Adaptive Model-based Scheduling in Software Transactional Memory

Pierangelo Di Sanzo, Alessandro Pellegrini, Marco Sannicandro, Bruno Ciciani, and Francesco Quaglia

Abstract—Software Transactional Memory (STM) stands as powerful concurrent programming paradigm, enabling atomicity and isolation while accessing shared data. On the downside, STM may suffer from performance degradation due to excessive conflicts among concurrent transactions, which cause waste of CPU-cycles and energy because of transaction aborts. An approach to cope with this issue consists of putting in place smart scheduling strategies which temporarily suspend the execution of some transaction in order to reduce the transaction conflict rate. In this article, we present an adaptive model-based transaction scheduling technique relying on a Markov Chain-based performance model of STM systems. Our scheduling technique is adaptive in a twofold sense: (i) it controls the execution of transactions depending on throughput predictions by the model as a function of the current system state, (ii) it re-tunes on-line the Markov Chain-based model to adapt it—and the outcoming transaction scheduling decisions—to dynamic variations of the workload. We have been able to achieve the latter target thanks to the fact that our performance model is extremely lightweight. In fact, to be recomputed, it requires a reduced set of input parameters, whose values can be estimated via a few on-line samples related to the current workload dynamics. We also present a scheduler that implements our adaptive technique, which we integrated within the open source TinySTM package. Further, we report the results of an experimental study based on the STAMP benchmark suite, which has been aimed at assessing both the accuracy of our performance model in predicting the actual system throughput and the advantages of the adaptive scheduling policy over literature techniques.

Index Terms—Transactional Memory, Transaction Scheduling, Performance Models, Model-based Performance Optimization.

1 Introduction

Transactional Memory (TM) allows transparent synchronization of shared-data accesses by concurrent threads. It simplifies the software design/coding process. However, a core issue to tackle is related to its intrinsic speculative nature. TM keeps consistency—atomicity and isolation—by relying on transaction abort, i.e. by squashing the effects of a transactional code block each time a running instance is detected to conflict with another transaction. Abort events generate, in their turn, waste of CPU time and consequently, energy. Therefore, TM systems require to be complemented with methods aimed at reducing as much as possible the negative impact of aborts, while still enabling high exploitation of parallelism in the underlying hardware. Two main techniques have been exploited in the literature to cope with performance and energy efficiency in TM systems: *transaction scheduling* and *thread scheduling*. In the former (e.g. [1]), a TM-scheduler may delay the execution of a transaction depending on the expectation of conflicts' occurrence. Thread scheduling techniques (e.g. [2]) are instead based on controlling the number of threads used to run the application. The objective is therefore the one of regulating the thread-level parallelism so as to maximize the transaction throughput. In any case, given that the workload profile of the applications can be unknown and/or may change along the application lifetime, scheduling techniques should generally entail on-line adaptive strategies.

For what concerns Software TM (STM) systems, literature scheduling techniques [3] can be separated into two groups: (a) the ones based on performance prediction models (e.g. [2]) and (b) the ones based on heuristic approaches (e.g. [4]). Existing model-based techniques have the drawback of requiring the a-priori profiling of the applications for collecting data needed to instantiate the performance models. Heuristic-based techniques typically require the user to configure parameters (e.g. conflict rate thresholds) based on which the scheduler takes its decisions. As a consequence, the effectiveness of these techniques depends on the suitability of the selected configuration with respect to the actual workload profile.

In other cases (e.g. [5], [6]), the heuristics require to periodically alter the current system configuration to understand if better configurations exist. This may lead to work with suboptimal settings along the exploration phase.

To cope with the limitations of the existing techniques, we developed a novel performance model for STM systems, which uses a Markov Chain to capture the system evolution as a function of the number of concurrent transactions. Also, we designed an adaptive transaction scheduler that exploits the performance model for taking on-line decisions about the number of transactions to be allowed to simultaneously reside in the processing stage, with the aim of maximizing the transaction throughput. To operate, the scheduler periodically estimates at runtime a few system parameters and (re-)instantiates on-the-fly the Markov Chain-based performance model. The performance predictions by the model are then used by the scheduler for adaptively tuning the number of transactions that are allowed to run concurrently.

Compared to literature proposals, our approach offers

• *P. Di Sanzo, A. Pellegrini, M. Sannicandro and B. Ciciani are with the Department of Computer, Control, and Management Engineering, Sapienza University of Rome.E-mail: {disanzo, pellegrini, sannicandro, ciciani}@dis.uniroma1.it*
• *F.Quaglia is with Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma"Tor Vergata". E-mail: francesco.quaglia@uniroma2.it*

Manuscript received 10 Oct. 2018; revised 5 Oct. 2019; accepted 6 Nov. 2019

the following main advantages: (a) it does not require a-priori (e.g. off-line) workload profiling; conversely, profiling is carried out at runtime in a very lightweight manner (since instantiating the model requires to sample a set of only 4 parameters, which does not include transaction read/write-sets) (b) it is extremely simple to be configured, given that the only parameter to be set by the user is the time interval after which (re-)sampling the system target parameters and (re-)instantiating the performance model.

To evaluate the effectiveness of our model-based adaptive scheduler, we have implemented it within the open source TinySTM package [7] and report experimental results that demonstrate the performance advantages of our solution compared to literature schedulers for STM. Further, we present experimental data showing that our scheduler is able to counteract the detrimental effects caused by excessive transaction aborts on energy waste.

The remainder of this article is organized as follows. Related work is discussed in Section 2. In Section 3 we present an experimental study motivating the need for innovative scheduling techniques like the one we introduce. The Markov Chain-based performance model is presented in Section 4. A model validation study is reported in Section 5. In Section 6 we provide an overall discussion on our modeling approach. The model-based adaptive transaction scheduling technique is described in Section 7. The implementation of our adaptive scheduler within TinySTM and its experimental assessment are presented in Section 8.

2 Related Work

The heuristic-based scheduling techniques in [1], [4], [6], [8] use feedback control to decide how to carry out their scheduling decisions. The solution in [1] blocks transactions in order to serialize their execution when the conflict rate exceeds a pre-configured threshold. The scheduler presented in [4] dynamically changes the number of active threads with the aim of keeping the transaction conflict rate (TCR) within a target range. This technique was revised in [9] by using a specific target value for TCR (rather than a range).

The work in [8] introduces a scheduling technique inspired to the hill-climbing search, where the number of transactions allowed to run concurrently is continuously incremented or decremented in order to find the value that maximizes the throughput. The techniques in [5], [6] use a similar hill-climbing search to regulate the number of active threads.

The scheduling technique in [10] relies on the assumption of *temporal locality*, according to which the probability that a new transaction to be executed by some thread accesses the same data accessed by the most recent transactions executed by the same thread is high. Based on this assumption, the scheduler predicts data conflicts among transactions that are going to be executed, thus possibly serializing them.

