# Analysis, Classification and Comparison of Scheduling Techniques for Software Transactional Memories

Pierangelo Di Sanzo

**Abstract**—Transactional Memory (TM) is a practical programming paradigm for developing concurrent applications. Performance is a critical factor for TM implementations, and various studies demonstrated that specialised transaction/thread scheduling support is essential for implementing performance-effective TM systems. After one decade of research, this article reviews the wide variety of scheduling techniques proposed for Software Transactional Memories. Based on peculiarities and differences of the adopted scheduling strategies, we propose a classification of the existing techniques, and we discuss the specific characteristics of each technique. Also, we analyse the results of previous evaluation and comparison studies, and we present the results of a new experimental study encompassing techniques based on different scheduling strategies. Finally, we identify potential strengths and weaknesses of the different techniques, as well as the issues that require to be further investigated.

**Index Terms**—Transactional Memory, Transaction Scheduling, Concurrent Applications, Performance Optimization.

## 1 Introduction

The recent proliferation of multi-core systems has exacerbated the need for developing concurrent applications allowing to exploit hardware parallelism. In this context, Transactional Memory (TM) [1] emerged as an effective programming paradigm, and has largely drawn the attention of both academia and industry. Retrieving the idea of database transactions, the TM paradigm was born with the aim of providing programmers with a level of abstraction to hide the thread synchronization complexity. Traditionally, programmers use mechanisms such as locks, semaphores or monitors, to avoid interferences between concurrent threads which execute code blocks performing shared data accesses (e.g. critical sections). However, this kind of approach is error-prone, and the correctness of the code is complex to verify, particularly when fine-grained synchronization is required for improving scalability. TM simplifies the development of concurrent applications, allowing programmers to use transactions to synchronize threads while accessing shared data. At run-time, the TM implementation layer ensures atomic and isolated executions of transactions, and transparently implements fine-grained data access synchronization.

Software Transactional Memories (STMs) [2] represent the implementation via software libraries of the TM paradigm. After the release of the first popular STM, called DSTM [3], which is dated back to 2003, TM appeared as an interesting approach, encouraging a vast number of research studies. Notably, various studies focus on scheduling techniques for STMs. Indeed, STMs may suffer from performance degradation due to transaction conflict rate. In STMs, transactions are speculatively executed, and conflicts on concurrent data accesses are resolved by aborting and restarting one of the conflicting transactions. This speculative approach allows the application performance to take advantage of hardware parallelism of multi-core architectures. On the other hand, when the conflict rate grows up, the wasted time due to processing of aborted transactions might increase to the point that the performance no longer benefits from additional transaction parallelism. Even, this phenomenon may lead to *thrashing*, a situation where the performance drastically goes down due to the large amount of wasted work caused by excessive transaction aborts.

Scheduling techniques aim at optimizing the performance of TM applications by proactively controlling the transaction concurrency. A specific component called *scheduler*, which is integrated in the TM system, implements a *scheduling strategy* for deciding when a transaction, or a thread executing transactions, is allowed to run. Over the last decade, the variety of choices in the design of scheduling techniques encouraged researchers to explore many alternative solutions, leading to the development of several and very different techniques. Today, literature offers techniques based on various types of strategies, such as feedback control strategies [4], [5], [6], [7], [8], [9], prediction-driven strategies [10], [11], reactive strategies [12], [13], [14], and strategies based on performance models of STM applications [15], [16], [17], [18]. Some techniques use transaction scheduling strategies [4], [10], [11], while other techniques use thread scheduling strategies [5], [15]. Further, scheduling techniques differ in the target set of transaction features [10], [19] and/or system performance parameters of TM applications [4], [15], [20].

This article offers a detailed literature analysis, and it classifies and compares scheduling techniques for STMs. We provide a specific description of each technique, and we discuss the outcomes of the existing evaluation and comparison studies. Further, we present the results of a new experimental study that we conducted to compare

• *P. Di Sanzo is with the Department of Computer, Control, and Management Engineering, Sapienza University of Rome.*
*E-mail: disanzo@dis.uniroma1.it.*

techniques that use the different scheduling strategies that we identify in our classification. Finally, we identify the potential strengths and weaknesses of the different types of strategies, the limitations of the existing techniques and the current open issues.

This article sums up the outcome of research work on STMs also to support future research on scheduling techniques for Hardware TM (HTM). Indeed, the TM hardware support recently offered by some commercial processors (as Intel Haswell processors [21]) is encouraging research also on scheduling techniques for HTMs. Notably, even though HTMs differ in some basic aspects from STMs, research work on scheduling techniques for STMs can represent an important reference point for HTMs too.

The rest of this article is structured as follows. Section 2 focuses on the background. In Section 3, we present a classification of the scheduling techniques and we describe the characteristics of the different classes. Section 4 presents a literature analysis. In Section 5, we discuss the results of the previous experimental studies that compare different scheduling techniques. The results of our experimental study are presented and analysed in Section 6. Section 7 reports the overall conclusions and the current open issues about scheduling techniques for STMs. Finally, in Section 8, we introduce and discuss the case of HTMs.

## 2 Background

This section provides the reader with the background on the TM programming paradigm and on scheduling techniques in TMs. Then, we provide a brief overview of the most used benchmark applications in literature studies on TMs.

### 2.1 The TM Programming Paradigm

As discussed, the TM programming paradigm originated from the need of simplifying the development of concurrent applications, where it is required to synchronize threads while concurrently accessing shared data. TM provides programmers with a simple interface allowing to use transactions as a synchronization construct. The programmer simply uses the notation *ATOMIC{}* (or a couple of statements like *TM_BEGIN* and *TM_END*) to mark code blocks to be executed as atomic and isolated transactions. The set of shared data read (written) by a transaction is called *read-set* (*write-set*). Two concurrent transactions conflict if the read-set or the write-set of a transaction intersects the write-set of the other transaction. Conflicts are automatically handled in TM. STMs use specific mechanisms to detect conflicts, such as *read/write locks* or *read validation* [22]. A conflict is resolved by aborting and restarting one of the conflicting transactions. The component in charge of resolving conflicts is called Contention Manager (CM). STMs typically ensures *opacity* [23], an isolation criterion that is stronger than *serializability*, which is the one traditionally used in database systems.

### 2.2 Scheduling in TMs

Transactions that conflict and get aborted carried out wasted work. Scheduling techniques use specific strategies to prevent conflicts and, consequently, to reduce the amount of wasted work. In the case of transaction scheduling strategies (we refer to as *transaction scheduling*), the scheduler can temporarily block the execution of transactions to prevent potential conflicts. In the case of thread scheduling strategies (we refer to as *thread scheduling*), the scheduler can change the number of active threads executing transactions to optimize the transaction concurrency. Basically, as opposed to conflict resolution, the scheduling approach is proactive [10], since it aims at preventing conflicts rather than resolving them after they occurred.

Scheduling techniques for STMs use online strategies, since the workload profile of a TM application may be unknown in advance. Also, given that the workload profile may change along the application execution, scheduling techniques often rely on adaptive strategies, where the scheduler is sensitive to variations, e.g., of the workload profile and/or of some application performance parameters (e.g. application throughput).

From a theoretical perspective, the scheduling problem in TM has been principally studied to analyse the *competitive ratio* [24], i.e. the worst-case ratio between the time for executing a set of transactions with an online scheduler and the time with an optimal, clairvoyant scheduler that knows in advance all characteristics of the (future) workload (such as start time, execution time and read-/write-sets of all transactions). As an example, in [24] it has been demonstrated that a lower bound for the competitive ratio of a deterministic transaction scheduler is $\Omega(s)$, where $s$ is the number of shared data accessed by transactions. Subsequently, theoretical studies have been extended to the case of some contention management and scheduling algorithms (e.g. [10], [25], [26]). In some cases, the achieved results have been used to understand theoretical limits of some scheduling strategies (e.g. in [10]).

### 2.3 Benchmarks

Various benchmark applications for STMs exist, including both *Micro-benchmarks* and *Complex Benchmarks*. Micro-benchmarks are applications where transactions execute simple operations on common data structures (e.g. insertion, extraction and deletion of elements of a data structure). Examples of micro-benchmarks include RBTree, Hash Table, Sorted Linked List, RandomGraph and LFU-Cache. Complex benchmarks mimic the behaviours of applications where transactions execute more elaborated operations. Some of the most popular complex benchmarks are: (i) *Lee-TM* [27], an application which builds circuit maps for printed boards using the Lee's routing algorithm [28], (ii) *STAMP* [29], a benchmark suite with eight applications in different domains (Bayes - bayesian network, Genome - gene sequencing, Intruder - network intrusion detection, Kmeans - data clustering, Labyrinth - path routing in a maze, Scca2 - efficient graph representation, Vacation - travel reservations, and Yada - Delaunay mesh refinement), (iii) *STMBench7* [30], a benchmark derived from CAD/CAM applications, where transactions read and update data structures consisting of a set of graphs and indexes.

## 3 Classification and Features of Scheduling Techniques

We classify scheduling techniques on basis of the adopted type of scheduling strategy. A classification scheme is shown in Figure 1. In the following, we describe the peculiarities of the different types of techniques of our classification.

***Heuristic-based Techniques***. These techniques use heuristic strategies, i.e. strategies based on methods (also said *heuristics)* that aim at "guessing" the "right" decisions. Thus, they do not guarantee to converge to the optimal solution. However, given the number and the complexity of factors involved in the scheduling problem, heuristic methods often represent a practical and low-overhead solution. Heuristic-based techniques can be distinguished as follows:

- *Feedback-driven techniques* are based on *feedback-driven* strategies and implement the typical *closed-loop* scheme used for controlling the state or the output of a dynamic system [31]. A target performance parameter of the application (e.g. transaction throughput [7] or transaction commit ratio [6]) is routed back as input to the scheduler and is used for deciding the action to be performed. As an example, in the thread scheduling technique proposed in [7], the scheduler monitors the throughput of the application and uses a strategy inspired by the hill climbing search [32]. Specifically, it continuously increments or decrements the number of active threads in order to find the configuration ensuring the highest throughput.
- *Prediction-driven techniques* use *predictive* strategies for increasing the probability of making the right decision. As an example, in [10] the scheduler makes decisions on the basis of the predicted read-/write-sets of transactions that are going to run. It relies on the assumption of *temporal locality* of transaction data access patterns, i.e. there is a high probability that data sets that will be accessed by new transactions reflect data sets accessed by recently executed transactions.
- *Reactive techniques* adopt *reactive* strategies, where the scheduler comes in action after that a transaction has been aborted to avoid *repeated* conflicts. For example, if a transaction $a$ gets aborted due to a conflict with a transaction $b$, the scheduler delays the re-start of $a$ until $b$ completes. These types of strategies rely on the assumption that data accessed by an aborted transaction are likely the same that will be accessed during the subsequent transaction re-execution. In such a case, restarted transactions may likely conflict with the same (still running) transactions with which they have conflicted in the past. An example of reactive technique is presented in [13], where the scheduler changes the execution order of conflicting transactions to serialize them after a conflict.
- *Mixed heuristic-based techniques* use *mixed* strategies for taking advantage of different heuristics in front of different transaction profiles. For example, the scheduler proposed in [19] uses a lightweight feedback-driven technique when the average length of transactions is below a given threshold. If the length overcomes the threshold, it switches to a more complex heuristic, which is based on the estimation of the conflict probability between subsets of transactions.

