Markov Chain-based Adaptive Scheduling in Software Transactional Memory

Pierangelo Di Sanzo, Marco Sannicandro, Bruno Ciciani, Francesco Quaglia
DIAG–Sapienza Università di Roma

***Abstract*—Software Transactional Memory (STM) may suffer from performance degradation due to excessive conflicts among concurrent transactions. An approach to cope with this issue consists in putting in place smart scheduling policies which temporarily suspend the execution of some transaction in order to reduce the actual conflict rate. In this paper, we present an adaptive transaction scheduling policy relying on a Markov Chain-based model of STM systems. The policy is adaptive in a twofold sense: (i) it schedules transactions depending on throughput predictions by the model as a function of the current system state; (ii) its underlying Markov Chain-based model is periodically re-instantiated at run-time to adapt it to dynamic variations of the workload. We also present an implementation of our adaptive transaction scheduler which has been integrated within the open source TinySTM package. The accuracy of our performance model in predicting the system throughput and the advantages of the adaptive scheduling policy over state-of-the-art approaches have been assessed via an experimental study based on the STAMP benchmark suite.**

***Keywords*-transactional memory; scheduling; performance modeling; performance optimization;**

I. Introduction

The large diffusion of multi-core architectures has raised the need for programming paradigms aimed at simplifying the development of concurrent and parallel applications. Particularly, efficient synchronization of thread accesses to shared-data has become a core aspect to cope with.

Coarse-grain locking is a quite immediate synchronization mode, but it generally provides reduced performance levels, especially for scaled up numbers of threads. Conversely, fine-grain locking strategies provide the potential for improved performance levels, but are time-consuming and error-prone for programmers.

Transactional Memory (TM) stands as an alternative, simple and intuitive, transaction-based synchronization approach. With TM, programmers can think the easy way in terms of coarse-grain locking (transactions), while application performance can benefit from fine-grain data-conflict management mechanisms transparently operated at the level of the TM layer. The relevance of this kind of approach is clearly evident by the recent inclusion of TM support in world-leading multiprocessor hardware and open source compilers (see [1]).

However, although providing a number of advantages especially on the side of programmability, the performance of TM systems can be strongly affected by trashing phenomena. Indeed, depending on the actual data access pattern by transactions, an high degree of transaction concurrency may lead to experience (execution phases of) workloads generating an unacceptably high rate of aborts. This is detrimental to performance, given that an aborted transaction leads to useless carried out work. This phenomenon affects both Software based implementations of the TM layer (STM) [2] and Hardware based ones (HTM) [3], thus being a highly general and very relevant problem to cope with.

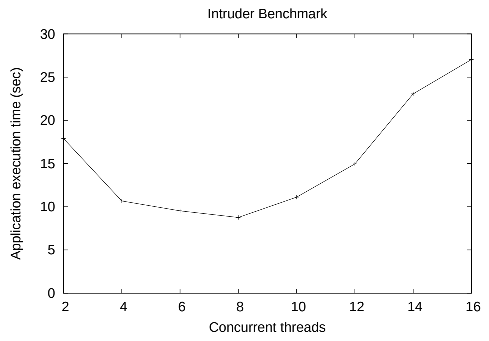

Figure 1. Execution time of the Intruder benchmark on top of TinySTM.

To provide the reader with empirical data quantifying this problem, we report in Figure 1 the variation of the execution time of the Intruder benchmark—taken from the STAMP benchmark suite [4]—while changing the number of concurrent threads. This experiment has been executed on top of a 16-cores HP ProLiant server equipped with 2GHz AMD Opteron 6128 processors and 64 GB of RAM. The underlying operating system is Linux (kernel version 2.7.32-5-amd64) and the TM layer is TinySTM [5]. By the plot we see that with more than 8 concurrent threads the performance rapidly decreases because of thrashing phenomena. With 16 concurrent threads the execution time increases up to about 3 times the one achieved with 8 concurrent threads. Overall, mechanisms avoiding thrashing and keeping performance close to optimal values for the target (changing) workload are mandatory.

Such an issue has been dealt with in literature by relying on either a *transaction scheduler* [6] or a *thread scheduler* [7]. The transaction scheduling approach is based on delaying the execution of a transaction depending on the current system state and some scheduler-embedded policy. As an example, a transaction could be temporarily blocked until the number of already running transactions falls below a given threshold value. Alternatively, it could be delayed if its estimated probability to conflict with already running transactions is high. Thread scheduling in TM systems is instead based on dynamically changing the number of threads sustaining the application execution, depending on how this variation favors or not the system throughput. As for a high level comparison of the two approaches, transaction scheduling looks to be a more general one since it can be employed in contexts where different threads are bound to different transactional tasks. In such a scenario, temporarily blocking the execution of some specific thread via a thread-scheduling approach would result in starvation of any thread-bound task. Nonetheless, the two approaches (transaction vs thread scheduling) are kind of orthogonal methods that could be ideally combined. In this article our focus is on transaction scheduling, particularly for STM layers.

State-of-the-art scheduling approaches for STM can be divided in two groups: (a) the ones based on performance prediction models (e.g. [7]), such as analytical or machine learning models describing the system performance as a function of the degree of concurrency among transactions, and (b) the ones based on heuristic methods (e.g. [8]). Model-based approaches have the drawback of requiring a-priori profiling of the system for collecting measurements and parameter values to instantiate the performance model. On the other hand, heuristic-based ones require the user to configure scheduler parameters (such as conflict rate thresholds) based on which the decisions on how to schedule transactions is taken. This may not be an easy task given that the optimal parameter configuration may depend on both workload and system level settings. Additionally, since the workload profile may change over time, the selected parameter settings could be optimal for given execution phases of the application, but could be unsuitable for others.