In the heuristic-based techniques in [11], [12], [13], the scheduler acts in response to transaction aborts. When a transaction a is aborted due to a conflict with a transaction b that is being executed by some thread T, the schedulers in [11], [12] move a to a queue of transactions that are bound to T. This avoids that a is again executed concurrently with b, preventing repeated conflicts between them. In [13], the scheduler tracks, for each transaction, the transactions it conflicted with, and the transactions that, in turn, conflicted with them (second-hop transactions). After an abort, a transaction is serialized if the wounding transaction or a fraction of second-hop transactions are still running. In [14], the authors present a heuristic-based approach that changes the scheduling strategy depending on the average length of transactions. For short running transactions, the scheduler applies the strategy based on the transaction commit rate presented in [1]. Otherwise it decides which transactions can be executed concurrently on the basis of the recent conflict history. The proposal in [15] uses two different strategies in combination, one for short and the other for long transactions. The concurrency level of short transactions is increased/decreassed on the basis of whether the wasted time is below/above a threshold. Differently, long transactions are allowed to run concurrently if there is no intersection between their predicted write and read sets.

The reliance on heuristic methods makes all the aforementioned techniques very different from our proposal, since we use a model-based approach.

For what concerns performance models of STM applications, in the literature we find alternative proposals. The model in [16] aims at predicting the speed-up of STM applications as a function of the number of concurrent threads. This is achieved via the identification of a function that approximates the speed-up curve. The approximation is based on fitting methods that exploit a set of speed-up measurements collected by executing the application with different numbers of concurrent threads. The proposals in [17], [18], [19], [20] exploit Continuous Time Markov Chains (CTMC) to model the evolution of concurrent threads that execute transactions. Also, in [17], the execution of a transaction is modelled through a Discrete Time Markov Chain. The transaction throughput is estimated by calculating the steady state probability vector of the CTMC. All these performance models have limited (if not null) applicability to the problem of on-line performance forecasting and adaptive transaction/thread scheduling, thus mostly representing tools for off-line analysis since they require a non-lightweight and detailed workload profiling phase to be executed in advance. Also, the size of the Markov Chains used by these models grows quadratically with the number of concurrent threads and the number of operations executed by transactions, thus their calculation can become costly. Differently, in our approach we purposely developed a lightweight Markov Chain-based model with very reduced instantiation cost, whose size—which is independent of the number of operations by transactions—grows linearly with respect to the number of concurrent threads.

The model-based scheduling techniques presented in the literature use analytical models or Machine Learning (ML) models. The analytical model used by the thread scheduling technique in [21] is instantiated via regression analysis applied to a family of reference functions, based on measurements preventively collected by profiling the workload. Similarly, the neural network model used in [2] requires a wide training set built via a workload profiling phase. This technique was improved in [22] through a dynamic feature selection mechanism. The work in [23] proposes a solution for instantiating a performance model via the combination of analytical and ML approaches. Overall, all these proposals still need preliminary workload profiling phases in order to instantiate the performance models. This is a problem only partially addressed by the pure analytical approach in [21] or the mixed (analytical/ML) one in [23].

3 Preliminary Experimental Analysis

In this section, we report experimental data showing the impact of transaction concurrency on performance and energy consumption. They also demonstrate the effects of the variation of the configuration parameters in some literature STM schedulers. These effects motivate the need for more effective solutions like ours. The data we show have been collected by executing experiments with two different applications taken from the STAMP benchmark suite [24], i.e. Intruder and Vacation. We have run these applications on a 16-core machine exploiting the TinySTM package [7][1]. Each application was run with two different configurations of its input parameters, which generate different workload profiles. Figure 1 shows the variation of the normalized application execution times as a function of the number of threads. For Vacation, the thread-level concurrency exactly corresponds to the transaction-level concurrency since threads only run transactions. For Intruder, the transaction-level concurrency is a fraction of the thread-level concurrency since threads alternate between transactions and non-transactional code blocks. In any case, increasing the number of threads leads to an increase of the actual transaction-level concurrency. For each application, the curves related to the different configurations of the input parameters are quite different. For *Configuration A* of Intruder, the minimum is reached with 8 concurrent threads. With more than 8 threads the execution time increases because of an increase of the incidence of transaction abort/retry events. For *Configuration B* of Intruder, the minimum is achieved with 4 concurrent threads. As for Vacation, with *Configuration A* the execution time constantly decreases while increasing the number of concurrent threads up to 16. On the other hand, with *Configuration B* the optimal number of concurrent threads is 6.

Overall: i) TM applications' performance is highly affected by the degree of actual concurrency among transactions, ii) the optimal concurrency level changes depending on the application, and iii) it may be different when an application runs with different workload profiles. Finally, we observe that, since in general settings the workload profile may vary along the application lifetime, the optimal concurrency level among transactions may vary as well.

The level of concurrency among transactions also affects the amount of energy required for running the applications. We also report the normalized energy consumption

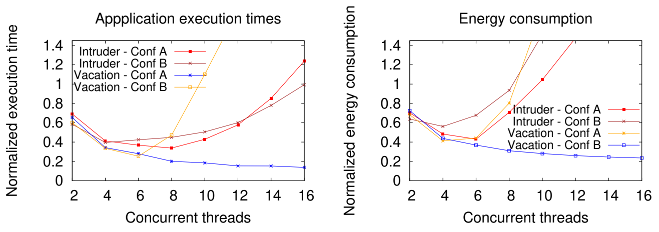


Fig. 1. Normalized application execution time and energy consumption of Intruder and Vacation. Input parameters for "Intruder - Configuration A": -a8 -l176 -n109187", Input parameters for "Intruder - Configuration B": -a20 -l16 -n32768", Input parameters for "Vacation - Configuration A": -n2 -q90 -u98 -r1048576 -t2097152", Input parameters for "Vacation - Configuration B": -n2 -q60 -u90 -r32768 -t1048576".

(see Figure 1) that we measured while running the above-mentioned benchmark applications[2]. The plot shows that the energy consumption curves are quite similar to the ones related to the execution time. However, there is one point worthy to note. If the applications run with a number of threads greater than the optimal one (in terms of performance) then the energy consumption tends to degrade even more rapidly with respect to performance—see the curves with more than 8 concurrent threads and *Configuration A*, or 4 concurrent threads and *Configuration B* of Intruder, and more than 6 concurrent threads and *Configuration B* of Vacation.

In summary, the need for solutions allowing STM applications to keep the actual transaction-level concurrency close to the optimal value is crucial for the application performance, and it is still more important for energy efficiency.

As mentioned in our literature survey, the optimization of the actual transaction-level concurrency can be addressed via transaction scheduling approaches. However, existing schedulers based on performance models have the drawback of requiring the a-priori profiling of the applications in order to build their underlying performance model. On the other hand, heuristic-based schedulers, although being more lightweight to instantiate, show an effectiveness that depends on the user ability to properly configure specific scheduling parameters—alternatively they require exploration phases that may lead the system to work with suboptimal settings for a while. To provide examples showing the effects on performance by different user-tunings of scheduling parameters, we report data referring to *Configuration A* of Intruder and Vacation for the case of two literature heuristic-based transaction schedulers, namely ATS [1] and Shrink [10], both integrated in TinySTM. As outlined in Section 2, ATS temporary blocks transactions when the abort rate oversteps a given threshold, referred to as *Contention Intensity*. Differently, Shrink uses a prediction-based strategy, and activates the scheduling algorithm only if the transaction success rate is lower than a so-called *Success* threshold. We focus on the impact on performance while varying the values of these thresholds, which can in principle be set by the user to any value in the interval between 0% and 100%. Reported data—plotted in Figure 2—are related to executions with 8 concurrent threads. Nevertheless these

1. The used machine is a HP ProLiant server equipped with two 2GHz AMD Opteron 6128 processors—each one having 8 cores, for a total of 16 cores—64 GB RAM NUMA, and Linux kernel 3.2, with support for NUMA allocation. The actual memory allocator used by TinySTM is malloc-glibc (v2.26), configured to rely on the first-touch NUMA policy offered by the kernel. When scaling up the number of threads in any experiment, the threads are pinned to CPU-cores residing as much as possible in a same NUMA node.