***Model-based Techniques***. These techniques use strategies based on performance models of the applications. These models allow to calculate performance indicators of an application (e.g. transaction throughput) depending on: (i) a set of workload features (such as the transaction profiles), and (ii) one or more parameters controlled by the scheduler. An example of such a scheduling parameter is the maximum number of transactions [18] or threads [15] admitted to concurrently run. At run-time, the scheduler uses the performance model for *what-if* analysis purposes, i.e. to estimate, based on the current workload profile, how the performance of the application would change as a function of the controlled parameter(s). This allows the scheduler to find out the parameter configuration which is expected to provide the best performance. Model-based techniques can be distinguished on the basis of the nature of the performance models, i.e:

- *Machine Learning-based techniques* use performance models build on Machine Learning (ML) methods [33]. A *learning* algorithm takes as input a dataset which (partially) represents existing relations between the workload profile (input of the model) and the target performance parameters (outputs of the model). Then, the algorithm generates an instance of a ML model allowing to calculate these parameters as a function of inputs.
- *Analytical Model-based techniques* rely on performance models built via mathematical equations (e.g. [34]), which are solved at run-time to calculate the target performance parameters.
- *Mixed Model-based techniques* use models built via different modelling approaches. An example is the technique presented in [35], which uses both an analytical or a ML-based performance model.

Besides the scheduling strategy, there are other features it is worth considering when analysing scheduling techniques for STMs, such as the *input features* and the *tuning features*.

***Input features***. Scheduling techniques may use input features to monitor at run-time the state of the system, allowing to implement adaptive strategies. Typically, these features include performance parameters of the application (e.g. transaction commit rate, transaction execution time, wasted processing time due to aborted transactions) and/or specific information related to the workload profile (e.g. read- and write-set of transactions).

***Tuning features***. Scheduling techniques may offer tuning features, allowing the user to tune some parameters which affect scheduling decisions. As an example, in some techniques ( [4], [10]) the user can set a reference threshold for the transaction conflict ratio, based on which the scheduler decides whether activating the scheduling policy or not. Other techniques (e.g., [5]) require the user to set the length of a sample interval for timing the gathering of run-time statistics. Tuning features can be used by the user to optimize the behaviour of the scheduler in front of different workload profiles.

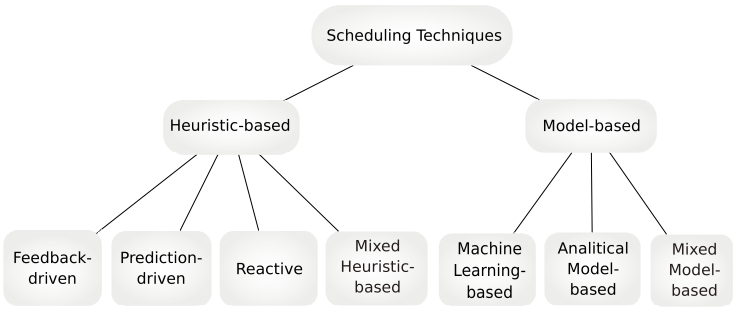


Fig. 1. Classification of Scheduling Techniques.

## 4 Overview and Analysis of Scheduling Techniques

In this section, we present and discuss the scheduling techniques proposed in literature. We follow the classification scheme we introduced in Section 3. For each technique, we also present the results of the experimental studies that compare the performance of the scheduler with the *baseline*. The latter refers to the base STM implementation devoid of scheduling support, where the scheduler has been integrated for experimental purposes. Experimental studies that compare different techniques are discussed in Section 5. Relevant details about all scheduling techniques that we mention are summarized at the end of this section in Table 1.

### 4.1 Feedback-driven Techniques

***ATS***. *Adaptive Transactional Scheduling* (ATS) [4], [36] is the first transaction scheduling technique for STMs. ATS uses the *Contention Intensity* (CI) as a feedback parameter. CI is calculated individually for each thread using a dynamic average. Specifically, upon the thread commits or aborts a transaction, it is calculated as: $CI = \alpha \cdot CI_{prev} + (1-\alpha) \cdot CC$, where $CI_{prev}$ is the previous value of CI, and $CC$ is set equal to $0$ (1) if the transaction commits (aborts). $\alpha$ has a value between 0 and 1, and affects the weight of the past transaction execution history of the thread. When CI is above a given *CI threshold*, the scheduler serializes transactions executed by the thread with respect to transactions executed by other threads. Transactions to be serialized are inserted in a global shared queue.

*Experimental study*. ATS was implemented in RSTM [37], a C++ TM library supporting transactions on shared objects. Experiments were conducted on a 2-core and a 8-core machine, using five micro-benchmarks: RBTree, HashTable, LinkedList, RandomGraph and LFU-Cache. Each application was run with up to 32 thread. CI threshold was set equal to 0.5. With the 2-core machine, $\alpha$ was set equal to $0.3$, while, with the 8-core machine, equal to $0.5$. These values are the ones that provided the best average performance. Average results, calculated over all runs of all applications, show that with the 2-core machine ATS provides a speed-up between 1.3 and 1.5 while changing the number of concurrent threads. With the 8-core machine, the speed-up is between 1.1 and 1.4. However, ATS does not provide benefits for all applications. In the case of LinkedList, the performance is penalized. Additionally, overall results show that performance improves only when using more threads than the number of available cores.

***Ansari et al. 2008*** and ***PoCC***. The first thread scheduling technique was proposed by Ansari et al. [5]. This technique aims at controlling the *Transaction Commit Ratio* (TCR), i.e. the fraction of committed transactions over the total number of executed transaction runs. The scheduler changes the number of active threads in order to keep TCR within a *target range*. A parameter named *sample interval* (SI) establishes the length of a single step of the control loop executed by the scheduler. For each step, the scheduler calculates TCR, then decrements (increments) the number of active threads if TCR is above the upper (below the lower) threshold of the target range. The authors investigate four different policies. The first policy (called *SimpleAdjust)* activates or deactivates one thread per step and keeps SI fixed. The second policy (*ExponentialInterval)* extends the first one by halving the SI if TCR is out of the target range, otherwise SI is doubled. The third policy (*ExponentialAdjust)* changes the number of active threads proportionally to the difference percentage between the TCR and the target threshold when TCR is out of the target range. The last policy (*ExponentialCombined)* combines ExponentialInterval and ExponentialAdjust.

In [6], Ansari et al. proposed another policy, P-only Concurrency Control (PoCC), with the aim of improving responsiveness and robustness of the scheduler in front of TCR variations. PoCC uses a target value (*setPoint*) for TCR, rather than a range. Further, the number of threads to be activated/deactivated for each step changes proportionally to the current number of active threads.

*Experimental study*. The proposed scheduler was implemented in DSTM2 [3], a JAVA based STM which offers various contention management algorithms. The experimental study with the four original policies was conducted using Lee-TM. It was run on a 4 x dual-core machine, using 30% and 60% as range thresholds and 20 seconds as initial value of SI. Runs were executed with 1, 2, 4 and 8 threads. Results show, on average, a speed-up between 1.14 (with ExponentialInterval) and 1.18 (with ExponentialAdjust), a 0.76 speed-up as a worst case and 2.47 as a best case. PoCC has been evaluated also with Genome, Kmeans, Vacations and StepChange are. In this case, *setPoint* was set equal to 70%, and SI = 1s. Results show that, for all execution scenarios, PoCC provides a speed-up similar to the best policy among the ones proposed in the previous study. With respect to the best case when running the application with a static number of threads, PoCC shows a slowdown equal to 5%. However,

results show that the main advantage of PoCC is the ability to reduce variance of the application execution time over multiple runs of the application (variance is reduced up to 31% with respect to the best case with the other policies). Further, PoCC reduces the average resource usage (by 24%) and the wasted work (by 16%) over the next best policy, i.e. ExponentialCombined.

***Chan et Al. 2011***. The work presented in [7] proposes a technique that controls the maximum number of transactions, say $m$, allowed to concurrently run. When a thread is going to run a new transaction, it stalls if there are $m$ running transactions. The authors present two alternative policies to tune $m$. *Throttle* policy uses the transaction commit ratio. At the end of each sample interval, it decrements $m$ if the ratio drops below a specific *threshold*. Conversely, $m$ is decremented when the ratio overcomes the threshold and at least one thread has stalled during the previous sample interval. *Probe* policy uses the transaction commit rate (transaction throughput) and the hill climbing search that we already mentioned in Section 3. Specifically, by continuously incrementing and decrementing $m$ it tries to discover the value for which the throughput curve has a peak.

*Experimental study*. The scheduler was implemented in TinySTM. Experiments were conducted on top of a machine with 2 x quad-core processors with hyper-threading, using STAMP benchmark suite. By experimental results, there is no significant performance improvement with respect to the baseline when the number of threads is less or equal to the number of physical cores, i.e. 8., except for one application (Bayes). Really, for some applications the throughput is penalized. On the other hand, with more threads than physical cores (where generally the throughput curve drops down due to contention on physical resources) the scheduler reduces the performance loss. Average results over all scenarios demonstrate that Probe performs better than Throttle, providing a performance improvement of 12.4% vs. 11.1%.

***Weighted Adaptive Concurrency Control***. More recently, Ansari presented another TCR-based technique, *Weighted Adaptive Concurrency Control* [8], a technique that selects *which* thread have to be activated or deactivated depending on the their own TCR. Particularly, he introduced the notion of *expected TCR*, i.e. the global TCR that would be achieved when only threads of a given subset are active. The author investigates four policies to find the subset with the expected TCR as much as close to *setPoint*: 1) *WBEST*: this policy sorts threads by individual TCR, then incrementally selects threads to be deactivated (activated) starting from the one with the lowest (higher) TCR, until the expected TCR starts diverging from *setPoint*. 2) *WFIT*: is similar to WBEST, except that it does not stop when the expected TCR starts diverging, but also checks if by activating/deactivating some other threads the expected TCR is closer to *setPoint*. 3) *WOPTIMAL*: calculates the expected TCR that for all permutations of inactive (active) threads, then actives (deactivates) the set of thread for which the TCR is closest to *setPoint*. 4) *WCOMPLETE*: is similar to WOPTIMAL, except that it explores all possible permutations of all threads (both active and inactive).