Example data showing the effects on performance by different configurations of scheduler parameters are provided in Figure 2. The data refer to two heuristic-based state-of-the-art transaction schedulers, namely Shrink [9] and ATS [6]. These schedulers become operational as soon as the contention level among transactions oversteps a given threshold referred to as *Contention Intensity* (CI), which needs to be set by the user. The plotted data show the execution time of the Intruder benchmark when using (in a given system configuration) different CI values, ranging from 10% to 90%. We can note that the delivered performance significantly changes (with the different settings) for both the schedulers. Overall, the efficiency of these schedulers clearly depends on the ability to set up the right parameters' configuration, which might be a non-trivial task especially for unknown/unforeseen workloads.

To tackle the aforementioned problems, we devise a Markov Chain-based performance modelling approach with the aim at developing an innovative adaptive STM scheduler. Our proposal exploits a lightweight model predicting the system throughput, which can be instantiated on-the-fly at

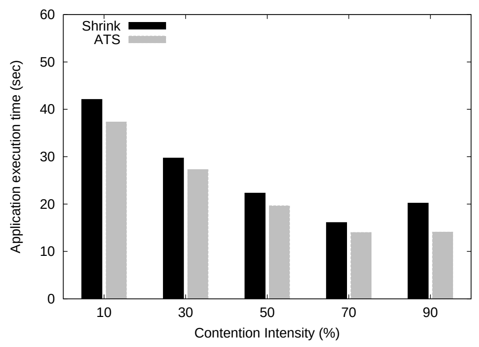

Figure 2. Performance variation with different values of CI.

run-time (e.g. according to a periodic basis) by collecting a reduced number of samples related to the current transactional workload profile. The predictions by the model are then used for scheduling purposes by adaptively tuning the number of transactions that are allowed to run concurrently along a given application execution phase.

Our approach provides two main advantages: (a) it does not require a-priori (e.g. off-line) workload profiling, just because the performance model can be instantiated on-the-fly, and (b) it does not rely on any complex configuration parameter, given that it only requires to set the length of the time period for sampling the workload and re-instantiating the model. Additionally, the model instantiation requires the measurement of a very small set of parameters at run-time (namely, 4 parameters in total) and does not need to sample read/write-sets of transactions.

We integrated our Markov Chain-based adaptive scheduler within the open source TinySTM package and we also report data showing its advantages, when compared to a few state-of-the-art proposals, for the case of benchmark applications belonging to the STAMP suite.

The remainder of this article is organized as follows. Related work is discussed in Section II. The Markov Chain-based performance model is presented in Section III, together with a model-validation study. The model-based adaptive transaction scheduling policy is described in Section IV. The implementation of the adaptive scheduler within TinySTM and its experimental assessment are presented in Section V.

II. Related Work

Our proposal is related to results in the field of performance modeling of TM systems, as well as to scheduling approaches based on either models or heuristics.

As for analytical models targeting STM performance prediction, the works in [10], [11], [12] share with our proposal the reliance on Markov Chain-based modeling formalisms. However, the proposals are suited for off-line STM performance analysis and rely on training data collected during system profiling phases in order to instantiate the models.

They show therefore limited (if not null) applicability to the problem of on-line performance forecasting and adaptive scheduling. This limitation is also linked to the high number of parameters to be estimated for instantiating the models. Another drawback of these modeling approaches is that the size of the Markov Chain grows quadratically with the number of concurrent threads and/or the number of operations executed by transactions. This may generate non-negligible overhead in case one would decide to periodically instantiate and resolve these models at run-time for adaptive scheduling purposes. Unlike the above approaches, we present a lightweight Markov Chain-based model, which: (a) requires the estimation of only 4 parameters for being instantiated, which can be done at run-time (and periodically) in a non-intrusive manner, given that they correspond to transaction start/end times and transaction commit/anort outcomes; (b) grows linearly (not quadratically) with respect to the number of concurrent threads.

Another model-based study aimed at forecasting STM performance has been presented in [13]. In this proposal, the performance model is built off-line collecting (for a given application) speedup measures while varying the level of concurrency among transactions. Performance prediction functions are generated by interpolating the collected data via a set of parametric functions. This approach is again non-suitable for on-line scheduling, given that it still requires an a-priori profiling phase of the system behavior (under a given workload).

Although our focus is on transaction scheduling, our proposal is anyhow loosely related to a few results in the area of thread scheduling in TM systems, which we shortly discuss. A thread scheduler for STM based on an analytical performance model has been described in [14]. In this solution the model is instantiated via regression analysis applied to a family of reference functions on the basis of measurements collected during system profiling phases. The work in [7] presents a machine learning-based thread scheduling approach for STM. This solution has been then improved, as described in [15], by introducing a dynamic feature selection mechanism to reduce the run-time workload sampling overhead for determining the input to the pre-instantiated machine learning-based performance model. The work in [2] proposes a solution for instantiating an STM performance model via the combination of analytical and machine learning techniques so as to reduce the long training-phase of the pure machine learning-based approach. Finally, in [16] a machine learning-based model, instantiated off-line, is used to derive a thread scheduling mechanism suited for HTM systems. As compared to our approach, all these proposals still rely on lengthy preliminary profiling phases in order to instantiate the performance models that drive thread scheduling, a problem that is only partially addressed by either the pure analytical approach in [14] or the mixed (analytical/machine-learning) one in [2]. Rather, we provide a very lightweight model with true capabilities for on-line fast (re-)instantiation so as to cope with dynamic and unforeseen workloads. Also, our solution targets the orthogonal objective of transaction scheduling (not the one of thread scheduling in TM systems).