2. Energy consumption is related to the CPU-memory subsystem, and has been measured through the "Power Gov" tool [25].

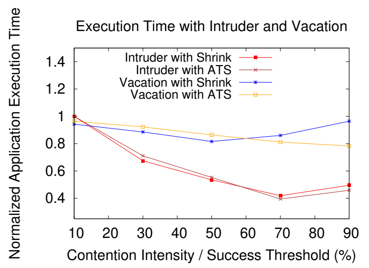


Fig. 2. Performance variation for Intruder and Vacation with different values of Contention Intensity or Success Thresholds.

outcomes are a consistent representation of what we have observed with different numbers of concurrent threads. The plot shows the execution time of Intruder and Vacation while changing the values of the Contention Intensity and Success thresholds between 10% and 90%. As we can see, the execution time remarkably changes as a function of the values of these thresholds. For Intruder, 70% is the optimal value for both thresholds. For Vacation, the optimal values of the two thresholds are different. The optimal threshold for the Contention Intensity in ATS is 90%, while the optimal value of the Success threshold in Shrink is 50%. Overall, the results show that the effectiveness of these schedulers strongly depends on the ability of the user in selecting the values of the scheduler parameters for each specific application (and its configuration). This is a non-trivial task, especially for unknown/unforeseen workloads, and may require to preventively execute various runs of an application to explore different choices. The scheduling approach we present in this article exactly aims at overcoming the need for (i) executing pre-profiling phases of the application workload and (ii) configuring (complex) scheduling parameters that may require experienced users and/or preventive studies to identify well-suited values. Also, it does not require altering the system configuration for carrying out exploration phases aimed at driving scheduling decisions.

4 The Markov Chain-based Model

4.1 Modeled Scenario

We assume an STM application running with N concurrent threads. At any point in time, a thread can execute a transaction or a non-transactional code (*ntc*) block. Hence, our model copes with general scenarios where the actual number of concurrent transactions can vary over time.

If a transaction is aborted because of conflicting data accesses with concurrent transactions, the thread executes a new run of the same transaction. The transaction execution time is therefore the elapsed time between the beginning of the first run of the transaction and the completion of its finally successful run—the one that commits. We assume the presence of a transaction scheduler that admits up to m concurrent transactions to their processing stage. Thus, a thread that starts a new transaction is blocked by the scheduler if m transactions are already being processed. We denote with k the number of threads that have started the execution of a transaction (including the blocked ones, if any) and have not yet committed the transaction. When a thread starts a transaction and $k \geq m$ the transaction is temporarily blocked, thus it enters a waiting phase. When one of the m non-blocked transactions is committed, one blocked transaction (if any) is unblocked—FIFO unblocking can be used to guarantee fairness. Clearly, the actual number of concurrently running transactions can vary over time between 0 and m, just because threads can execute either transactions or ntc blocks along time. The objective of our scheduling technique is the one of determining the optimal value of m depending on the application execution profile, which may vary over time. This is achieved by exploiting the performance model we introduce in the next section, which is aimed at predicting the transaction throughput as a function of the value of m.

4.2 The Model

We model the execution of the STM application via a discrete state-space model with $N+1$ states. In state k, with $0 \leq k \leq N$, there are k (out of N) threads processing transactions, including both running and blocked transactions. Clearly, k assumes the value 0 when none of the N running threads is currently executing a transactional code block. When the system is in state k and a thread starts a new transaction, the system transits to state $k+1$. Conversely, if one of the k threads running transactions successfully executes the commit operation, the system transits to state $k-1$. We note that in each state $k \leq m$ exactly k transactions are running. Differently, for $k > m$ then m transactions are running and $k-m$ transactions are blocked.

We assume the residence times within the states (the time spent continuously within a state after entering it) to be exponentially distributed. The validity of this assumption is analyzed in Section 6 by the means of experimental data. Based on this assumption, we model the STM application execution via a time-homogeneous Continuous Time Markov Chain (CTMC) [26]. A representation of our CTMC-based model is shown in Figure 3.

4.2.1 *State Transition Rates*

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

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

As for the transition rate from state k to $k-1$, it depends on k and m—we remark that m represents the maximum number of transactions that are allowed to run concurrently by the scheduler. 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 any state $k \leq m$, since exactly k transactions are running (i.e. none of them is blocked), the transition rate from state k to $k-1$ is

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

Conversely, for any state $k > m$, the running transactions are m, while the remaining $k-m$ transactions are blocked. Hence, for $k > m$, the transition rate to state $k-1$ is

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

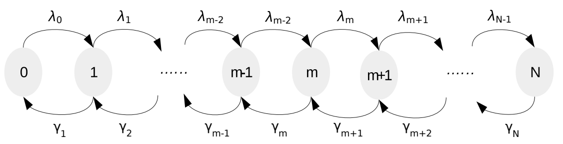


Fig. 3. The CTMC-based model.

4.2.2 Transaction Execution Time

In this section, we focus on determining the average transaction execution time t_k when the system is in state k. We note that t_k is affected by the number of times a transaction is aborted (hence re-executed) 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[3]. Thus, if p_k is the transaction abort probability when the system is in state k, we have

$$r_k = p_k/(1 - p_k) \tag{5}$$

Finally, we can safely assume that all the abort probability values p_k for $k > m$ are equal to p_m, given that in any state $k > m$, exactly m transactions are running (they are not blocked). Also, for this subset of states, we can assume that $w^r_{t,k}$ and $u_{t,k}$ are equal to $w^r_{t,m}$ and $u_{t,m}$, respectively, since none of them depends on the number of blocked transactions. Consequently, for each state $k \geq m$, we have

$$w_{t,k} = w^r_{t,m} \cdot r_m \tag{6}$$

which implies

$$t_k = w_{t,m} + u_{t,m} \tag{7}$$

and finally

$$\gamma_k = m \cdot \mu_m \tag{8}$$

For clarity, in Figure 4 we show the CTMC with transition rates expressed as a function of N, λ, m, and μ_m.

3. The reasonableness of this assumption has been already demonstrated via experimental results proposed to validate various literature analytical performance models of transactional systems (see, e.g., [18], [19], [27], [28]).

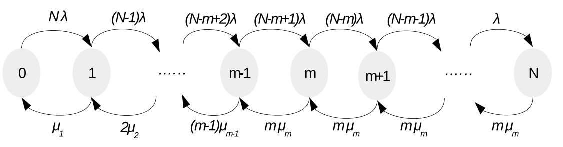


Fig. 4. The CTMC-based system model with transition rates as a function of N, λ, m, and μ_m.