*Experimental study*. The scheduler was implemented in DSTM2. In addition to benchmark applications used in the study on PoCC, two synthetic benchmarks, RBTree and SkipList were used. The experimental study was executed on top of a 2 x 8-core machine and varying the number of initial threads between 2 and 16. By results, the proposed techniques improved speed-up over 10% in the most of the experimental scenarios, and in some cases up to 30%. No one of the alternative policies performs definitively better than others.

***F2C2-STM***. The study in [9] describes Flux-Based Feedback-Driven Concurrency Control for STMs (F2C2-STM), that uses a control algorithm similar to the one based on the hill climbing search proposed by Chan et Al. 2011 [7]. However, F2C2-STM controls the number of active threads rather then the number of transactions allowed to concurrently run. Also, F2C2-STM introduces an initial phase (inspired to slow-start phase of TCP congestion control), where the application starts with 2 active threads, that are exponentially incremented until the measured throughput starts decreasing. Then, the number of active threads is always incremented or decremented by 1.

*Experimental study*. F2C2-STM was implemented on TinySTM. It was evaluated with STAMP on a 4 x 8-cores machine. With the experimental setting used in this study, Labyrinth, Genome and Ssca2 showed to be fully scalable (i.e. the speed-up increases up to 32 concurrent threads). With these applications, the performance with F2C2-STM is on average slightly penalized. Intruder, Yada, Kmeans and Vacation showed limited scalability (i.e. the speed-up starts decreasing with less then 32 concurrent threads). In this case, F2C2-STM clearly performs better, avoiding performance loss due to high concurrency while incrementing the number of concurrent threads. In these scenarios, F2C2-STM improved performance with respect to the best static configuration of the baseline by up to 10%.

**Discussion** As evidenced by our presentation, the proposed feedback-driven techniques use quite simple strategies, where decisions depend on a single input feature (the feedback parameter) and/or some (tunable) thresholds. Some of them differ only in a few details (e.g. Chan et Al. 2011 vs F2C2-STM). Nevertheless, we note that the simplicity is an advantage of these techniques, since this reduces the intrusiveness of the scheduler (e.g., few scheduler metadata are required) and allows to keep the run-time overhead low. This is favourable especially with low contention. Indeed, in such a case, a scheduler can do little to improve the performance. Rather, it should not cause slowdown due to its overhead.

One point to note is that in the case of techniques that use tunable thresholds (e.g. ATS and POCC) the effectiveness of the scheduler may depend on the goodness of the selected thresholds' values, which in turn depends on the workload profile of the application. As an example, a given TCR target range may be suitable for some workload profiles, while it may be sub-optimal in other cases. Accordingly, with these techniques, the user should to be able to select the proper configuration.

Some feedback-driven techniques rely on exploration-based approaches, such as the hill climbing search in F2C2-STM and the *Probe* policy in Chan et Al. 2011. One disadvantage of these approaches is that the scheduler has to execute

continuous exploration steps. Exploration is still required even after that the peak throughput has been found, since the workload profile may change along the application execution (e.g., see experimental data presented in [9]). The results is that the scheduler may keep a sub-optimal setting for a non-minimal time of the application execution due to the exploration. This disadvantage could be partially counteracted by dynamically regulating the duration of the exploration step after the peak throughput is found. In any case, this problem has not been addressed in any of the above-mentioned studies.

### 4.2 Prediction-Driven Techniques

***Shrink***. Shrink [10] uses a heuristic strategy, supported by a prediction scheme based on the *temporal locality* assumption (that we already mentioned in Section 3). The read-set of a transaction which is going to be executed by a thread is predicted on the basis of data read by the last $n$ executed transactions by the same thread, where $n$ represents the *locality window*. Specifically, if a data item has been read by the previous *i-th* transaction executed by the thread, a parameter named *confidence* is incremented by a constant $c_i$ (*confidence constant*). When *confidence* overcomes a threshold (*confidence threshold*), the data item is added to the predicted read-set. Write-set is predicted only for re-starting transactions, and it is assumed to include data belonging to write-set of the last aborted run of the transaction. Upon (re-)starting, transactions are serialized if the predicted data sets show that a potential conflict may occur with some concurrent transaction. To reduce the scheduler overhead with low contention, Shrink comes in actions for a thread only when its transaction success ratio is below a threshold (*success threshold*). Further, Shrink takes advantage of a heuristic called *serialization affinity*, according to which the probability of serializing a transaction should be proportional to the amount of contention. Thus, it activates the scheduling policy with a probability proportional to the number of transactions waiting to be serially executed.
*Experimental study*. Shrink was integrated in TinySTM and SwissTM [38] and was evaluated using STAMP, STMBench7 and RBTree on a 4 x dual-core machine, running applications with up to 24 concurrent threads. The following setting was used: $n = 4$, $c_1 = 3$, $c_2 = 2$, $c_3 = 1$, *confidence threshold = 3* and *success threshold = 0.5*. The temporal locality assumption was validated through a set of experiments executed with STAMP and STMBench7. Results show an average prediction accuracy of 70% for STAMP applications. For STMBench7, read-set prediction accuracy varies between 30% (for write dominated workloads) and 80% (for read dominated ones), and it is around 70% for write-set. As regards performance, results show that, generally, applications do not take advantage of Shrink when running with a number of threads less or equal to the number of cores. On the other hand, with more threads, Shrink preforms better. In the case of SwissTM with STMBench7 and RBTree, it improves performance in some scenarios. For TinySTM with STMBench7 and STAMP, Shrinks partially prevents the performance loss of the baseline, that rapidly drops down while incrementing the number of threads. In a few cases, Shrink outperforms the peak performance of the baseline. In the worst cases scenario for the baseline, where the throughput drops down close to zero, Shrink improves performance by up to 32 times.

***SCA***. Speculative Contention Avoidance (SCA) was proposed in [11]. The presented study defines *contention locality* the likelihood that an aborted transaction conflicts again during the subsequent run. SCA relies on a mechanism based on saturating counters, similarly to branch predictors used in pipelined microprocessor [39]. In SCA, the saturating counters are used to estimate the contention locality, thus for predicting whether a restarted transaction is likely to conflict again. Each thread has a saturating counter (SC) and a contention bit (CB). Two different prediction polices are proposed. With the first policy (*reset SC*), when a transaction aborts (commits), CB is set to 1 (0) and SC is incremented by one (is set to zero). With the second policy (*decrement SC*), SC is decremented by one when a transaction aborts. A transaction is serialized upon starting if CB is equal to 1 and SC is above a given threshold, that means that the transaction is predicted to conflict.
*Experimental study*. SCA was incorporated in TL2 [22] and was evaluated with Bayes, Kmeans, Labyrinth, Ssca2 and Vacation. Experiments were executed only on top of a simulated server with 2 x 8-core, running up 64 concurrent threads. Six different configurations of SCA were used, varying SC from 0 to 2, with both *reset SC* and *decrement SC* policy. Threshold was set equal to 1. Results on prediction accuracy over all experiments show that it varies from 25% to 99%. As for performance results, only aggregate results are presented, thus they not allow to verify the effectiveness of the scheduler depending on the number of concurrent threads. They show that SCA reduces the application execution time with respect to the baseline from 37% (for Bayes) to 82% (for Kmeans). *Decrement SC* policy generally performs better than *reset SC*. However, the presents results

***Heber et Al***. In [40], the authors investigate how serialization influences performance of CMs in STMs. The presented study analyses various optimization mechanisms to reduce the phenomenon of *mode oscillation*, which may lead to performance degradation due to continuous activations a deactivations of transaction serialization. Also, the study introduces a scheduling technique that serializes the execution of a transaction after that it has been aborted $k$ consecutive times. Essentially, the rationale behind this technique is that if a transaction is observed to conflict $k$ consecutive times then the probability to conflict again is predicted to be high. Accordingly, it could be more convenient to execute the transaction in isolation.
*Experimental study*. Since the study focuses on differentiated aspects that can affect the performance of STMs, only a few experimental data related to the proposed scheduling technique are shown. The scheduler was implemented in RSTM and was evaluated with CBench [40], RandomGraph and Swarm [41], running up to 32 concurrent threads on a 8 x 4-core machine. Only results for the cases of $k = 1$ and $k = 100$ are presented. In both cases, the proposed scheduler performs better than the baseline with all benchmarks. Further, results show that, on average, the scheduler achieves better results with $k = 1$ rather than with $k = 100$.
**Discussion** Basically, prediction-driven techniques try to exploit the knowledge gained by observing data access

patterns or conflict patterns of transactions. They use more complex strategies with respect to feedback-driven techniques. Anyway, the assumptions behind the proposed prediction-driven techniques are based on empirical observations, and these may not be valid to the same extent for any application. Indeed, the experimental results with both Shrink and SCA show that the prediction accuracy significantly changes across different applications. One factor that can negatively affect the prediction accuracy of strategies based on the observation of the recent transaction data access patterns is the frequency of write operations. In fact, write operations often modify data structures traversed by transactions. This may lead to variations of the access patterns of transactions which access the modified data structures right after. This phenomenon is also evidenced in [10]. In conclusion, the proposed prediction-driven techniques work well in the case of applications with read-dominated workloads and/or with workload profiles that change relatively slowly over time. Finally, we note that also all the proposed prediction-driven techniques use tunable parameters that affect the decision of the scheduler, and that have to be configured by the user (e.g. Shrink uses six different parameters).

### 4.3 Reactive Techniques

***CAR-STM***. CAR-STM [12] is a transaction scheduling technique that offers two policies. The first one, called *Basic Serializing Contention Management* (BSCM), aims at reducing the probability that two previously conflicted transactions conflict again. The second one, called *Permanent Serializing Contention Management* (PSCM), ensures that two conflicting transactions never conflict again. With PSCM, if a transaction $a$ conflicts with a transaction $b$, and $a$ has started after $b$, $a$ gets aborted. Then $a$ gets placed in a queue of transactions that will be executed by the same thread that was executing $b$. This reduces the probability that $a$ conflicts again with $b$. We note that a conflict between $a$ and $b$ is still possible. Indeed, if $b$ conflicts with a third transaction, then $b$ will be placed in a queue of transaction that will be executed by another thread, hence it may conflict again with $a$. To prevent such a situation, the second policy, PSCM, marks $a$ as a *subordinate transaction* of $b$. In this case, if $b$ is placed to another queue after a conflict with a third transaction, also all $b$'s subordinate transactions are moved to the same queue. CAR-STM also provides the option to schedule transactions in a proactive way (i.e. without waiting for the first conflict) by allowing the application to provide the scheduler with information about conflict probability between transaction pairs. However, this shifts to programmers the burden of developing applications aware of transaction access patterns, thus its not a real advantage offered by the scheduling technique itself.