Still for thread scheduling, we can find the heuristic based approach in [17], where the authors propose a hill-climbing scheme that dynamically increases or decreases the number of concurrent threads running the application. As compared to this work, our proposal deals with the orthogonal issue of transaction scheduling and is model-based (rather than being based on an heuristic approach).

As for the transaction scheduling problem in STM systems, joint to the reliance on heuristics as the solution method, we can find the proposals in [8], [6], [9]. The proposed schedulers delay the execution of some transaction when its probability of conflicting with already running transactions is estimated to be high, and the proposals differ from each other by the way they estimate such a conflict probability. The main drawbacks of these transaction schedulers are the following ones: (a) their underlying heuristic techniques do not guarantee convergence to the optimal solution, and their effectiveness may change with respect to the workload profile, and (b) the user (as we discussed before) is in charge of setting configuration parameters that affect the scheduler efficiency. Our proposal aims at overcoming both these drawbacks.

An alternative transaction scheduler for STM systems has been presented in [18]. This is based on the concept of serialization queues where all the transactions accessing a same data partition are sequentialized to avoid aborts due to conflicting concurrent accesses. Proactive move of the transactions to the correct queues requires user specified information in relation to the access pattern, hence the approach is not fully transparent. Unlike this proposal, we do not rely on any transaction queuing discipline, and do not base our scheduler on the concept of partitions of data (to be specified by the user). Hence transactions can be run concurrently in our scheme independently of their actual data access, whose effects on conflict likelihood (in case of actual concurrency) are captured by the Markov Chain-based model. Also, the decisions of our scheduler about whether transactions need to be delayed are fully application transparent.

Still concerning transaction scheduling, we can find the recent work in [3], which is targeted at HTM systems. This proposal copes with the issue of estimating what are the actual sources of data conflicts across transactions executed via HTM support, given that no software layer is used for HTM-based transactions to drive their execution and to profile data access. The estimation is then used to temporarily block the execution of HTM-based transactions that are supposed to access the same data slices touched by already running ones according to a threshold based heuristic scheme. Conversely, our proposal is targeted at STM systems (not HTM ones) and does not rely on heuristics, rather on an analytical performance model.

Finally, less loosely related to our proposal are the works in [19], [20], where operating system support is provided for running TM applications. These proposals target the modification of the Linux scheduler so as to reduce the likelihood of a thread running a transaction to be interfered on the same CPU-core by some other thread, which might lead to the increase of the transaction abort probability. Our proposal is again fully orthogonal to these approaches given that we base it on a user-space scheduling policy/mechanism.

III. The Markov Chain-based STM Performance Model

We present our modelling approach by initially providing a description of the target STM environment, thus introducing the reference system model for our analysis. Then we focus on the derivation of the performance model.

A. Target STM System

We assume an STM system where a number N of concurrent threads are run. A thread can execute either transactional code or non-transactional code (ntc) blocks. A transaction commits if no conflicts with other concurrent transactions occur, otherwise, it is aborted and a new run of the same transaction is executed. We assume a transaction scheduler that admits up to m transactions to be run concurrently, so as to avoid thrashing due to excessive conflicts. Hence, upon the start of a new transaction along any thread, it may either enter the *running* state or the *waiting* state depending on the value of m. The transaction is put into the *waiting* state in case there are m transactions (with $m \leq N$) that already entered the *running* state. When one of these m transactions commits, a waiting transaction, if any, is unblocked, thus being allowed to proceed along its original execution path.

B. Performance Model Derivation

Our performance model leverages a Continuous Time Markov Chain (CTMC) [21] with $N+1$ states. A graphical representation of the CTMC is shown in Figure 3. A state marked with k in the CTMC represents a system state when there are k threads executing transactions, where k accounts for both already running and blocked transactions. Consequently, when the system resides in state k there are $N-k$ threads executing ntc blocks. A transition from state k to $k+1$ occurs upon the startup of a transaction along any thread. A transition from state k to $k-1$ occurs upon the successful commit of whichever running transaction.

We denote with t_{ntc} the average time for executing some ntc block. Thus, the transaction inter-arrival rate along any thread is $\lambda = \frac{1}{t_{ntc}}$. Consequently, denoting with λ_k the transition rate from state k to $k+1$, we have

$$\lambda_k = (N-k) \cdot \lambda \tag{1}$$

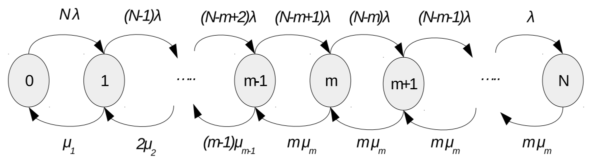

Figure 3. The CTMC used to model the system performance.

As for the transition rate from state k to $k-1$, it depends on k and m (we remark that m represents the number of concurrent transactions that have been allowed to run by the scheduler, hence not being blocked). Denoting with t_k the average transaction execution time when there are k executing transactions, the transaction execution rate in state k is equal to $\mu_k = \frac{1}{t_k}$. Accordingly, for all the states marked with $k \leq m$, since exactly k transactions are running (i.e. none of these transactions is blocked), the transition rate from state k to $k-1$ is

$$\gamma_k = k \cdot \mu_k \tag{2}$$

Conversely, for any state marked with $k > m$, the running transactions are m, while the remaining $k-m$ transactions are blocked. Consequently, for any of the states such that $k > m$, the transition rate to $k-1$ is

$$\gamma_k = m \cdot \mu_k \tag{3}$$

The assumption allowing us to rely on a Markov Chain for modeling the system behavior is that the parameters used in the above equations, which express latency values related to state transitions (e.g. the latency for executing some ntc block) are exponentially distributed.