4.2.3 System Throughput

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

$$\begin{aligned} thr_m &= \sum_{i=1}^{N} q_k \cdot \gamma_k = q_1\mu_1 + q_2 2\mu_2 + ... \\ &\quad ... + q_m m\mu_m + q_{m+1} m\mu_m + ... + q_N m\mu_m \end{aligned} \tag{9}$$

In order to compute q_k, with $0 \leq k \leq N$, we can use the CTMC general-equilibrium solving equations [26]

$$q_k = q_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\gamma_{i+1}} \tag{10}$$

$$q_0 = \frac{1}{1 + \sum_{k=1}^{N} \prod_{i=0}^{k-1} \frac{\lambda_i}{\gamma_{i+1}}} \tag{11}$$

We define, for any $k \leq m$

$$a_k = \prod_{i=0}^{k-1} \frac{(N-i)\lambda}{(i+1)\mu_{i+1}} \tag{12}$$

and, for any $k > m$

$$b_k = \prod_{i=m}^{k-1} \frac{(N-i)\lambda}{m\mu_m} \tag{13}$$

Hence, by Equation 10, for any state $k \leq m$ we have

$$q_k = q_0 \cdot a_k, \tag{14}$$

and for any state $k > m$ we have

$$q_k = q_0 \cdot a_m \cdot b_k \tag{15}$$

Finally, Equation 11 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 have

$$q_0 = \frac{1}{1 + \sum_{k=1}^{m-1} a_k + \sum_{k=m}^{N} a_m \cdot b_k} \tag{16}$$

By relying on Equations 12-16 we can finally calculate q_k, for any k from 0 to N. Hence, we can calculate the throughput values thr_m via Equation 9 for any value of m.

We note that instantiating our model only requires knowing the values of the four parameters listed in Table 1, which can be easily and non-intrusively measured at runtime (or even approximated as we shall discuss). Further, by construction of the CTMC-based model, once fixed a value for m, for any state $k > m$ the aforementioned parameters have the same values (given that p_k does not change for $k \geq m$). This further contributes to keep the number of observations to be collected for instantiating the CTMC-based model small.

$u_{t,k}$	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 execution time of ntc blocks

TABLE 1
Input parameters for instantiating the performance model.

5 Model Validation

In this section, we present experimental data for an assessment of the accuracy of our performance model. Further, we illustrate how the model can be used to perform what-if analysis vs the number of concurrent transactions admitted to run by the scheduler. What-if analysis constitutes a building block on top of which the adaptive scheduler we present in this article is built. In this study, we exploit three applications of the STAMP benchmark suite [24], Intruder, Yada and Vacation[4]. We selected them since they show very different workload features (as reported in [24]), such as the transaction length, the percentage of time spent in transactions (vs non-transactional code blocks), the contention intensity and the read/write set size. We have run these applications using the TinySTM open source package [7], deploying them on the 16-core HP ProLiant machine described in Section 3. We augmented TinySTM with profiling capabilities to estimate the parameters listed in Table 1 and with the possibility to admit a given maximum number of concurrent transactions to the running stage.

5.1 Throughput Prediction

In order to assess the model accuracy, we compared the predicted system throughput and the real one measured while executing the selected benchmark applications. The throughput prediction has been made at runtime, every 1000 executed transactions, by dynamically re-instantiating the model. This has been done by exploiting samples for estimating the parameters listed in Table 1, collected along the execution interval of those 1000 transactions. In these experiments we varied the number of threads N used to run the applications, and the number m of concurrent transactions admitted to the running stage.

The results are shown in Figure 5. By the plots, we observe an accurate throughput prediction by the performance model in all the tested configurations, including the one with 16 threads—corresponding to the maximum number of CPU-cores available in the underlying machine—all of them allowed to execute their transactions concurrently ($m = 16$). This is a relevant achievement when considering that, for two of the three benchmarks, $m = 16$ is a configuration leading to thrashing (repeated transaction aborts). In fact, with such settings, the execution times of both Intruder and Yada are definitely stretched (compared to settings with lower levels of transaction parallelism) just due to trashing phenomena. These phenomena, as well as more favorable runtime dynamics, are reliably captured by our performance model. Also, the predictions by the model are accurate independently of the stability of the real throughput curve associated with the different settings.

4. The corresponding configurations are "Intruder: -a8 -l176 -n109187", "Vacation: n4 -q90 -u98 -r1048576 -t4194304", "Yada: -a10 ttimeu1000000.2"

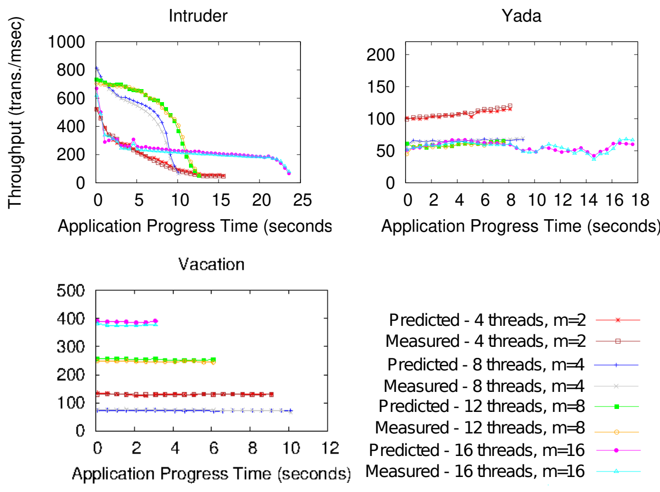


Fig. 5. Predicted vs measured throughput.

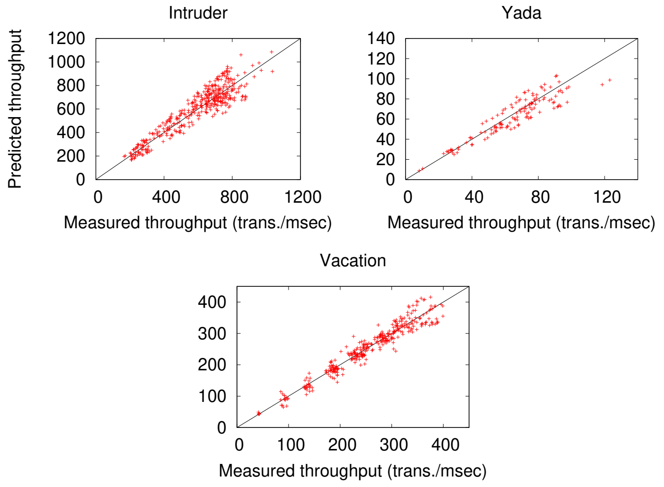


Fig. 6. Prediction accuracy.

Overall, the average relative throughput prediction error by our model (across all the configurations) falls in the intervals between 5.5% and 8.5% (with standard deviation between 0.032 and 0.071) for Intruder, 2.4% and 5.9% (with standard deviation between 0.0072 and 0.019) for Yada, and 1.3% and 3.3% (with standard deviation between 0.011 and 0.041) for Vacation.

5.2 What-if Analysis

In our adaptive scheduler, the performance model is exploited to predict the transaction throughput for any arbitrary value of m while the application is running with some other value of m (what-if analysis).

In more detail, when the application runs with $m = x$, the transaction throughput for $m = x'$ (with $x \neq x'$) is predicted by 1) measuring the values of the parameters listed in Table 1, 2) instantiating the model using the measured values, and then 3) solving the model for $m = x'$. The last step provides as output the predicted throughput for $m = x'$.