*Experimental study*. CAR-STM was incorporated in RSTM and was evaluated with STMBench7 on a 4 x 8-core machine, running up to 32 concurrent threads. Experimental results are presented only for 2 execution scenarios of STMBench7 (read/write workload and write-dominated workload). With these workload configurations, the baseline shows poor scalability. CAR-STM shows, with both policies, better performance than the baseline. BSCM policy performs better. On average, it provides 4x throughput improvement. CAR-STM also improves the system stability by reducing standard deviation of both throughput and application execution time.

***Steal-on-Abort***. The idea of moving aborted transactions to the queues of transactions to be processed by other threads is also used in *Steal-on-Abort* [13]. This technique uses two queues for each thread. The first one maintains the new transactions to be processed by the thread. The second queue is called *steal* queue. When a transaction $a$, processed by a thread $T$, gets aborted due to a conflict with a transaction $b$ processed by a thread $T'$, $T'$ "steals" transaction $a$ from $T$ and puts $a$ in its own steal queue. After $T'$ completes the execution of $b$, transactions in the steal queue of $T'$ are moved to its own main queue. Steal-on-Abort offers two strategies to move these transactions. With *Steal-Tail* strategy, transactions are inserted at the tail of the main queue. With *Steal-Head* strategy, they are inserted at the head.

*Experimental study*. Steal-on-Abort was implemented in DSTM2. It was evaluated with LinkedList, RBTree and Vacation, running up to 8 concurrent threads on top of a 4 x dual-core machine. Experiments were executed for both strategies and with tree CMs, i.e. Aggressive, Polka and Priority. By results across all experiments, Aggressive takes advantage of steal-on-abort more than the other CMs, showing a performance improvement from a minimum of 16x. With polka, performance improvement is lower. With Priority, there are no benefits with any policy, and the performance is penalized in some cases.

***RelSTM***. RelSTM [14] is a transaction scheduling technique which uses the *second-hop conflict* relation. Namely, if a transaction $b$ conflicts with a transaction $a$, and another transaction $c$ conflicts with $b$, then $c$ becomes a *second-hop transaction* of $a$. Each transaction is marked with a unique Id. Upon a conflict, a transaction stores the Id of the opponent transaction and all Ids of transactions the opponent conflicted with (second-hop transactions). After an abort, a transaction is serialized if the previous conflicting transaction is still running, otherwise it stalls while the percentage of second-hop transactions which are still running is over a given threshold.

*Experimental study*. RelSTM was implemented in TinySTM, and was evaluated with all applications of STAMP running with up to 64 concurrent threads on a 4 x 16-core machine. The percentage threshold of second-hop transactions was set equal to 30%. Results achieved by running up to 4 threads show that RelSTM improves performance with respect to the baseline up to 1.42 for Bayes, and up to 1.7 for Kmeans. For Intruder, Labyrinth, Yada and Scca2, on average, the performance is penalized. For the other applications, the performance is quite similar to the baseline. The improvement is more evident in scenarios with a high number of threads, particularly for 32 and 64 threads, where all applications, except Labyrinth, Scca2 and Kmeans, show reduced execution time with respect to the baseline. For Labyrinth, performance improves only in some scenarios, while Scca2 and Kmeans show performance loss in all scenarios.

**Discussion**. The peculiarity of reactive techniques is that they act in response to conflict events, and provide the

advantage of reducing the number of repeated conflicts. On the other hand, the approaches adopted by these techniques in some cases are too pessimistic. For example, assume that a transaction $a$ is aborted due to a conflict with another transaction entered in the commit phase, or due to a read operation accessing data modified by a concurrent transaction that has already committed. In these cases, if $a$ is immediately restarted, the probability that the other transaction is still running is low or null. Accordingly, it would be more convenient to quickly restart transaction $a$ along the same thread rather than to pay the processing cost of moving $a$ to the queue of another thread. Further, we note that if an aborted transaction is immediately restarted along the same thread, the transaction likely will be executed by the same CPU-core (unless the thread is de-scheduled by the operating system and/or moved to another CPU-core). Thus, the transaction could take advantage of very reduced latency to access data that were accessed along its previous run and are still (valid) in the CPU-core private level cache. Conversely, if the transaction is re-executed by another CPU-core, these data likely will be not found in the cache. This kind of phenomenon was studied also in the context of database transactions [42]. Therefore, since the difference of latency to access data in private level cache vs. other memory levels is of one or two orders of magnitude, moving the transaction to another thread could penalize the performance in some cases. Definitely, the approaches used by the proposed reactive techniques are generally more fruitful in the case of high contention, where transactions are subject to be aborted a number of times. In such a scenario, these techniques can counteract the performance loss by effectively reducing the transaction abort rate.

### 4.4 Mixed Heuristic-Based Techniques

***LUTS***. The technique presented in [19], called Lightweight User-level Transaction Scheduler (LUTS), dynamically changes the scheduling strategy depending on the transaction length. LUTS uses a specific pool of threads in charge of processing transactions, whose size is equal to the number of available cores in the system. Transactions are grouped by Id. All transactions generated by the same application code block have the same Id. The strategy used by the scheduler depends on the average execution time of the last hundred committed transactions. If it is below a preconfigured time threshold *tc*, LUTS uses the same strategy of ATS based on contention intensity (CI). The difference is that LUTS keeps values of CI per transaction Id, rather than per thread. When the average duration of transactions is above the threshold, LUTS uses a more complex heuristic. It uses data structures for keeping information about conflict probabilites for each transaction Id as a function of the set of active transactions in the system. When a thread in the pool becomes available to process a new transaction, the scheduler reads Ids of all running transactions, and extracts from data structures information to select, among the ready-to-run transactions, the one with the lowest conflict probability. Upon a transaction aborts (commits), data structures are updated by decreasing (increasing) the conflict probability of that transaction Id by a given probability constant *pc*.
*Experimental study*. LUTS was integrated in TinySTM. It was evaluated with STAMP and STMBench7, running up to 128 concurrent threads on a 4 x 10-core machine. The used configuration is: CI threshold = 0.5, $\alpha = 0.75$, *pc* = 0.1, *tc* = 100k cycles. Concerning results with STAMP, in underloaded scenarios (up to 32 threads) LUTS does not improve the performance with respect to the baseline, except with 16 and 32 threads with Genome and Intruder. With Scca2, the performance degrades. When running more the 32 threads, LUTS always performs better than the baseline, except with Scca2. This may be due to the profile of transactions in Scca2, given that they are very short and very similar, thus not allowing to benefit from strategy switching that never occurs. With STMBench7, the performance improvement is more evident. In effect, the transaction profiles of STMBench7 are very heterogeneous. LUTS achieves up to 10x speed-up. In the worst case, the performance is equal to the baseline.

***ProVIT***. ProVIT [20] is another transaction scheduler that uses two different strategies for short and long transactions. However, differently from LUTS, ProVIT uses the two strategies at the same time. Transactions with a given Id are classified as short (long) transactions if the average red-set size is below (above) a predefined threshold (*VIT-threshold)*. ProVIT regulates the number of short transactions with the same Id that can concurrently run on the basis of the wasted work ($ww_{Id}$). When a transaction with a given Id commits, the wasted work is updated using the formula: $ww_{Id} = k \cdot ww_{Id,prev} + (1 - k) \cdot numRestarts$, where $ww_{Id,prev}$ is the previous value of $ww_{Id}$, $numRestarts$ is the number of times a transaction is aborted before to commit, and $k$ is constant between 0 and 1. If $ww_{Id}$ is below a given threshold (*RL-threshold*), only one transaction with that Id is allowed to run. Otherwise, the number of allowed transactions is increased proportionally to the square of the difference between $ww_{Id}$ and *RL-threshold*. As regards long transactions, ProVIT uses a fine-grained strategy for preventing them from multiple aborts. ProVIT relies on the assumption that, when a transaction aborts, data read by the transaction represent a reliable prediction of the read-set of the subsequent transaction re-execution. Hence, if aborted, a long transaction becomes a *Very Important Transaction* (VIT) and makes public its predicted read-set to other transactions. A non-VIT transaction, upon committing, is allowed to proceed only when its write-set does not intersect the predicted read-set of any concurrent VIT transactions. When such an intersection occurs between two concurrent VIT transactions, the older one is allowed to commits as a first.
*Experimental study*. ProVIT was integrated in FlashbackSTM [43], a word-based multi-version STM implemented in Java. The experimental study was executed on top of a 4 x 12-core machine, using STMBench7 and six STAMP applications (Genome, Intruder, KMeans, Labyrinth, SSCA2, and Vacation) and running up to 48 concurrent threads. The following configuration was used: $k = 0.1$, *RL-threshold=4*, *VIT-threshold=350*. With STMBench7, in two out of three presented scenarios, ProVIT performs better than the baseline with more than 4-8 threads, and keeps the throughput close to (and in a few cases slightly higher than) the peak throughput achieved with the baseline. In all other cases, the performance is comparable to the baseline. With all STAMP applications, ProVIT generally overcomes the performance of the baseline with more than 8-12 threads, except for Labyrinth, for which the performance is similar.

The baseline performs better than ProVIT only for Scca2 up to 4 threads.

**Discussion**. Substantially, both the above-mentioned mixed techniques are based on the idea of exploiting a lightweight approach for short transactions, and a more complex one, albeit more expensive, for long transactions. Both techniques use a lightweight feedback-driven strategy in the first case, and a prediction-based strategy in the second case. The rationale is that with short transactions the overhead introduced by a complex strategy could be excessive. Rather, it could be more convenient to use a less accurate strategy, but less expensive. In general, the goal of mixed techniques is to take advantage of different strategies that fit different workload profiles. Hence, a key aspect is represented by the approach to decide at run-time when using one or another strategy. Indeed, an improper selection of the strategy may be counterproductive, leading even to drastic loss of performance. Both in LUTS and in ProVIT, the strategy is selected on the basis of preconfigured thresholds that are used to differentiate short and long transactions. Unfortunately, although the configuration of these thresholds plays a key role for these techniques, an effective approach to select the most suitable values has not been discussed in any of the two studies. The values used in the experiments of the two studies were selected by preventively trying different values, and the ones that produced the best results for the majority of the workloads have been chosen. Further, data about sensitivity analysis have not been presented. Ultimately, the effectiveness of these mixed techniques still depends on the ability of the user to properly configure the strategy selection mechanisms.