Transaction execution time. Now we focus on the average transaction execution time t_k. We note that t_k is affected by the number of times a transaction is aborted (hence restarted) while the system is in state k, before successfully committing. We refer to as *wasted time*, which we denote with $w_{t,k}$, the average time spent for executing all the aborted runs of a transaction (including the time to execute abort operations and transaction restarts) while the system is in state k. Further, we refer to as *useful time*, which we denote with $u_{t,k}$, the average time to execute the last transaction run (i.e. the successfully committing one), when the system is in state k. Hence, we have that $t_k = w_{t,k} + u_{t,k}$ by definition.

Further, we also have that the wasted time $w_{t,k}$ is equal to the product between the average time $w^r_{t,k}$ to execute a transaction run that is aborted while the system is in state k and the average number of times r_k a transaction is aborted while the system is in that same state, say

$$w_{t,k} = w^r_{t,k} \cdot r_k \tag{4}$$

Assuming that, for a given state k, the transaction abort event is independent of previous abort events affecting the same transaction, the probability distribution of the number of runs of a transaction, before a successful commit takes place, is geometric. Thus, if p_k is the transaction abort probability when the system is in state k, we have

$$r_k = \frac{p_k}{1-p_k} \tag{5}$$

As a final observation, we can safely assume that all the abort probability values p_k for $k > m$ are equal to p_m, given that in all the states marked with $k > m$, exactly m transactions are in the *running* state.

System throughput. The system throughput thr_m when the scheduler admits at most m transactions to the *running* state can be estimated through the CTMC stationary distribution. Specifically, denoting with q_k the stationary probability of state k, we have

$$\begin{aligned} thr_m &= \sum_{i=1}^{N} q_k \cdot \gamma = q_1\mu_1 + q_2 2\mu_2 + ... \\ &\quad + q_m m\mu_m + q_{m+1} m\mu_m + ... + q_N m\mu_m \end{aligned} \tag{6}$$

In order to calculate q_k, with $0 \leq k \leq N$, we can use the solution equations for general equilibrium [21], say

$$q_k = q_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\gamma_{i+1}}, \tag{7}$$

$$q_0 = \frac{1}{1 + \sum_{k=1}^{N} \prod_{i=0}^{k-1} \frac{\lambda_i}{\gamma_{i+1}}} \tag{8}$$

In order to apply Equation 8 to our CTMC, we define, for any $k \leq m$

$$a_k = \prod_{i=0}^{k-1} \frac{(N-i)\lambda}{(i+1)\mu_{i+1}} \tag{9}$$

and, for any $k > m$

$$b_k = \prod_{i=m}^{k-1} \frac{(N-i)\lambda}{m\mu_m} \tag{10}$$

Hence, by Equation 7, for any state marked with $k \leq m$ we have

$$q_k = q_0 \cdot a_k, \tag{11}$$

and for any state marked with $k > m$ we have

$$q_k = q_0 \cdot a_m \cdot b_k \tag{12}$$

Finally, Equation 8 can be rewritten by spitting the sum at the denominator into two sums, where k varies from 1 to $m-1$ and from m to N, respectively. Thus we achieve

$$q_0 = \frac{1}{1 + \sum_{k=1}^{m-1} a_k + \sum_{k=m}^{N} a_m \cdot b_k} \tag{13}$$

$u_{t,k}$	average transaction useful time in state k
$w^r_{t,k}$	average time for an aborted transaction run in state k
p_k	transaction abort probability in state k
t_{ntc}	average latency of the ntc block (independent of the CTMC state)

Table I
PARAMETERS REQUESTED FOR MODEL INSTANTIATION

By relying on Equations 9-13 we can finally calculate q_k, for any k between 0 and N. Hence we can calculate the throughput values thr_m via Equation 6 for any value of the parameter m corresponding to the maximum number of concurrent transactions admitted to the *running* state.

We finally note that instantiating our model only requires the knowledge of the four parameters listed in Table I, which can be easily and non-intrusively sampled at run time (or even approximated as we will discuss in the remainder of this article). Further, by construction of our CTMC-based model, once fixed a value for m, we have that for any state marked with $k > m$ these parameters will have the same values (given that p_k does not change for $k \geq m$). This further contributes to keep small the number of observations to be collected for instantiating the parameter values characterizing the different states of the CTMC-based model.

C. Validation of the Performance Model

In this section, we present experimental data for an assessment of the accuracy of the above introduced performance model. Further, we illustrate how the model can be used to perform what-if analysis vs the number of concurrent transactions admitted to the *running* state by the scheduler. What-if analysis constitutes one of the building blocks on top of which the adaptive scheduler we present in the next section is built.

We report data that have been achieved with the Intruder, Yada and Vacation applications of the STAMP benchmark suite [4]. We run these applications by deploying them on the open source TinySTM layer [5], hosted by the same 16-core HP ProLiant machine we used for the experiments whose outcomes have been reported in Section I. We augmented TinySTM with profiling capabilities to estimate the parameters in Table I and with the possibility to admit a given maximum number of concurrent transactions to the *running* state. Details on the implementation of these facilities within TinySTM will be provided in Section V, where the description and the experimental assessment of our adaptive transaction scheduler are presented.

Throughput prediction accuracy. In order to evaluate the model accuracy, we compared the predicted system throughput and the real one as measured while the selected benchmark applications were in progress. The throughput prediction has been performed at run-time, every 1000 executed transactions, by dynamically re-instantiating the model. This has been done exploiting samples for the estimation of the parameters in Table I collected along the execution interval of those 1000 transactions. In these experiments we selected a set of different configurations in relation to the number of threads used to run the applications, and the maximum number m of concurrent transactions admitted to the *running* state. Thus we assessed the accuracy of the CTMC-based model in predicting the real system behavior for relatively broad settings.