If the throughput has to be predicted for $m = x'$, with $x' > x$, the set of transaction abort probabilities

$\{p_k : x < k \leq x'\}$, which are required to solve the model for $x' > x$, cannot be determined on the basis of the measurements collected when running with $m = x$. This is because the application is not allowed to run with more than x concurrent transactions, leading to the impossibility to sample p_k with $k > x$. To estimate the missing probability values we rely on the following considerations. When there are x running transactions, 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 transactions, the probability for a transaction to experience no conflicts with 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

$$p_x = 1 - (1 - p_a)^{x-1} \tag{17}$$

Solving by p_a the above equation we have

$$p_a = 1 - (1 - p_x)^{\frac{1}{x-1}} \tag{18}$$

and if we know the transaction abort probability p_x for a generic state x, we can calculate p_a using Equation 18. 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 17.

To estimate the accuracy of what-if analysis, we performed a set of experiments with the selected benchmark applications. We ran each application by randomly changing the value of m between 1 and 16 for each interval of 1000 consecutive transaction commits. At the end of each interval, the model was instantiated using the collected measurements and was used to predict the throughput for the next interval, during which the application runs with the new randomly selected value of m. At the end of each interval, the real throughput was measured and compared with the throughput previously predicted by the model. In Figure 6, we report the scatter plots that show the prediction accuracy for all throughput values measured for all intervals along the execution of the three benchmark applications. For each point, the closer the black line, the smaller the error. The average prediction errors we measured are 10.8% (standard deviation 0.069) for Intruder, 9.9% (standard deviation 0.057) for Yada and 6.4% (standard deviation 0.064) for Vacation, which demonstrate good matching of predicted and measured throughput.

6 Discussion on the Modeling Approach

Our performance model recalls a queuing system model where: (i) the number of concurrent transactions k corresponds to the number of customers in the system, (ii) the transaction inter-arrival times correspond to the customer inter-arrival times, and (iii) the transaction execution times correspond to the service times. Based on these similarities, the most suited stochastic process to describe the evolution of the system we target in our analysis is one accounting for both the customer inter-arrival time distribution and the service time distribution. For STM systems, these distributions may change depending on the application, and, generally, they may be arbitrary. Hence, using Kendall's notation, the STM system behaves, in our analogy, like a $G/G/m/N$ queuing system, where: a) the first and the second G represent general distributions for inter-arrival times and service times, respectively, b) m represents the maximum number of concurrently-served customers (i.e. the number of transactions allowed to run concurrently by the scheduler), c) N represents the population size (the number of threads). One aspect that makes the STM system we target in our analysis more complex with respect to the above-mentioned queuing system is that the average transaction service time may change depending on state k, just because of (potentially) different conflict probabilities leading to aborts. Anyway, even neglecting this aspect, little is known about exact resolution methods for queuing systems with general inter-arrival/service time distributions.

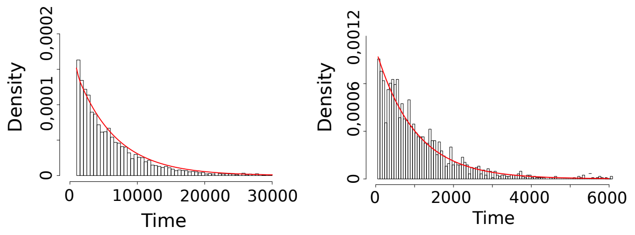


Fig. 7. Residence times (microseconds) in state $k = 8$ for Vacation and Kmeans.

On the other hand, a model easily solvable at runtime, and having closed-form solution, can be built using some assumption, like our assumption of exponential residence times in the different states.

In fact, we observed by experimental data that this assumption well matches the behavior of actual STM applications. To provide the reader with some example data, in Figure 7 we show the histograms of the state residence times that we measured with the Vacation and Kmeans applications[5], which have been still taken from the STAMP benchmark suite, running with 16 concurrent threads on our reference computing system. Data refer to the residence times in the state $k = 8$ (measured via `rdtsc`), but similar results were observed for other values of k. We also plot the calculated probability density functions (red curves) of the exponential distributions whose rate is equal to the average of the measured residence times. The shape of the histograms well reflects the plotted curve. This indicates that our modelling approach based on a CTMC is a pragmatic and effective alternative to less tractable models like the $G/G/m/N$ one. On the other hand, the suitability of the CTMC modelling choice is confirmed by the validation study of our model (see Section 5).

7 Model-based Transaction Scheduling

In this section, we describe the adaptive transaction scheduler based on the performance model presented in Section 4.2, which we call MCATS (Markov Chain-based Adaptive Transaction Scheduler). MCATS targets the maximization of the system throughput by dynamically regulating the number of concurrent transactions admitted to the running stage (which we denoted as m) along the application lifetime. MCATS works by initially setting m to some default value,

5. The corresponding configurations are "Vacation: -n2 -q90 -u98 -r1048576 -t2097152" and "Kmeans: -m10 -n10 -t0.05".

e.g., the number of threads running the application. Then, it iteratively performs the following two steps:

Step 1: Sampling. The workload is sampled for a given time interval (keeping fixed the selected value of m) so as to evaluate via measurements the four parameters listed in Table 1, whose values are needed for instantiating the CTMC-based performance model (see Section 4.2).

We remark that, since m is the maximum number of transactions allowed to run concurrently, the actual number of concurrent transactions can vary between 0 and m. However this does not guarantee that samples can be collected for every k such that $0 \leq k \leq m$. For example, this can happen when λ_k values are relatively higher than γ_k values. In fact, this causes the system to work mostly in states closer to m, rather than 0, thus reducing the possibility to collect samples for values of k closer to 0.

If no measurements are available for some state k, the following fall-back sub-step is executed: the missing values of the parameters $u_{t,k}$, $w^r_{t,k}$ are set equal to the average values of these same parameters as observed for all the other states for which measurements were available (while p_k is estimated through Equations 17 and 18). This approximation likely causes 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 under the current workload. Hence, the corresponding q_k values in Equation 9 will be low (or negligible), thus contributing in a very marginal manner to the weighted sum in that equation.

Step 2: Tuning. The throughput thr_m, for any m such that $1 \leq m \leq N$, is predicted via what-if analysis, and the maximum number of concurrent transactions admitted to the running stage by the scheduler is set to the value of m for which the predicted thr_m is maximal.

8 Experimental Assessment

8.1 Implementation Details

We developed a Posix compliant implementation of MCATS integrated within TinySTM, which is targeted at x86 platforms[6]. We used a shared global variable c, managed atomically by all the threads, to count the number of active transactions. A thread is allowed to process its current transaction only if the value of c after the increment is less than the value of m that has been lastly selected as the maximum transaction concurrency level by MCATS. Otherwise, the thread enters a spinlock phase which ends when the condition $c < m$ becomes true. The duration of the workload sampling phase (**Step 1**) corresponds to the execution of T subsequent committed transactions[7].

To collect timing-samples to estimate $u_{t,k}$, $w^r_{t,k}$ and t_{ntc}, we used the `rdtsc` machine instruction, which provides the number of CPU-cycles since the machine was started[8]. The abort probability p_k is estimated by computing the ratio between the number of aborted transaction runs and the sum of aborted and committed transaction runs when the system resides in state k. To compute the different values of the parameters that depend on k ($u_{t,k}$, $w^r_{t,k}$ and p_k), all the collected samples are managed within N distinct sets. A sample falls in the set S_k if the value of c when the sample is taken is found to be equal to k. The values managed within S_k are those that are used to estimate any parameter value related to state k of the CTMC-based model. We note that, given a value of m, the values of all the parameters depending on k for which $k > m$ are equal (see Equation 8).