### 4.5 Machine Learning-based Techniques

***SAC-STM***. Self-Adjusting Concurrency STM (SAC-STM) [15] is a thread scheduling technique that uses Artificial Neural Network (ANN)-based performance models. An ANN is a Machine Learning (ML) model [33] that can be instantiated via a *training process* and allows to approximate a function. The training process takes as input a data set composed of samples which (partially) represent relations between input and output of the function. SAC-STM uses an ANN for estimating the average *wasted time* of transactions (i.e. the average execution time of aborted runs of transactions) as a function of the application workload profile and the number of concurrent threads. Although the training process uses only a few samples to build the function, interpolation/extrapolation abilities of the ANN are expected to allow to calculate the function for whichever number of threads and workload profile. At run-time, the scheduler executes a continuous loop to regulate the number of active threads. For each step in the loop, it executes the following actions: 1) Collects workload samples over the last $n$ executed transactions for calculating the average values of the input features of the ANN. 2) Uses the ANN for estimating the expected transaction wasted times in variation of $k$, for each $k \leq M$, where $k$ is the ANN input feature representing the number of concurrent threads, and $M$ is the maximum number of threads. 3) Calculates, based on the expected transaction wasted times, the expected application throughput in variation of $k$. 4) Keeps active $k'$ threads, where $k'$ is the number of threads for which the throughput is estimated to be the highest one.
*Experimental study*. SAC-STM was integrated within TinySTM. Experiments were executed with Genome, Intruder, Kmeans on top of 2 x 8core machine, running up 16 concurrent threads. For the training process, 800 samples have been used, which were randomly selected along 64 runs of an application. Each sample includes averaged values over 2000 consecutive transactions. By results, in low contention scenarios (i.e between 2 and $4/6$ threads where the throughput with the baseline grows while increasing the number of concurrent threads), the overhead introduced by the scheduling mechanisms sometimes penalizes performance of SAC-STM, with a slowdown between 0% and 22%. On the other hand, in high contention scenarios (where the performance with the baseline rapidly drops down while increasing the number of threads) SAC-STM keeps performance close to the best value achieved with the baseline, independently of the number of threads. Also, in a few cases, SAC-STM provides higher performance than the peak performance of the baseline.

***DSF-STM***. SAC-STM has been improved in [16] with a technique, called *Dynamic Feature Selection* (DSF)-STM, that dynamically resizes the set of input features of the ANN. DSF-STM aims at reducing the overhead for run-time feature sampling, as well as the complexity of the function to be approximated by the ANN. The proposed technique discards an input feature if, along the application execution, its variance goes below a given threshold. Further, for each couple of features, DSF-STM discards one of them if their correlation exceeds a given threshold. In the context of ML, this approach is known as Correlation-based Feature Selection [44]. After having discarded some feature, DSF-STM changes the current ANN with another that has been previously trained for the specific reduced set of features. Discarded features are added again to the set if DSF-STM detects that the ANN prediction error (i.e. the difference between the real throughput and the throughput estimated via the ANN) overcomes another threshold. *Experimental study*. DSF-STM was evaluated in the same experimental setting of SAC-STM, using Intruder, Ssca2 and Vacation. Further, the authors used a modified version of Vacation, where the percentage of different transaction types dynamically changes over time. Results show that DSF-STM effectively reduces the scheduling overhead with respect to SAC-STM, providing performance improvement independently of the number of concurrent threads. The improvement varies between 15% and 45%, depending on the application.

**Discussion**. The key aspect of the model-based techniques, thus including ML-based techniques, is that the scheduler makes decisions on the basis of estimations provided by a performance model of the application. Compared to techniques in which decisions are made on the basis of (generic) heuristics, the advantage of the model-based techniques is that the model can embed more specific knowledge about the application/system. In fact, the prevailing factor influencing the effectiveness of a model-based technique is the reliability of the model. To improve the model estimation accuracy, the proposed ML-based techniques use a specific model instance for each application. With this approach, it is possible to instantiate reliable models for applications with

very different workload profiles. In fact, the experimental results of SAC-STM are good in high contention scenarios with all the applications. The drawback of these techniques is that they represent a viable solutions providing that it is possible to perform a specific training processes for any application. This process may require a certain amount of time, as well as may require specific user skills.

### 4.6 Analytical Model-based Techniques

***CSR-STM***. Concurrency Self-Regulating STM (CSR-STM) [17] is an analytical model-based thread scheduling technique. The basic scheduling scheme is similar to the one used in SAC-STM, where the number of active threads $k$ is regulated on the basis of estimations of the application throughput in variation of the $k$. However, rather than ANNs, CSR-STM uses an analytical performance model for estimating the transaction abort probability as a function of the same set of input features of SAC-STM. The analytical model is parametric, allowing it to be customized for a specific application and a given machine. Indeed, the model parameters also enable to capture the effects due to different hardware configurations. The values of these parameters are estimated via regression analysis, using data collected during an a priori sampling phase of the application.
*Experimental study*. CSR-STM was incorporated in TinySTM. Kmeans, Yada and Vacation were used for the experimental evaluation study, which was executed on a 2 x 8 core machine, running up 16 concurrent threads. Authors presents a preliminary evaluation study for assessing the model accuracy depending on the amount of samples collected during the a priori profiling phase of the application. Results shows that with 80 random samples, totally collected with 2 and 4 concurrent threads, the model calculates the abort probability spanning from 2 to 16 concurrent threads with an error between 2.1% and 18.9%. When samples are collected with 2, 4, 8 and 16 threads, the error is bounded to 2.6%. As for performance assessment, CSR-STM notably provides better performance results that the baseline in high contention scenarios. However, with respect to the peak throughput of the baseline, sometimes there is a slightly degradation while increasing the number of threads. In low contention scenarios, the overhead introduced by the scheduler slightly penalizes the performance of CSR-STM with respect to the baseline.

***MCATS***. Markov Chain Based Transaction Scheduling (MCATS) uses a performance model based on a Markov Chain [45]. The scheduler controls the maximum number $m$ of transactions allowed to concurrently run. The Markov Chain captures the evolution of the system state, which is represented by the number of threads concurrently executing transactions (including both running and blocked transactions). A state transition in the Markov Chain occurs upon a thread commits a transaction or starts a new one. The model does not require an a priory sampling phase of the application. Indeed, at run-time, the scheduler runs a continuous loop, where, for each step, it (re-)instantiates the model on the fly by calculating transition rates of the Markov Chain. These rates are derived by estimating four parameters, which represent the input features of the model, including (i) the average execution time of committed transactions, (i) the average wasted time of transactions, (iii) the transaction conflict ratio and (iv) the average execution time of non transactional code blocks. Then, at the end of each step, the scheduler calculates by the model the expected system throughput as a function of $m$, and select the values of $m$ that maximizes the throughput.
*Experimental study*. MCATS was incorporated in TinySTM and was evaluated on top of a 16-core machine. The study on the model prediction accuracy shows that the throughput prediction error with Intruder, Yada and Vacation is below 10%. The results achieved with the same applications show that in low contention scenarios, performance with MCATS is slightly penalized due to scheduling overhead. On the other hand, in high contention scenarios, this technique works effectively, being able to preserve in many cases a performance level close to the peak performance of the baseline.

**Discussion**. Similarly to ML-based techniques, also analytical model-based techniques take advantage of performance models of the applications. Nevertheless, the analytical models used by the proposed techniques require less onerous workload sampling phases compared to the ML-based ones. This may represent a significant benefit, since it may allow to instantiate the model on the fly while the application runs (as for MCATS), thus avoiding (costly) training processes to be performed in advanced.

However, by the analysis of the experimental results, the techniques that use ML models seem to behave, on average, better than the analytical model-based counterparts. Thus, the currently available model-based techniques offer a trade-off between affordability of the model instantiation process and model efficiency. We will discuss in more detail pros and cons of the ML-based and the analytical approach in the next section, after the presentation of a mixed model-based technique.

### 4.7 Mixed Model-based Techniques

***AML***. Analytical/Machine Learning (AML) Scheduling Technique was presented in [35]. It is a mixed model-based technique that uses the analytical performance model of CSR-STM and the ML model of SAC-STM. AML Scheduling Technique tries to take the best of the two models, i.e. the lightweight workload profiling phase of the analytical model, and the high performance estimation accuracy of the ANNs. Thus, to reduce the duration of the a priori profiling phase, the scheduler initially exploits the analytical model, since it requires to collect smaller sets of samples. Also, the analytical model is used to generate *virtual samples* to cover a larger subset of the input feature domain. Alongside, additional samples are collected during the application execution. Then, real and virtual samples are joined within a single extended data set, which is used for training the ANN. After this step, the scheduler uses the ANN in place of the analytical model. In addition to reduce the duration of the initial profiling phase, this mixed approach also allows to speed-up the instantiation of the ANN, that provides more reliable performance estimations after being trained with the extended set of samples.
*Experimental study*. AML Scheduling Technique was evaluated in the same experimental context of SAC-STM, using Intruder, Kmeans, Yada and Vacation. Results about the

evaluation of the prediction error demonstrate that, when using a few samples, the analytical model is more accurate than the ML-model. This trend reverses while increasing the number of samples. In any case, the mixed AML-based technique always provides smaller prediction error. Regarding performance, results show that the AML-based technique always ensures the peak throughout of the best of the two models, and, generally, the time to reaches the peak performance is notably reduced.

**Discussion**. AML Scheduling Technique is the only technique relying on a mixed model-based approach. The rationale behind this technique is tied to pros and cons of the ML models against the analytical approach. ML models generally require a lot more samples than analytical models to be instantiated, and can provide good prediction accuracy within the sub-domain of sample values in the training set [46]. Conversely, analytical models require a reduced number of samples, and can provide reliable results also outside of the sub-domain of the sample values. The drawback is that analytical models typically rely on approximations or assumptions aimed at simplifying the model development (or at making it feasible). Thus, the models may become unreliable in those scenarios where the approximations or the assumptions are not sufficiently adequate. Another point to note is that, in AML Scheduling Technique, the use of the analytical model is twofold. It is used as a predictor and for speeding up the instantiation of the ML model. Overall, AML Scheduling Technique proves the potential advantages of a mixed model-based approach. However, this kind of approach is still poorly explored.

## 5 Overview of Previous Performance Comparison Studies

In this section we focus on results of comparison studies carried out by authors of some scheduling techniques we presented in Section 4. For each technique, we report in Table 1 the list of the other techniques (if any) it was compared with.

Feedback-driven techniques that use TCR as a feedback parameter have been compared in [6], [8]. The reported experimental results (which compare PoCC, the four control policies described in [5] and the four policies of Waisted Adaptive Concurrency Control described in [8]) demonstrate that there is not a policy that definitively improves the performance more than the other ones for all applications. Basically, as we pointed out in Section 4.1, the advantage of PoCC compared to other policies is mainly the reduced variance of the application execution time. This suggests that the dynamics of TCR variations change remarkably depending on the application, and it is complex to optimize the behaviour of a scheduler through a single TCR-based heuristic policy for whichever application.

In [10], the authors present a few comparison results of Shrink with ATS. Data show that noteworthy differences in performance exist only in the case of read-dominated workload as long as the number of threads overcomes the number of cores. In these scenarios, Shrink performs better than ATS. The feedback-driven scheduler proposed by Chan et Al. is compared with Shrink and ATS in [7]. By results with all STAMP applications, no one wins in all cases. Further, also in the presented experimental scenarios, all techniques improve performance only when the number of threads overcome the number of cores.