The results are shown in Figure 4. By the plots we can observe an extremely accurate throughput prediction by the CTMC-based performance model in all the tested configurations, including the ones where the applications are run by relying on 16 threads. This is a relevant achievement when considering that, for two of the three benchmarks, the reliance on 16 threads, none of them ever blocked while running a transaction, represents an over-parallelism configuration leading to thrashing. In fact, with such a settings, the execution times of both Intruder and Yada are definitely stretched (compared to settings with lower levels of parallelism) just due to trashing phenomena. Anyhow, these phenomena, as well as more favorable run-time behaviors, are reliably captured by our performance model. Also, the predictions by the model are extremely accurate independently of the stability of the real throughput curve associated with the different settings. Overall, the average relative error in predicting the system throughout by our model (across all the configurations) falls in the intervals between 5.5% and 8.5% for Intruder, 2.4% and 5.9% for Yada, and 1.3% and 3.3% for Vacation.

What-if analysis accuracy. The CTMC-based model can also be used to perform what-if analysis, which is a core building block for the construction of our adaptive transaction scheduler. We assess this capability by showing how the application throughput can be predicted for some value $m = x'$ of the number of concurrent transactions admitted to the *running* state by instantiating the model on the basis of statistics collected while running under the settings $m = x$.

Thanks to the way our model is built, the throughput for $m = x'$ can be predicted by measuring the values of the parameters in Table I when running with $m = x$ and then solving the model for $m = x'$. However, we note that for scenarios where $x' > x$, the set of transaction abort probabilities $\{p_k : x < k \leq x'\}$, which are needed to solve the model, cannot be determined on the basis of the available observations, hence they need to be estimated in a different way. To this end we rely on the following considerations and approach. When there are x running transactions in the system, a transaction can conflict with any of the other $x - 1$ transactions. Denoting with p_a the probability that a transaction conflicts with one of the other running transactions, the probability for a transaction to experience no abort when there are x running transactions (i.e. the probability that no conflicts occur with any of the other $x - 1$ running transactions) is equal to $(1 - p_a)^{x-1}$. Thus, the abort probability when there are x running transactions can be calculated as

$$p_x = 1 - (1 - p_a)^{x-1} \tag{14}$$

Solving by p_a the above equation we have

$$p_a = 1 - (1 - p_x)^{\frac{1}{x-1}} \tag{15}$$

and if we know the transaction abort probability p_x for a generic state x, we can calculate p_y using Equation 15. Hence, we can calculate p_k for any $k \neq x$, thus also for any $k > x$, using k in place of x in Equation 14.

To evaluate the model accuracy in performing what-if analysis according to the above scheme, we report both real and predicted throughput values for configurations with $m = 2$ and $m = 5$, respectively, where the predictions are based on observations gathered with $m = 4$. Figure 5 reports both real and predicted throughput curves, which show that the average prediction errors are 8.8% ($m = 2$) and 9.2% ($m = 5$) for Intruder, 7.2% ($m = 2$) and 5.9% ($m = 5$) for Yada, and 2.6% ($m = 2$) and 1.9% ($m = 5$) for Vacation. Overall, a very good matching is still observed when using the presented CTMC-based model for what-if analysis purposes according to the devised approach.

IV. Model-based Adaptive Transaction Scheduling

In this section, we describe the adaptive transaction scheduler based on the performance model presented in Section III-B, which we call MCATS (Markov Chain-based Adaptive Transaction Scheduler). The objective of MCATS is the one of maximizing the system throughput by dynamically regulating the number of concurrent transactions admitted to the *running* state along the application lifetime.

MCATS works by initially admitting a default number of m concurrent transactions to the *running* state, where m could be set to correspond to the number of threads running the application. Then, it periodically performs the following steps:

Step 1: Observation. The workload is sampled for a given time period so as to estimate the four parameters listed in Table I, whose values are needed for instantiating the CTMC-based model underling MCATS to calculate the expected throughput values thr_m while varying m. Clearly, it might happen that no samples could have been collected for some particular state k. This may be the case when the transition rates λ_k are relatively high with respect to the transition rates γ_k. In fact, this settings would likely lead the system to work in states associated with high values of k, thus not allowing the measurements of the target parameters in correspondence with states associated with low values of k. Conversely, if γ_k values are relatively high with respect to the transition rates λ_k, measurements for states with

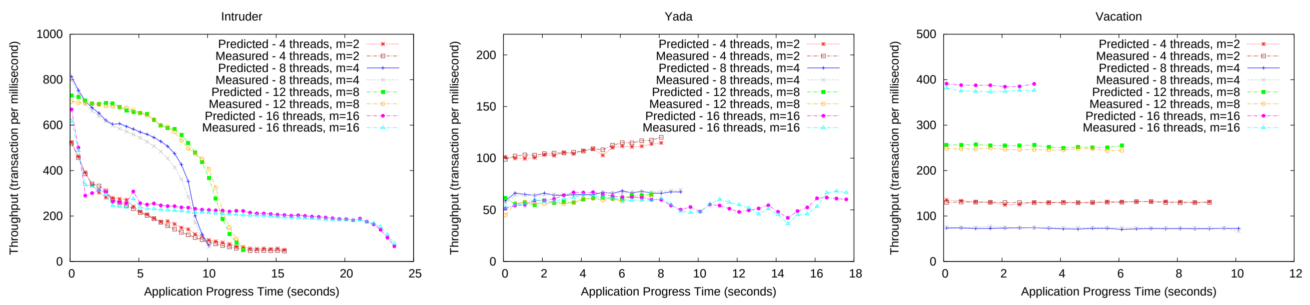

Figure 4. Predicted vs measured throughput for Intruder, Yada and Vacation.