8.2 Performance Results

We assessed the performance of MCATS by relying on seven (out of eight) benchmark applications taken from the STAMP suite, i.e. Genome, Vacation, Intruder, Kmeans, Labyrinth, Yada and Ssca2[9].

We exploited the computing environment already described in Section 5. For each STAMP application, we present results for three different workload profiles, determined by different configurations of the application input parameters. We refer the reader to [24] for details on the parameter-configuration space of these applications. We compared MCATS with the baseline implementation of TinySTM (which is devoid of transaction scheduling support) and with two literature schedulers, i.e. Shrink and ATS, still integrated within TinySTM[10]. For the configuration parameters of these schedulers we used the values suggested by the authors in [10] and [1], respectively. For completeness, we also report data with F2C2 [6], which follows the orthogonal approach of thread scheduling, rather than transaction scheduling[11].

The results are reported in Figure 8, where we plot the application execution time as a function of the number N of concurrent threads used to run the application, (we used the same thread-to-core pinning scheme for all the tested schedulers)[12]. Even though we explore with $N \geq 2$, MCATS,

6. Code available at `https://github.com/HPDCS/stmMCATS`

7. The value of T can impact the statistical goodness of the parameters that need to be estimated via sampling for instantiating the CTMC-based model—thus it can somehow impact the effectiveness of MCATS. However, as we have shown via model validation data, T can be set to a few thousands of transactions independently of the managed workload while still guaranteeing reliable outcomes.

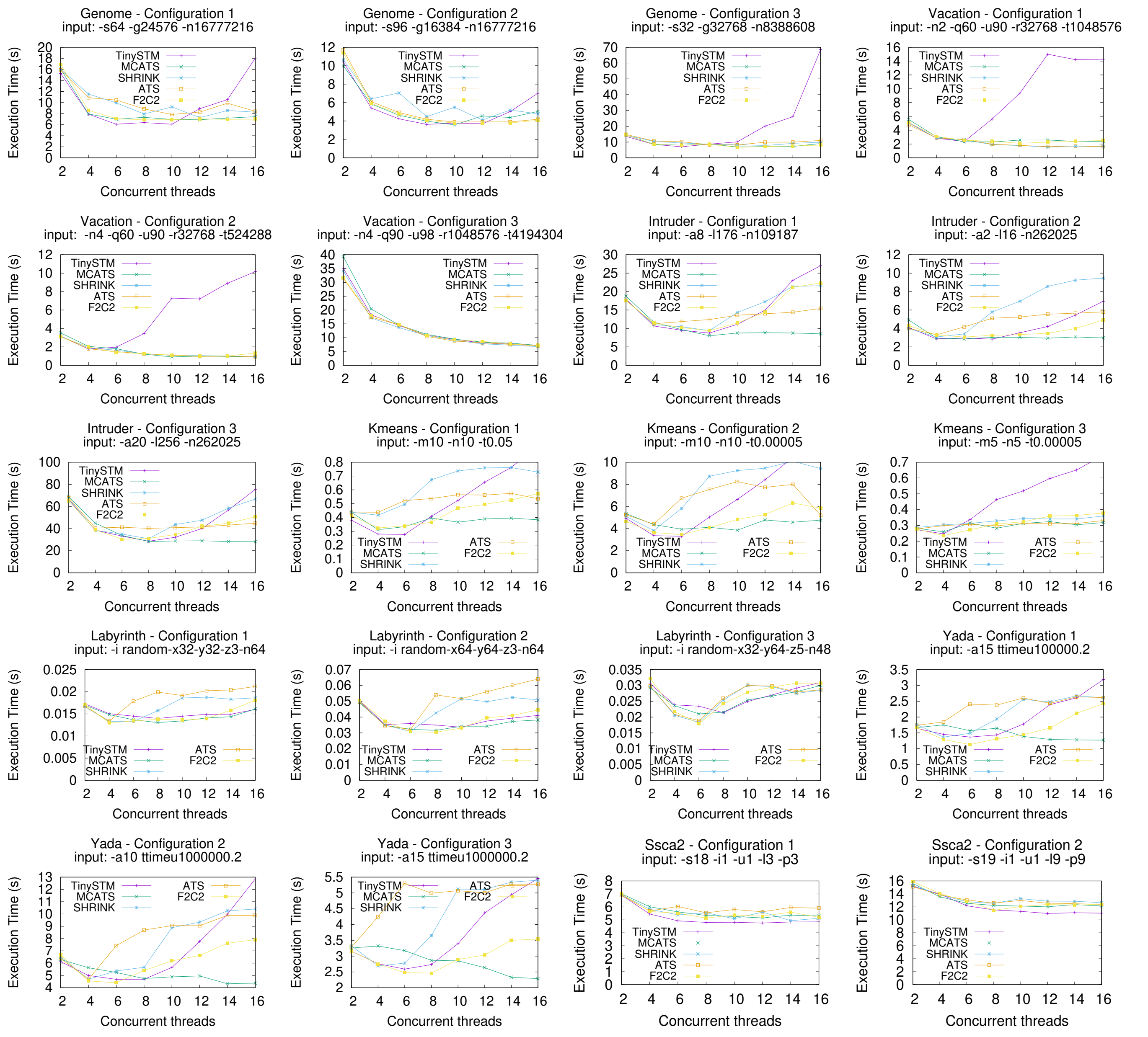
8. This approach allows measuring wall-clock-time intervals' durations. Cross-thread interference caused by time-sharing with, e.g., kernel-level housekeeping threads, can be tackled via the elimination of spike values from the statistics.

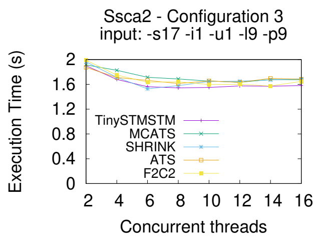
9. We excluded Bayes since, as also evidenced by the authors of STAMP [29], it shows large variability of the execution times along different runs, thus not allowing a reliable statistical evaluation of the experimental results.

10. In [14] the authors have presented a transaction scheduler that appears as a good alternative to Shrink and ATS. However, the actual advantages from this scheduler especially appear when running with more threads than CPU-cores, a configuration that we avoid since it is suboptimal because of cross-thread interference on CPU usage. For this reason we keep Shrink and ATS as reference schedulers in our study.

11. Thread scheduling is not always transparently applicable, since modifications of the application source code could be necessary to avoid anomalies, such as thread stuck [6]. This problem is particularly relevant in applications where different threads process differnet types of tasks. In fact, blocking a thread because of a scheduling decision would lead a specific type of tasks bound to the thread to remain unprocessed until the thread is eventually unblocked. Overall, the data with F2C2 are reported just to include in the analysis a scheduling technique that stands aside from the one we pursue.

12. Each reported sample is the average over 10 runs, and a maximum distance of 3% has been measured between any pairs of samples related to the same settings.





Average performance increment with MCATS vs. other techniques				
	Baseline TinySTM	Shrink	ATS	F2C2
Genome	+6%	+12%	+17%	-1%
Vacation	+321%	-10%	-9%	-2%
Intruder	+58%	+64%	+40%	+16%
Kmeans	+38%	+59%	+41%	+6%
Labyrinth	+2%	+11%	+25%	+2%
Yada	+80%	+128%	+135%	+4%
Ssca2	-8%	+2%	+3%	+3%

Fig. 8. Comparison of the application execution times with all the benchmark applications.

as well as other schedulers, might decide to allow a single transaction to be processes at any time, thus temporarily blocking the transactions that would be run by other threads. On the other hand, experimenting with $N = 1$ is of no interest. In fact, a scenario where at most one transaction can be running at anytime leads transaction schedulers to offer no useful optimization in relation to transaction conflicts and their effects on the runtime dynamics.