Some interesting results come from the experimental study in [9]. In this study, experiments are executed on a machine with more cores (i.e. 32 cores) with respect to previous studies. Authors compare F2C2-STM with ATS and Shrink. Results demonstrate that, with limited-scalability applications, ATS and Shrink can improve performance also in some scenarios where the number of threads is less then the number of cores. On the other hand, in some cases (e.g. with Kmeans), they never perform better than the baseline. Ultimately, with limited-scalability application, the feedback-driven technique F2C2-STM performs generally better than ATS and Shrink.

LUTS has been compared with ATS and Shrink in [19]. Again, experiments executed with all STAMP applications show that no one wins in all cases. The mixed heuristic-based technique used by LUTS, which is optimized for short/long transaction-mixed workloads, does not achieve better results than the other techniques. The authors ascribe this result to the low transaction diversity of transaction profiles in STAMP applications. Conversely, with STMBench7, which is characterized by highly differentiated transaction profiles, experiments with various data structure sizes (small, medium, big, and huge) show that LUTS outperforms ATS and Shrink, ensuring more than 2x throughput improvement in the most of cases. However, results related to Shrink appear partially in contrast with the ones presented in [10] for scenarios of STMBench7. For example, in [10] Shrink never provides lower performance than the baseline with 8 concurrent threads. Conversely, this does not hold true in [19], which shows that Shrink provides lower performance in all cases with 8 threads. We observe that, in both studies, authors use the same Shrink implementation on TinySTM and the same values of tuning parameters. Thus, these results suggest that the hardware architecture, which is different in the two studies (see Section 4 for details), can significantly impact on the effectiveness of a scheduling technique.

In [20], ProVIT has been compared with CAR, ATS and Shrink, using FlashbackSTM as a baseline. For execution scenarios of STMbench7 with long transactions (including read-dominated, read-write and write-dominated workloads), ProVIT outperforms other techniques, although it generates higher abort rate. This result reveals that the other schedulers are too pessimistic with these kinds of workloads, i.e. they serialize too many transactions that otherwise would not conflict. For scenarios with short transactions, ProVIT equates other techniques with read-dominated workloads, while outperforms them while increasing the percentage of write operations. Results of this study for ATS and Shrink for scenarios of STMbench7 with long transactions are in accordance with the study in [19], showing that both techniques, as well as CAR, perform worse than the baseline. On the other hand, with short transactions, they perform better than the baseline while increasing the contention level. In the case of STAMP applications, ProVIT still performs better than any other technique, except that for some configurations of Ssca2.

As concerns model-based techniques, we already dis-

TABLE 1
Summary table of strategies, features and evaluation studies of scheduling techniques

| Technique | Transaction/ Thread Scheduling | Strategy | Input features | Tuning parameters | Evaluation Studies | | |
|---|---|---|---|---|---|---|---|
| | | | | | Integrated within (baseline) | Evaluated with (benchmarks) | Compared with |
| AML | Thread scheduling | Mixed Model-based | - Execution times of transactions and non-transactional code blocks<br>- Transaction read-/write-set | - Sample interval | TinySTM | STAMP (Intruder, Kmeans, Yada, Vacation) | CSR-STM<br>SAC-STM |
| ATS | Transaction scheduling | Feedback-driven | - Contention Intensity | - Weight for calculating CI<br>- CI threshold | RSTM | RBThree, HashTable, LinkedList, RandomGraph, LFU-Cache | - |
| Ansari 2014 | Thread scheduling | Feedback-driven | - Transaction Commit Ratio (TCR) | - Target TCR value<br>- Initial sample interval | DSTM2 | Lee-TM, Genome, Kmeans, Vacations, StepChange, RBTree, SkipList | PoCC |
| Ansari et al. 2008 | Thread scheduling | Feedback-driven | - Transaction Commit Ratio (TCR) | - Target TCR range<br>- Sample interval | DSTM2 | Lee-TM | - |
| CAR-STM | Transaction scheduling | Reactive | - Transaction abort probabilities (optional, provided by user) | | RSTM | STMBench7 | - |
| Chan et Al 2011 | Transaction scheduling | Feedback-driven | - Transaction Commit Ratio or<br>- Transaction Commit Rate | - Sample interval<br>- Min. No. of transactions per sample interval | TinySTM | All STAMP applications | ATS<br>Shrink |
| CSR-STM | Thread scheduling | Analytical Model-based | - Execution times of transactions and non- transactional code blocks<br>- Transaction read-/write-set | - Sample interval | TinySTM | STAMP (Kmeans, Yada, Vacation) | - |
| DSF-STM | Thread scheduling | Machine Learning-based | - Execution times of transactions and non- transactional code blocks<br>- Transaction read-/write-set | - Variance threshold<br>- Correlation threshold<br>- Prediction error threshold | TinySTM | STAMP (Intruder, Ssca2, Vacation) | SAC-STM |
| F2C2-STM | Thread scheduling | Feedback-driven | - Transaction commit rate | - Sample interval | TinySTM | STAMP (all applications) | ATS<br>Shrink |
| Heber et Al. | Transaction scheduling | Prediction-driven | | | RSTM | CBench, RandomGraph, Swarm | CAR-STM |
| LUTS | Transaction scheduling | Mixed Heuristic-based | - Contention intensity (CI)<br>- Ids of conflicting transactions | - Weight for calculating CI<br>- CI threshold<br>- Conflict probability increment<br>- Short/long transactions threshold | TinySTM | STAMP (all applications), STMBench7 | ATS<br>Shrink |
| MCATS | Transaction scheduling | Analytical Model-based | - Execution times of transactions and non-transactional code blocks | - Sample interval | TinySTM | | ATS<br>Shrink |
| PoCC | Transaction scheduling | Feedback-driven | - Transaction Commit Ratio (TCR) | - Target TCR value<br>- Initial sample interval<br>- Min. No. of transactions per sample interval | DSTM2 | Lee-TM, STAMP (Genome, Kmeans, Vacations), StepChange | Four policies proposed by Ansari et Al. 2008 |
| ProVIT | Thransaction scheduling | Mixed Heuristic-based | - Transaction read/write-set<br>- Transaction wasted work (tcc) | - Weight for calculating tcc<br>- Restart limit (RL)-threshold<br>- VIT-threshold | FlashbackSTM | STAMP (Genome, Intruder, KMeans, Labyrinth, SSCA2, Vacation), STMBench7 | ATS<br>CAR<br>Shrink |
| RelSTM | Transaction scheduling | Reactive | | - Second-hop transactions threshold | TinySTM | STAMP (all applications) | - |
| SAC-STM | Thread scheduling | Machine Learning-based | - Execution times of transaction and non-transactional code blocks<br>- Transaction read/write-set | - Sample interval | TinySTM | STAMP (Genome, Intruder, Kmeans) | - |
| SCA | Transaction scheduling | Reactive | | - Serializing threshold | TL2 | STAMP (Bayes, Kmeans, Labyrinth, Ssca2, Vacation) | - |
| Shrink | Transaction scheduling | Prediction-driven | - Transaction Commit Ratio<br>- Transaction read-/wite-set | - Locality window size<br>- Confidence threshold<br>- Success threshold<br>- Confidence constants | SwissTM, TinySTM | STAMP (all applications), STMBench7, RBTree | ATS |
| Steal-on-Abort | Transaction scheduling | Reactive | | | DSTM2 | LinkedList, RBTree, Vacation | - |

cussed in Section 4 experimental results presented in [16] and [35] related to DFS-STM and AML, which were compared with CSR-STM and SAC-STM. In addition to these studies, MCATS was compared with ATS and Shrink in [18]. Results are related to three different configurations of three STAMP applications. With Vacation, all techniques scale well and show similar performance. With Intruder and Yada, MCATS significantly performs better than ATS and Shrink, which do not efficiently counteract performance loss in high contention scenarios.

In conclusion, we note that, although various comparison studies exist in literature, they are quite fragmented and are limited to few and restricted sets of scheduling techniques. Furthermore, criteria that led to the choice of the compared techniques are not always clearly explained. In the next section we present a new study that we conducted with a larger set of technique, which we compared under a common experiment setting. By complementing the existing results, our study allowed us to draw more general conclusions, particularly from the perspective of the alternative scheduling strategies.

## 6 Experimental Study

In our experimental study we selected six techniques, i.e. ATS, SHRINK, SAC-STM, MCATS (for these techniques we used the source code released by the authors) and other two techniques (for which we used our implementations, since the source code of the authors was not available), i.e. the technique presented by Chan et Al. based on *probe* policy [7], that we call PROBE-STM, and the technique presented by Heber et Al. [40], that we call K-ABORT. We selected this set of techniques in order to cover all basic types of scheduling strategies of our classification scheme. The set includes two feedback-driven techniques that we selected

in order to evaluate also the impact of the two different feedback parameters that are typically used in feedback-driven strategies. Specifically, PROBE-STM uses the transaction throughput, which is a direct measure of the application performance. Conversely, ATS uses the Contention Intensity (which is similar to the transaction commit ratio), which is a specific parameter of TM applications, but it is not a direct measure of the performance of the application. All schedulers of this study have been integrated in TinySTM (baseline), and the source code is publicly available[1].

In our study we focus on scenarios with "real" parallelism, i.e. where the number of concurrent threads does not exceed the number of cores. Indeed, in STMs, as shown by results of various performance studies we already discussed, having more concurrent threads than cores is generally unfavourable. In fact, this condition causes performance loss due to the to context-switching overhead. Particularly, when threads are suspended while executing transactions, the number of concurrent transactions, and consequently the probability of aborting transactions, may increase. Hence, it is generally more convenient preventing this condition [47].

### 6.1 Experimental settings

We executed experiments with all STAMP applications on a 16-core HP ProLiant server, equipped with 2 x 8-core 2GHz AMD Opteron 6128 processors, 64 GB of RAM and Linux OS (kernel version 2.7.32-5-amd64). For each application, we used three different configurations of input parameters, each one giving raise to a different application workload profile. Totally, we show results for 24 execution cases. As for the configuration of tuning parameters, in ATS we set $\alpha = 0.5$, and in PROBE-STM each sample interval terminates after the execution of 1000 consecutive committed transactions. Further, we set k equal to 2 in K-ABORT, given that this value provided, on average, the best results. For all the other tuning parameters, we used the same values of the experimental studies we presented in Section 4. In the case of Labyrinth, we reduced the length of the sample interval of MCATS and K-ABORT to terminate after the execution of 50 consecutive committed transactions, since the total number of executed transactions in Labyrinth is limited to some hundreds.

Each data point that we show in the results was calculated as the average over the execution of eight application runs. We observed that with eight runs the baseline provided stable results.

### 6.2 Analisys of results

All experimental results are reported in Figures 2 and 3. We show the speed-up of the application, i.e. the ratio between the application completion time when executed with a scheduler with x concurrent threads (reported on x-axis) and the application completion time when executed with the baseline with a single thread.