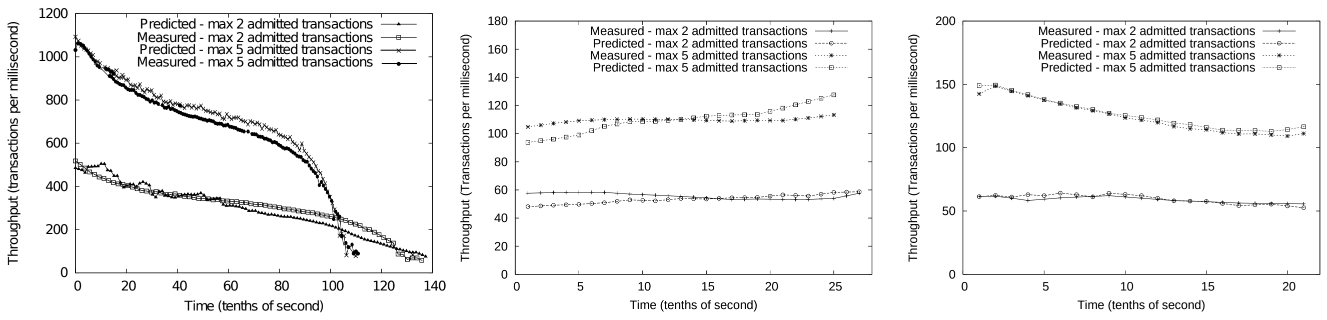

Figure 5. Predicted and measured throughput with m=2 and m=5 based on measurements taken with m=4 for Intruder (left graph), Yada (middle graph) and Vacation (right graph).

higher values of k might be missing. If no measurements are available for some state k, the following fall-back sub-step is executed:

Step 1.1: Fall-back. the missing values of the parameters $u_{t,k}$, $w^r_{t,k}$ and p_k, related to state k are set equal to the average values of these same parameters as observed for all other states for which measurements are available. We note that this settings leads to an approximation that is expected to cause a very low (or null) error on the throughput estimation by the CTMC-based model. In fact, if measurements for some state k are missing, this means that the system is expected to work in state k with very low probability, or even not to enter that state with the current workload. As a consequence, when predicting the system throughput via Equation 6, the associated probability values q_k is expected to be very low. We remark that Equation 6 calculates a weighted sum, where q_k probability values represent the weights.

Step 2: Tuning. The values of the system throughput thr_m, for $0 \leq m \leq N$, are estimated via what-if analysis, and the maximum number of concurrent transactions admitted to the *running* state by the scheduler is set to the value of m for which the predicted thr_m value is the maximum one.

V. Implementation and Assessment of MCATS

We developed an implementation of MCATS ([1]) integrated within the open source TinySTM layer, which is targeted at Posix/x86 platforms. In this section we initially describe a few implementation details and then provide experimental data for its assessment.

A. Implementation Details

We used a shared global variable c to count the number of transactions residing in the *running* state. This counter variable is modified by threads willing to enter a transaction by using the *compare-and-swap* instruction provided by the underlying hardware architecture. Upon the attempt to start a transaction, the thread tries to modify c atomically, and the access to the *running* state is allowed only in case the value of c does not oversteps the value of the scheduling parameter m. In case of check failure, the thread puts the transaction in the *wait* state, which is supported by simply letting the thread keep on retrying the above operation. The value of c is decremented when a transaction successfully commits, so as to allow other threads to eventually succeed in the access to the *running* state for the transactions they are handling.

The duration of **Step 1** in the scheduler logic (namely the system observation phase after which the actual tuning of the maximum number m of concurrent transactions admitted

[1]Available at `https://github.com/HPDCS/stmMCATS`.

to the *running* state is set) corresponds to T subsequent transaction commits, where T is the unique value to be specified by the user, which has anyhow no direct effect on the transaction scheduling logic ([2]).

The latency samples for the estimation of $u_{t,k}$, $w^{r}_{t,k}$ and t_{ntc} are taken by relying on the *RDTSC* instruction, which returns the time-stamp kept by a 64-bit CPU register measuring the passage of time by counting the number of CPU-cycles since the machine was started ([3]). The abort probability p_k is estimated by computing the ratio between the number of aborted transaction runs and the sum of all aborted and committed transaction runs. In order to associate each taken sample with the correct state k, we rely on the value of c at the time the samples are taken (recall that for $k \geq m$ all the statistics are collapsed together given that they coincide for all the states associated with those values of k).

For the purpose of keeping low the sampling overhead, only one thread at a time collects the target statistics. The thread responsible of collecting measurements is selected in a round-robin fashion, with round-robing taking place according to a fine-grain scheme operating within each observation window. This scheme allows more robust sampling outcomes, especially in contexts where the underlying architecture is characterized by asymmetries such as *Non-Uniform Memory Access* (NUMA). In such a case, the latencies sampled by one individual thread could be biased on the basis of the distance between the CPU-core where the thread is running and the NUMA node where actual memory accesses were performed. Averaging the samples taken by multiple threads operating in round-robin fashion would allow for reducing this bias. Further, a better estimation of the average values of the target parameters is guaranteed by the round-robin scheme in case the application is such that specific transactional profiles are bound to given threads (a kind of application level asymmetry).