For almost all the test cases, the best performance (i.e. the minimum application execution time) with the baseline TinySTM is achieved with less than 16 concurrent threads—exceptions are noted for Ssca2 and for *Configuration 3* of Vacation.

In other words, there is a point beyond which increased transaction parallelism does not pay off. Therefore, in our analysis, we span between two antithetical scenarios: one where the application runs with more threads than the optimal concurrency level, and one where it runs with fewer threads than the optimal concurrency level. In the former case, a scheduler is expected to reduce the transaction conflict/abort rate in order to prevent performance loss. Conversely, in scenarios with fewer threads than the optimal number, the scheduler should not penalize performance by temporarily blocking (too many) transactions. When running with more threads than the optimal concurrency level, MCATS avoids the performance loss of the baseline TinySTM. In fact, for those application configurations where the execution time with the baseline TinySTM remarkably grows (i.e. all the configurations of Genome, Intruder, Kmeans and Yada, and *Configurations 1* and *2* of Vacation), MCATS keeps the execution time close to the values achieved with the optimal number of threads, independently of the actual number of threads used to run the applications. The other two schedulers, ATS and Shrink, do not consistently achieve the same results. For example, with all the configurations of Intruder, Labyrinth and Yada, both ATS and Shrink show poor performance and, in some cases, execution times that are even worse than those of the baseline TinySTM. Generally, ATS and Shrink ensure better performance than the baseline TinySTM only in scenarios where the number of threads is relatively high (see, e.g., Intruder and Kmeans). One motivation for these non-optimal results stands in the setting of the configuration parameters of these schedulers, whose default values suggested by the authors are suitable for some application profiles, but are largely sub-optimal for others. In other words, for these schedulers there is no one-size-fits-all settings of their parameters. This imposes to pre-tune them depending on the specific workload (if known). MCATS overcomes this drawback since it is based on a runtime adaptive strategy that allows tailoring its scheduling decisions on the basis of the current workload profile.

We note that a similar problem of performance degradation when the number of threads is larger than the optimal concurrency level is noted for F2C2, as for the case of *Configuration 1* of Intruder, *Configuration 1* of Kmeans, and all the configurations of Yada. This shows how MCATS provides a better way of contrasting thrashing phenomena caused by excessive transaction aborts in these scenarios.

As for the scenario where the used number of threads is smaller than the optimal number, all the schedulers, including MCATS, give rise to execution times that are (slightly) worse than the baseline TinySTM. Essentially, this is due to the overhead caused by the implementation of the schedulers, and to the fact that, with lower than optimal concurrency, a transaction scheduler has no real opportunity to improve performance via transaction parallelism regulation and conflicts/aborts avoidance. The only exception is for F2C2 in a few cases (like for example *Configuration 1* of Yada), just thanks to its very reduced overhead. However, MCATS shows execution time no more than 11.3% worse than the baseline TinySTM, a value only reached with Yada.

Figure 8 also shows (in tabular form) the average performance improvement of MCATS for each application—evaluated across all the different values of the used number of threads. We list the percentage speedup by MCATS with respect to the baseline TinySTM, ATS, Shrink and F2C2. MCATS shows better speedup for 23 out of 28 test cases, and up to the value 321%. On the other hand, worse speedup—which is although limited to no more than 10%—is observed for the other three cases, which are essentially related to scenarios where the applications run with under-parallelism (namely, when performance would still increase with additional threads because of the scarce incidence of transaction conflicts and consequent aborts). As said, in such a situation no transaction scheduler has opportunities to regulate transaction parallelism for consistently optimizing performance via the avoidance of thrashing.

8.3 Energy Efficiency

We report in Figure 9 results for a comparison of the energy consumption achieved with MCATS and with the baseline TinySTM[13]. For brevity, we only show results referring to *Configuration 1* of each considered application. They are representative of what we observed with other configurations. Energy consumption trends are similar to the already discussed performance trends. In more detail, without a scheduler, the energy consumption of an STM system generally grows very fast when using more threads than the optimal number (see the plots for Genome, Vacation, Intruder, Kmeans and Yada). Conversely, with MCATS the energy consumption is not (or marginally) influenced when increasing the number of threads beyond the optimal value.

We conclude this section providing data revealing that the negative impact of over-parallelism on energy efficiency in STM systems can be even more drastic than on performance, a phenomenon that is effectively counteracted by MCATS. The plots in Figure 10 show the difference between the normalized energy consumption and the normalized execution time of each benchmark application. Specifically, each curve is calculated as $E_n(x) - T_n(x)$, where $E_n(x)$ is the normalized energy consumption, x is the number of concurrent threads and $T_n(x)$ is the normalized execution time (normalization is made with respect to the energy consumption/execution time with one thread). When the curve grows (or decreases) it means that the energy consumption grows more (or less) rapidly than the execution time. With five out of seven applications the baseline TinySTM shows a point beyond which the curves rapidly grow. With MCATS,

13. These measures have been collected by using "Power Gov" tool [25].

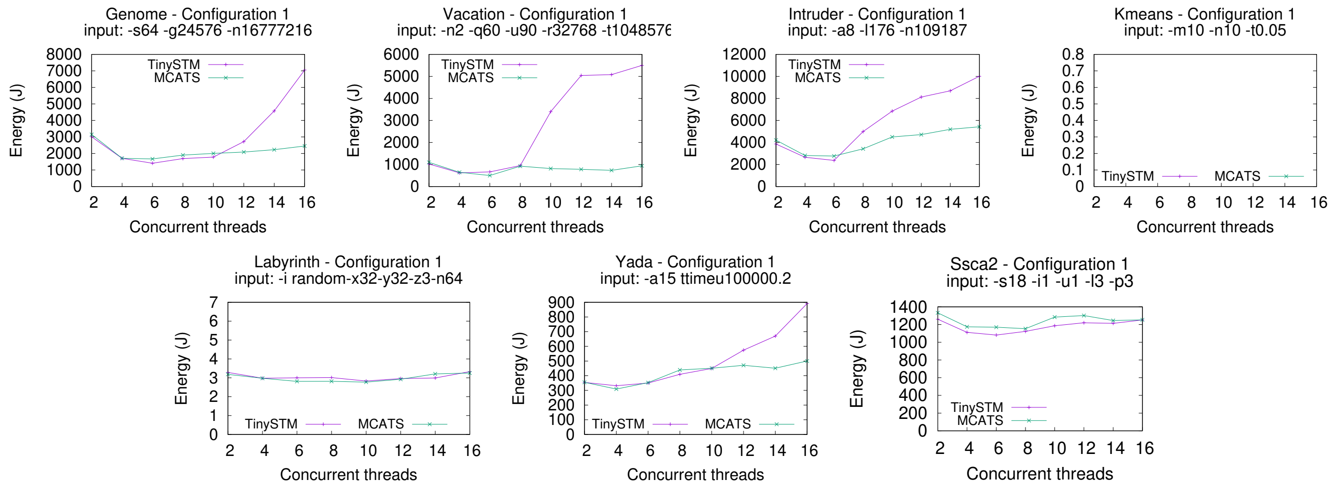


Fig. 9. Energy consumption (Joule) with different benchmark applications.