Firstly, we analyse results of heuristic-based techniques. In scenarios where the baseline scales quite well up to 16 threads (i.e. all configurations of Ssca2 and *configuration 2* of Vacation and Labyrinth Vacation) these techniques do not cause a relevant performance degradation, except ATS with Labyrinth. On average, ATS shows good performance with Genome and Vacations, even when the performance of the baseline quickly drops down due to high contention. In the other cases, ATS shows poor results with respect to both the baseline and the most of the other techniques. We remark that the feedback-driven strategy used by ATS serializes transactions when the Contention Intensity is above the CI-threshold, whose value is selected by the user (in our experiments, it was equal to 0.5). Essentially, the results show that the performance is very sensitive to this value while varying the workload profile. As a confirmation of this, we observed that, for some configurations of Intruder, the peak performance of the baseline is achieved when the transaction abort probability is about 0.7, while, for other configurations, when it is about 0.4. As such, the value of CI-threshold equals to 0.5 is too conservative in some cases, while it is too high in other cases. Ultimately, the strategy used by ATS requires the choose of the right value of CI-threshold for each specific workload profile.

The other feedback-driven technique of our study, PROBE-STM, on average performs better than ATS. It performs well with scalable workloads. Also, with some limited-scalability workloads (including all configurations of Genome and Yada, and *Configuration 1* and *3* of Vacation), PROBE-STM performs better than the baseline. In other cases, the performance results are similar to the baseline, excluding *Configuration 1* of Kmeans and *Configuration 2* of Labyrinth. Nevertheless, we observed that in scenarios where the transactions commit ratio becomes highly variable, which typically happens with high contention, PROBE-STM often loses ground to other techniques. Highly variable commit rate leads to continuous throughput fluctuations along the application execution. This kind of "noise" on the feedback parameter can reduce the effectiveness of the hill climbing search strategy used by PROBE-STM, since it can cause many (wrong) changes of searching direction. In this case, the problem could be mitigated by incrementing the length of the sample interval in a way to reduce the noise due to high commit rate variability. On the other hand, this may reduce the reactivity of the scheduler in the face of changes of workload profiles.

The predictive strategy of Shrink improves the performance only in some cases. Shrink performs well in scenarios with scalable workload (e.g. Ssca2) and in some scenarios with limited scalability (e.g. all configurations of Genome and *configurations 1* and *3* of Vacation). In some other cases it performs very poorly (e.g. see Intruder and Kmeans). Also, results show that, in general, the effectiveness of the scheduler decreases when the contention increases (e.g. see Yada with more than 8-10 concurrent threads). This suggests that the accuracy of predictions decrease when the thread parallelism increases. This would not be surprising, given that the transaction execution dynamics become more complex while increasing concurrency, and shows that a prediction-driven technique is likely to be penalized in high concurrency scenarios. In addition, we note that Shrink exposes various tuning parameters. Similarly to ATS, the different behaviour with different workload profiles sug-

1. All links to the source code of each scheduler implementation can be found at http://www.dis.uniroma1.it/~disanzo in Section "Software and Links"

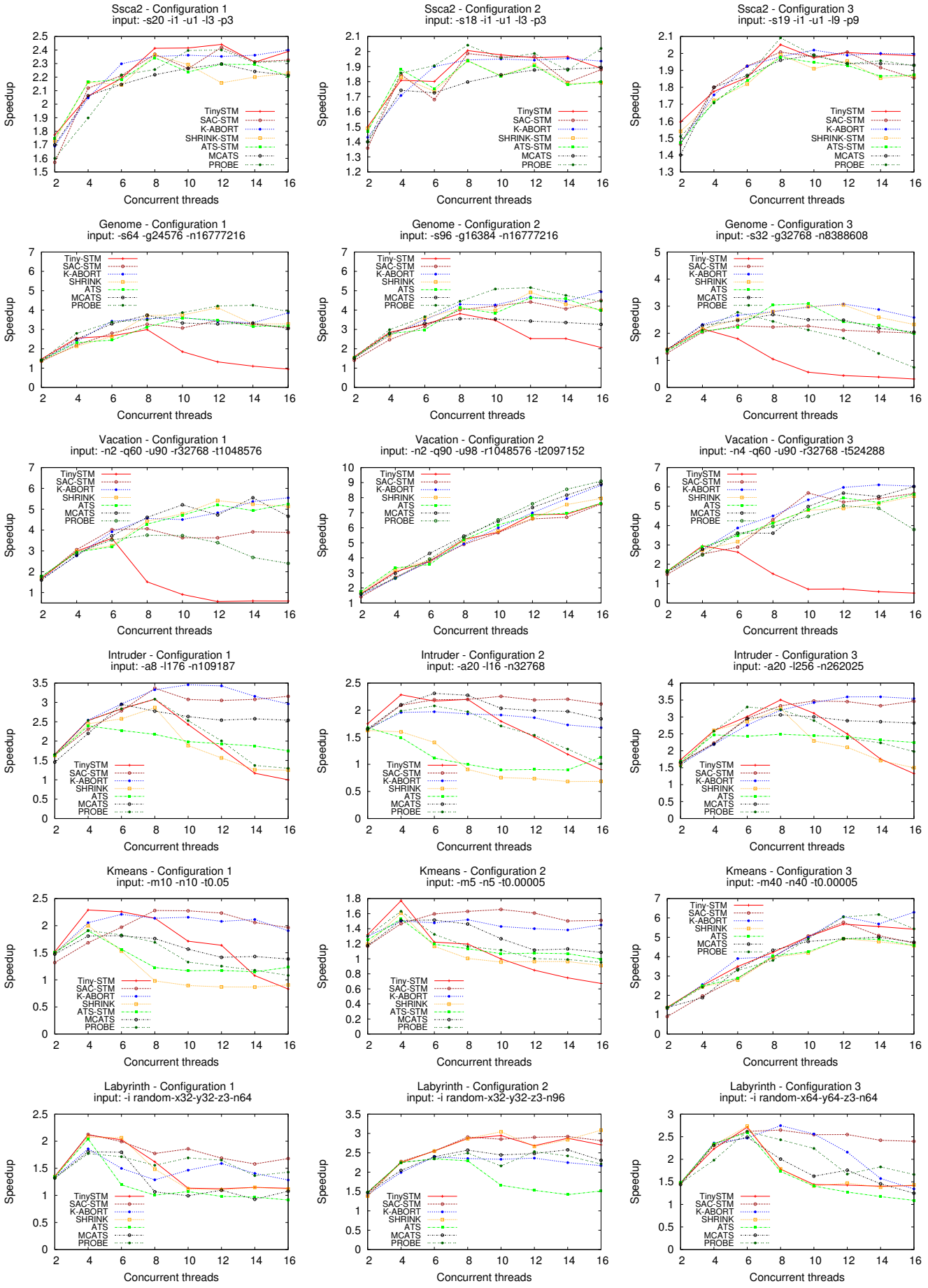


Fig. 2. Speedup for Ssca2, Genome, Vacation, Intruder Kmeans and Labyrinth

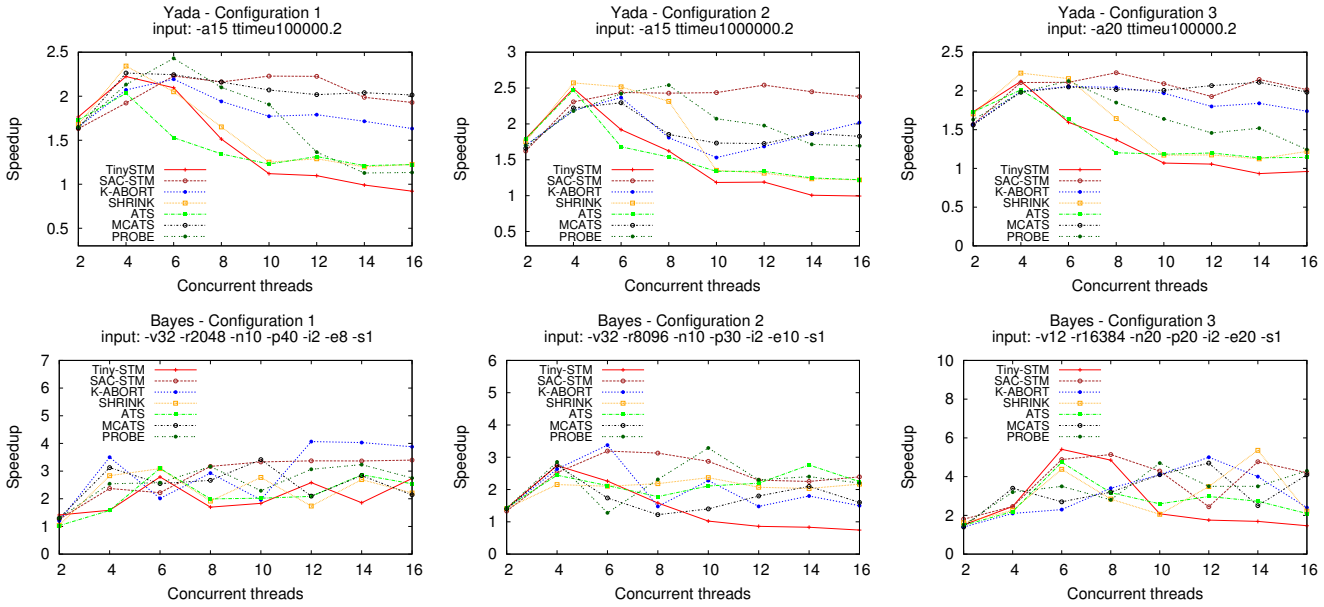


Fig. 3. Speedup for Yada and Bayes

gests that a given parameter configuration can be suitable in some cases, but not in other cases. Thus, also with Shrink, specific tuning would be required for each execution case.

K-ABORT, which uses a reactive strategy, performs well in scenarios with scalable workload and in some scenarios with limited scalability, where it shows good performance also with high contention. In effect, serializing aborted transactions, such as used by reactive strategies, brings benefit especially with high abort rate. K-ABORT performs quite well with all configurations of Ssca2, Genome and Vacation, and with *configuration 1* and *3* of Intruder and Kmeans. However, with Yada, Labyrinth and Bayes, there are other techniques that, on average, perform better than K-ABORT. We note that these three applications are characterized by the presence of long transactions and by high percentage of time spent while executing transactions [29]. With such a workload profile, serializing transactions lead threads to stall for a long time, and this extremely reduces the parallelism. In this case, the configuration with $k = 2$ is too conservative. However, we observed that a higher value of $k$ penalizes performance with other applications.