B. Experimental Results

We assessed MCATS by relying on the same experimental setting that has been described in Section III-C. For all the selected benchmark applications, we present results for three different inputs, which gives rise to different workload profiles of the applications, as described in [4]. We compared the performance results achieved with MCATS with those achieved by the baseline implementation of TinySTM (employing no transaction scheduling logic) and two other schedulers proposed in literature, i.e. Shrink and ATS, both integrated within releases of TinySTM. For these schedulers we used the configurations proposed in [9] and [6], respectively. The results of our experimental study are shown in Figures 6, 7 and 8 for Intruder, Yada and Vacation, respectively. We report the application execution time as a function of the number of concurrent threads used for running the application. By the plots we note that, for almost all the test cases, the best performance (i.e. the minimum application execution time) with the baseline TinySTM is achieved for a number of concurrent threads lower than 16 (except for configuration 2 of Yada, where the best performance is achieved with 16 concurrent threads). When executing the applications with parallelism degree higher than the optimal one, the performance provided by the baseline TinySTM (rapidly) drops down. This is because the transaction conflict rate grows more rapidly than the performance improvement potentially provided by the increased transaction parallelism. In these scenarios, a transaction scheduler should likely reduce the transaction conflict rate in order to prevent the performance loss caused by thrashing. Conversely, in scenarios with fewer threads (hence with lower transaction conflict rates), the scheduler should not hamper performance by excessively blocking transactions. The data we report allow assessing MCATS in these two antithetical scenarios.

The results show that, when running with fewer threads than the optimal parallelism level, the execution time values achieved by any of the considered schedulers is generally (slightly) worse that the one provided by the baseline TinySTM. This is essentially due to the overhead associated with the implementation of the schedulers, and to the fact that, in under-parallelism settings, a transaction scheduler is not expected to provide benefits, given that the system likely works far from thrashing conditions. For Yada, which represents a kind of worst case overhead scenario for MCATS, the response time with MCATS is, on the average, 11.3% higher than the one with baseline TinySTM for executions with fewer threads than the optimal number. For Intruder and Vacation, it is, on the average, 7.3% and 4.5% higher. However, the results also show that, still for these scenarios, the application response time of MCATS with respect to other schedulers is noticeably lower for Intruder and Yada, and comparable for Vacation.

In scenarios where the applications run with parallelism degree higher than the optimal one, which leads the performance of the baseline TinySTM to drop down, MCATS effectively prevents such a performance loss, for all the test cases. Also, it gives rise to a considerably lower application execution time than the other schedulers for all the configurations of Intruder and Yada, as well as for two configurations out of three of Vacation (say configurations 1 and 3). As for configuration 2 of Vacation, we note that

[2]The Value of T may only impact the statistical goodness of the parameters that need to be sampled for instantiating the CTMC-based model. However, as we have shown via model validation data, T could be set to a few thousands of transactions independently of the managed workload (at least for the considered applications) while still guaranteeing reliable sampling outcomes. A thorough study on how to automate the settings of this parameter is planned as future work.

[3]This approach allows measuring wall-clock-time interval durations. Hence any non-negligible bias on the duration of the sampled time intervals due to time-sharing interference, e.g. by kernel-level housekeeping threads, is tackled via the elimination of spike values from the statistics.

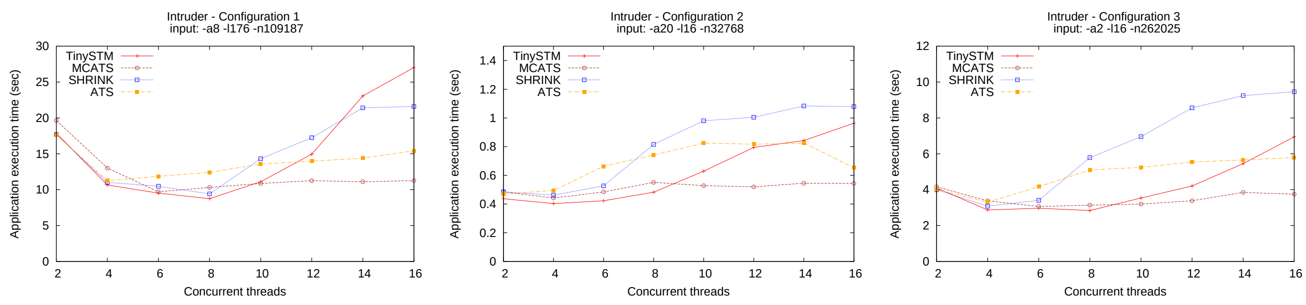

Figure 6. Performance comparison for Intruder.

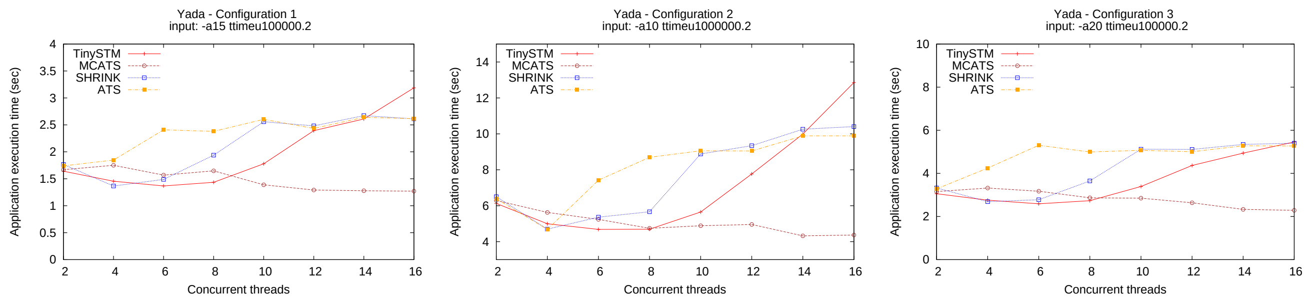

Figure 7. Performance comparison for Yada.