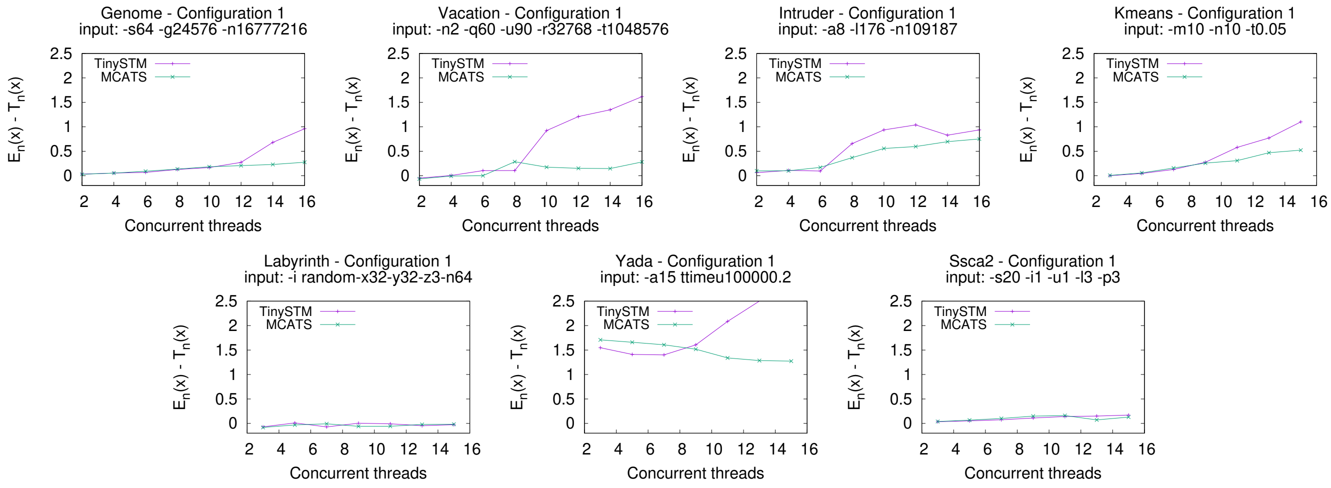


Fig. 10. Difference between normalized energy consumption and normalized execution time with different benchmark applications.

this phenomenon tends to disappear. In the worst cases (e.g. Intruder and Kmeans) the energy consumption growth is mild and constant. In the other cases, there is no noticeable increment. Such a positive impact on energy consumption by MCATS in over-parallelism scenarios is essentially due to the reduction of the application execution time.

9 Conclusions

We presented an adaptive model-based scheduler for Software Transactional Memory, which controls transaction concurrency by regulating the maximum number of transactions that are allowed to simultaneously run. One of the key aspects of our proposal is that the performance model can be easily (re-)instantiated on-the-fly, therefore allowing its continuous adaptation to the workload profile. We demonstrated the effectiveness of our approach via an experimental study carried with a real implementation of our scheduler and a set of benchmark applications. Results show that our technique is effective over a wide range of workload profiles in terms of improvement of both performance and energy efficiency.

References

[1] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 169–178.

[2] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based thread-parallelism regulation in software transactional memory," *J. Parallel Distrib. Comput.*, vol. 109, pp. 208–229, 2017.

[3] P. Di Sanzo, "Analysis, classification and comparison of scheduling techniques for software transactional memories," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3356–3373, dec 2017.

[4] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, and I. Watson, "Adaptive concurrency control for transactional memory," presented at MULTIPROG *1st Workshop Programmability MultiCore Comput.*, 2008.

[5] 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.

[6] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for STMs," in *Proceedings of the 28th Int. Parallel and Distributed Processing Symp.*, 2014, pp. 927–938.

[7] 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*, 2008, pp. 237–246.

[8] K. Chan, K. Tin Lam, and C.-L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," in *Proceedings of the 10th PDCN*, 2011, pp. 91–98.

[9] 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 on Parallel Processing*, 2008, pp. 719–728.

[10] 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*, 2009, pp. 7–16.

[11] 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*, 2008, pp. 125–134.

[12] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009, pp. 4–18.

[13] D. Sainz and H. Attiya, "Relstm: A proactive transactional memory scheduley," in *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013, pp. 1–8.

[14] D. Niccio, A. Baldassin, and G. Arajo, "Transaction scheduling using dynamic conflict avoidance," *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 89–110, 2013.

[15] H. Rito and J. a. Cachopo, "Adaptive transaction scheduling for mixed transactional workloads," *Parallel Comput.*, vol. 41, no. C, pp. 31–49, Jan. 2015.

[16] A. Dragojević and R. Guerraoui, "Predicting the scalability of an STM: A pragmatic approach," *5th ACM Workshop Transactional Comput.*, 2010.

[17] 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*, pp. 187–205, 2012.

[18] Z. He and B. Hong, "Modeling the run-time behavior of transactional memory," in *Proc. 18th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 307–315.

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

[20] A. Heindl and G. Pokam, "An analytic model for optimistic STM with lazy locking," in *Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, 2009, pp. 339–353.

[21] 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*, 2013, pp. 31–40.

[22] 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*, 2014, pp. 68–75.

[23] ——, "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*, May 2014, pp. 81–91.

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

[25] 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-power-governor.

[26] L. Kleinrock, *Queueing Systems*. Wiley Interscience, 1975, vol. I: Theory.

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

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

[29] W. Ruan, Y. Liu, and M. Spear, "Stamp need not be considered harmful," *9th ACM Workshop on Transactional Comput.*, 2014.

Pierangelo Di Sanzo received a M.S. degree and a Ph.D. degree in Computer Engineering from Sapienza University of Rome. He worked as a researcher at the Italian National Interuniversity Consortium for Informatics. Currently he is a postdoctoral researcher at DIAG (Sapienza University of Rome). His research interests lie in the area of concurrent programming and transactional systems.

Alessandro Pellegrini received a B.S. degree and a M.S. degree and a PhD in Computer Engineering at Sapienza, University of Rome. His main research area is on parallel and distributed architectures and applications, where he has published more than 50 technical articles. In 2015 he won the Sapienza prize for the best PhD thesis of the year. He has worked as a researcher at some national and international research centers (CINI, CINFAI and IRIANC).

Marco Sannicandro received a B.S. degree and a M.S. degree in Computer Engineering from Sapienza University of Rome. His master thesis was focused on performance modelling of STM systems. His research interests include distributed and parallel computing systems, cybersecurity and artificial intelligence.

Bruno Ciciani is Full Professor of Computer Engineering at Sapienza University of Rome. His research interests include fault tolerance, computer architectures, distributed systems, manufacturing yield prediction, performance and dependability evaluation. In these fields he has published more than 130 papers. He spent more than two years as visiting researcher at the IBM Thomas J. Watson Research Center (N.Y.).

Francesco Quaglia received his MS in Electronic Engineering in 1995 and his PhD in Computer Engineering in 1999, both from Sapienza University of Rome, where he has worked as Assistant Professor and then Associate Professor from September 2000 till June 2017. Currently he works as a Full Professor at the University of Rome Tor Vergata. His research interests include parallel/distributed computing systems, operating systems, and HPC.