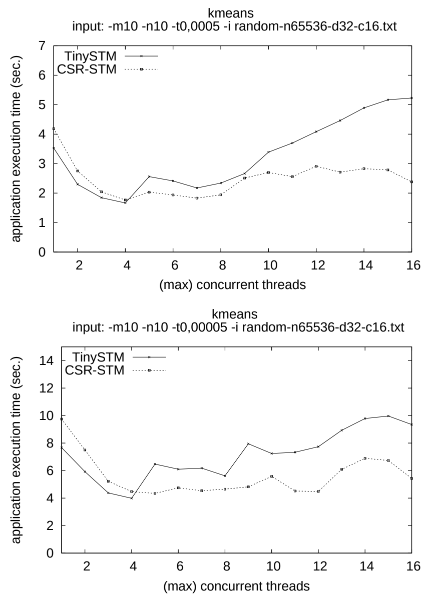

Figure 14: Execution time for Kmeans with CSR-STM and TinySTM

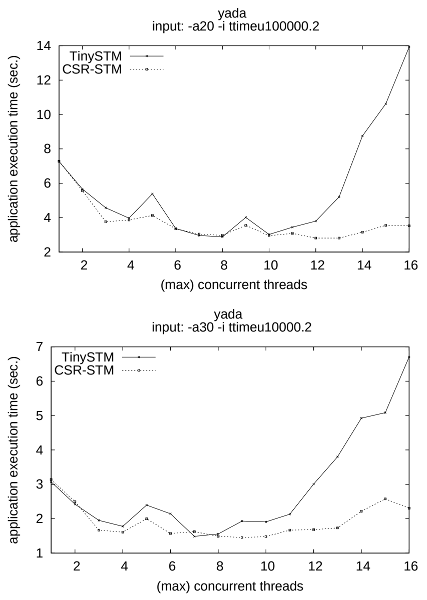

Figure 15: Execution time for Yada with CSR-STM and TinySTM

[3] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proc. 18th Int. Conf. on High Perf. Comp.*, pages 1-10. 2011.

[4] P. di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Perf. Eval.*, 69(5):187–205, 2012.

[5] P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano. A performance model of multi-version concurrency control. In *Proc. 16th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm. Systems*, pages 41–50. 2008.

[6] P. di Sanzo, R. Palmieri, B. Ciciani, F. Quaglia, and P. Romano. Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In *Proc. 26th Int. Conf. on Performance Eng.*, pages 69–78. 2010.

[7] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the optimal level of parallelism in transactional memory systems. In *Proc. Int. Conf. on Networked Systems*. 2013.

[8] A. Dragojević and R. Guerraoui. Predicting the scalability of an stm a pragmatic approach. In *Proc. 5th ACM Workshop on Transactional Computing*. 2010.

[9] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proc. 28th ACM Symposium on Principles of Distributed Computing*, pages 7–16. 2009.

[10] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel. Predicting replicated database scalability from standalone database profiling. In *Proc. 4th ACM European Conference on Computer Systems*, pages 303–316. 2009.

[11] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. 13th ACM Symposium on Principles and Practice of Parallel Programming*, pages 237–246. 2008.

[12] N. J. Gunther. *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services*. Springer, 2007.

[13] Z. He and B. Hong. Modeling the run-time behavior of transactional memory. In *Proc. 18th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm. Systems*, pages 307–315. 2010.

[14] D. Rughetti, P. di Sanzo, B. Ciciani, and F. Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proc. 20th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm. Systems*, pages 278–285. 2012.

[15] S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[16] I. K. Ryu and A. Thomasian. Performance analysis of centralized databases with optimistic concurrency control. *Performance Evaluation*, 7(3):195–211, 1987.

[17] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. 1995.

[18] Y. C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transaction on Database Systems*, pages 415–462, 1985.

[19] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comp. Surveys*, pages 70–119, 1998.

[20] Q. Wang, S. Kulkarni, J. V. Cavazos, and M. Spear. Towards applying machine learning to adaptive transactional memory. In *Proc. 6th Workshop on Transactional Computing*. 2011.

[21] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proc. 20th Symp. on Parallelism in Algorithms and Archit.*, pages 169-178. 2008.

[22] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, pages 831–872, 1993.