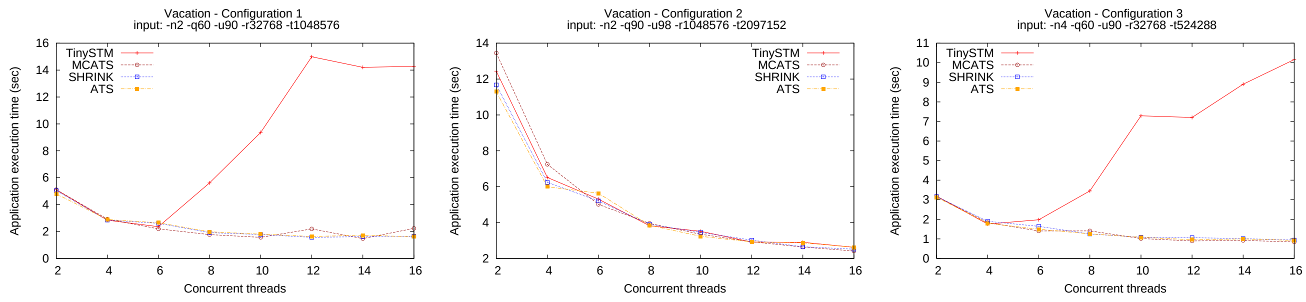

Figure 8. Performance comparison for Vacation.

the application scales up to 16 concurrent threads with all the tested schedulers and with the baseline TinySTM, which all provide very similar performance. Overall, in all the scenarios, the average performance improvement of MCATS for Intruder is about 58%, 63% and 40% compared to the baseline TinySTM, Shrink and ATS, respectively. With Yada, the performance improvements by MCATS are of about 80%, 128% and 135%, respectively. Finally, for Vacation, MCATS provides about 321% better performance with respect to the baseline TinySTM, while its is comparable with respect to Shrink and ATS. As a final observation, although Shrink and ATS represent state-of-the-art solutions for transaction scheduling, in our tests they generally did not provide exalting results with respect to the baseline TinySTM, except for two configurations of Yada. We believe that this is due to the problem of configuring the parameters of these schedulers, whose default values suggested by the authors can be optimal for some application configurations, while suboptimal for others. Conversely, the results show that MCATS does not suffer from this drawback.

VI. Conclusions and Future Work

We have presented a lightweight performance model for Software Transactional Memory based on Markov Chain formalisms, which we have then used to build an adaptive (on-line) transaction scheduler. Our scheduler controls at run-time the maximum number of transactions that are allowed to run concurrently thus avoiding thrashing phenomena due to excessive conflicts and transaction aborts. We integrated our scheduler within the open source TinySTM layer. Further, we provided experimental results, based on the STAMP benchmark suite (run on top of a 16-core HP ProLiant machine), assessing both the prediction accuracy of the Markov Chain-based performance model and the effectiveness of our scheduler compared to state-of-the-art proposals. As future work, we plan to study the applicability of our approach to Hardware Transactional Memory, and to assess our solution from the perspective of energy efficiency.

References

[1] https://gcc.gnu.org/wiki/transactionalmemory

[2] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "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)*, 2014, pp. 81–91.

[3] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015, pp. 224–233.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proceedings of the 4th IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[5] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, pp. 237–246.

[6] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008, pp. 169–178.

[7] D. Rughetti, P. di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2012, pp. 278–285.

[8] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proceedings of the 14th International Conference on Parallel Processing (Euro-Par)*, 2008, pp. 719–728.

[9] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, 2009, pp. 7–16.

[10] 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," *Performance Evaluation*, 69(5): pp. 187–205, 2012.

[11] Z. He and B. Hong, "Modeling the run-time behavior of transactional memory," in *Proceedings of the 18th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010, pp. 307–315.

[12] X. Yu, Z. He, and B. Hong, "An analytical model on the execution of transactional memory," in *Proceedings of the 22nd IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2010, pp. 175–182.

[13] A. Dragojević and R. Guerraoui, "Predicting the scalability of an STM: a pragmatic approach," in *Proceedings of the 5th ACM Workshop on Transactional Computing*, 2010.

[14] P. Di Sanzo, F. Del Re, D. Rughetti, B. Ciciani, and F. Quaglia, "Regulating concurrency in software transactional memory: An effective model-based approach," in *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2013, pp. 31–40.

[15] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Dynamic feature selection for machine-learning based concurrency regulation in STM," in *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2014, pp. 68–75.

[16] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, "Automatic tuning of the parallelism degree in hardware transactional memory," in *PRoceedings of the 20th International Conference on Parallel Processing (Euro-Par)*, 2014, pp. 475–486.

[17] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems*, ser. Lecture Notes in Computer Science, V. Gramoli and R. Guerraoui, Eds., vol. 7853. Springer Berlin Heidelberg, 2013, pp. 233–247.

[18] S. Dolev, D. Hendler, and A. Suissa, "Car-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the 27th ACM symposium on Principles of Distributed Computing (PODC)*, 2008, pp. 125–134.

[19] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel, "Txlinux: using and managing hardware transactional memory in an operating system," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 87–102.

[20] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2010, pp. 79–90.

[21] L. Kleinrock, *Queueing Systems*. Wiley Interscience, 1975, vol. I: Theory, (Published in Russian, 1979. Published in Japanese, 1979. Published in Hungarian, 1979. Published in Italian 1992.).