As regards model-based techniques, SAC-STM shows good results with scalable workloads, as well as in various scenarios with limited scalability. The most of times it ensures peak speed-up, also when the performance of the baseline rapidly drops down due to high contention. Compared to the other techniques, the strategy used by SAC-STM takes advantage of the a-priory training (as discussed in Section 4.5), which in advance provides the scheduler with knowledge about relations between the workload profile of a specific application and the parallelism level. Results confirm that SAC-STM can achieve good performance also in different execution scenarios. However, we note that SAC-STM is not very efficient with few concurrent threads. This is due to the implementation overhead of the scheduler, which has to collect a number of samples at run-time for calculating the values of the input features of the ML model. Indeed, in almost all scenarios with 2-4 concurrent threads, the speed-up with SAC-STM is (sometimes slightly) lower than the baseline.

The other model-based technique, MCATS, performs better than the baseline in various scenarios, particularly with high concurrency. With scalable workloads, the performance is penalized in a few cases (e.g. see Ssca2). With limited-scalability workloads, the most of times MCATS provides better results than ATS and SHRINK. Compared to PROBE and K-ABORT, the results change depending on the workload profile. However, MCATS is outperformed by SAC-STM in the most of scenarios. We remark that, differently from SAC-STM, MCATS does not rely on an a-priory training, but it exploits an analytical model that is cyclically (re-)instantiated on-the-fly at run-time. Compared to SAC-STM, MCATS requires a reduced set of input features and much less samples to instantiate the model. Thus, the model accuracy may be reduced and may be more affected by workload fluctuations. Indeed, in our experiments we observed that MCATS performs better when the workload profile changes slowly (e.g with Vacation). Finally, we note that MCATS performs poorly with Labyrinth, where we had to reduce the length of the sample interval. This confirms that fewer samples may be insufficient to capture the workload profile for instantiating the model. Thus, in the case of applications executing very few transactions, the model-based strategy of MCATS may be under-performing.

## 7 Conclusive Assessment and Open Challenges

Our literature analysis demonstrates that a variety of scheduling techniques for STMs exists. However, various comparison studies, including the one presented in this article, show that there is no a scheduling technique that definitively performs better than the other ones for whichever workload profile. We remark that the effectiveness over

a wide range of workload profiles and the ability to autonomously adapt to workload variations are important quality attributes for a scheduling technique. Anyway, experimental results reveal that the workload diversity represents a critical factor for the existing scheduling techniques.

As confirmed by our experiments, the existing heuristic-based techniques show a variegated behaviour in terms of performance with different applications, or when running the same application with different input configurations. Particularly, there is no a technique that demonstrated to be effective for a wide range workload profiles. To this purpose, tuning features offered by these techniques may be helpful. However, the selection of the proper configuration of the tuning parameters may be a non-trivial task, and may require the user to preventively execute a number of application runs to explore alternative parameter configurations. Nevertheless, a given optimal configuration may become sub-optimal if the workload profile changes. To cope with this problem, a try to develop a self-tuning mechanism was carried out in [4] for ATS. However, the proposed solution partially solves the problem, since it introduces another parameter that requires to be manually tuned.

Mixed heuristic strategies can help to cope with the problem of workload diversity, in particular when an application includes transactions with different profiles. However, all the proposed mixed heuristic-based techniques use tunable thresholds to discriminate the transaction profiles to be processed with different heuristics. Consequently, also these techniques may require the user to preventively explore alternative settings to find the best configuration of these thresholds (e.g. see the discussion on experimental results in [19], [20]). Nevertheless, once found an optimal value of a threshold, it is not certain that it is suitable for whichever transaction mix and/or application execution phase. Thus, heuristic-based techniques still require additional effort from the perspective of self-adaptivity.

Model-based techniques exhibit a more homogeneous behaviour and increased ability to adapt to different execution scenarios. However, we remark that the technique that achieves the best results, i.e. SAC-STM, uses a customized model for each application, and which requires to be instantiated in advance via a dedicated ML training process. Further, SAC-STM needs to collect samples of various input features at run-time, and this generates non-negligible overhead with low contention. Differently, with MCATS, which uses a lightweight and application-independent model that can be instantiated on-the-fly, the performance results become more variegated. This shows that model-based techniques should be further improved to optimize the trade-off between model instantiation overhead and their effectiveness over different workload profiles.

In conclusion, given that the existing scheduling technique offer differentiated qualities, the best choice depends on factors linked to each specific context. On the one side, ML-based techniques should be chosen when it is possible to sustain the cost of (onerous) training processes. In return, they can ensure high performance levels. On the opposite side, reactive techniques do not require any training process, neither manual tuning, but they are effective mainly with high contention. Particularly, they represent a good choice when it is necessary to cope with high contention peaks in the workload (e.g. in the case of burstiness). As for the other techniques, i.e analytical model-based, feedback-driven and prediction-driven techniques, it is difficult to establish in advance when one of these types of techniques is more suitable than the others. Rather, the choice should be driven by the quality attributes of each single technique depending on the specific context requirements.

As a last observation, we remark that another potential advantage offered by scheduling techniques is related to energy efficiency. The study on STM systems presented in [48] show that a relevant secondary effect when using a scheduling technique may be the reduction of energy consumption. In effect, since scheduling techniques aim at reducing the wasted work, this can lead to a reduction of the wasted energy. However, relations between performance improvement and energy efficiency need to be further analysed and quantified.

## 8 The case of HTMs

In the introduction of this article, we mentioned that the TM hardware support recently offered by some commercial processors is encouraging research also on scheduling techniques for HTMs. In this section, we focus on HTMs, with the aim of discussing which results of studies on STMs may support the development of scheduling techniques also for HTMs.

Preliminary studies on scheduling techniques for HTMs were conducted via simulated environments, due to the lack of processors offering TM hardware support. Mostly, the proposed techniques use prediction-based strategies. For example, the technique in [49] implements a confidence-based predictor that estimates the probability of future conflicts between transactions by profiling patterns of past transaction conflicts. The proposal in [50] uses a similar approach and introduces a specialized Bloom filter to infer future transaction behaviour in terms of memory accesses. The technique in [51] keeps track of past conflicts in order to group threads that execute transactions which have high probability of conflict. Hence, threads belonging to same group are not allowed to concurrently execute transactions. The common feature of these techniques is that all of them require custom extensions to the hardware architecture in order to speed up operations executed by the scheduler. Similarly, Steal-On-Abort was later re-designed for HTM, and requires specific hardware extensions to support transaction queuing [52]. Unfortunately, the currently available HTM implementations do not offer specialized hardware as required by the above techniques. Differently, the technique presented in [53] was designed for existing HTMs and was evaluated on top of Intel(R) Haswell processors that currently offer HTM support. The scheduler uses a lightweight probabilistic technique that gathers information about the set of concurrently running transactions upon abort and commit events. This information is used to select the transactions to serialize to prevent future conflicts.

To understand which approaches and results of studies on STMs can be helpful in the case of HTMs, it is worthy to consider that STM and HTM implementations are very different in some aspects. Particularly, some factors that cause

transactions to be aborted in HTMs do not exist in the case of STMs. Examples include aborts due to transactions data footprint exceeding the cache capacity, and due to interrupts generating user/kernel mode changes. In general, in HTMs, transactions have to be relatively short to take advantage of hardware processing capability, since long transactions may require software fall-back path to be successfully processed. Another aspect is that in HTMs it is more complex to trace transaction conflicts, since they are transparently resolved in hardware [53].

In practice, schedulers that use a number of metadata to operate, that require fine-grained tracking of transaction conflicts/data accesses, or that need precise information about conflicting transactions [53], may not be easily adapted to existing HTMs. We note that techniques like Shrink, CAR-STM, Steal-on-Abort and RelSTM, some model-based techniques like SAC-STM and CSR-STM, and some mixed techniques like LUTS and ProVIT, show at least one of the above requirements. Thus, their porting to HTMs could be complex, or even infeasible. Conversely, most of the feedback-driven strategies (like ATS, PoCC, F2C2 and Chan et Al. 2011, and the lightweight prediction-based technique SCA) use a few metadata, do not require fine-grained tracking, and do not need to identify couples of conflicting transactions. Hence, they could be more easily ported to current HTM implementations. Also, regarding the analytical model-based technique MCATS, given that it performs coarse-grained workload tracking, its porting to HTMs may be more feasible with respect to the other model-based techniques.

In any way, we remark that, given the number of factors that differentiate STM from HTM implementations, most of schedulers of the above-mentioned techniques should be re-designed and optimized on the basis of the specific characteristics of HTMs. This would be a necessary step to understand their effectiveness also in the case of HTMs.

## 9 Summary

In this article, we reviewed literature studies on scheduling techniques for STMs. We pointed out the characteristics of different scheduling strategies, based on which we derived a classification of the existing techniques. Also, we analysed previous experimental studies, and we discussed the results of a comparison study of techniques based on alternative scheduling strategies. By the experimental results, we identified advantages and drawbacks of the different techniques, and we identified the issues to be further investigated. Finally, we discussed the aspects that should be considered when moving to the case of HTMs.

## References


[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[2] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213.

[3] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 253–262.

[4] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 169–178.

[5] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, and I. Watson, "Adaptive concurrency control for transactional memory," in *In MULTIPROG 08: First Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[6] 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*. Springer-Verlag, 2008, pp. 719–728.

[7] 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 IASTED-PDCN*. ACTA Press, 2011, pp. 91–98.

[8] M. Ansari, "Weighted adaptive concurrency control for software transactional memory," *J. Supercomput.*, vol. 68, no. 3, pp. 1027–1047, Jun. 2014.

[9] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for stms," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 927–938.

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

[11] E. Atoofian, "Speculative contention avoidance in software transactional memory," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 1417–1423.

[12] S. Dolev, D. Hendler, and A. Suissa, "Car-stm: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, ser. PODC '08. New York, NY, USA: ACM, 2008, pp. 125–134.

[13] 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*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 4–18.

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

[15] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proc. 20th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2012, pp. 278–285.

[16] ——, "Dynamic feature selection for machine-learning based concurrency regulation in stm," in *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 68–75.

[17] 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 *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, Sept 2013, pp. 31–40.

[18] P. Di Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, "Markov chain-based adaptive scheduling in software transactional memory," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2016. Washington, DC, USA: IEEE Computer Society, 2016.

[19] 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.

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

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel transactional synchronization extensions for high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and*

*Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 19:1–19:11.

[22] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proc. 20th Intl. Symp. on Distributed Computing*, 2006.

[23] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 175–184.

[24] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir, "Transactional contention management as a non-clairvoyant scheduling problem," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '06. New York, NY, USA: ACM, 2006, pp. 308–315.

[25] G. Sharma, B. Estrade, and C. Busch, "Window-based greedy contention management for transactional memory," in *Proceedings of the 24th International Conference on Distributed Computing*, ser. DISC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 64–78.

[26] G. Sharma and C. Busch, "A competitive analysis for balanced transactional memory workloads," in *Proceedings of the 14th International Conference on Principles of Distributed Systems*, ser. OPODIS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 348–363.

[27] I. Watson, C. Kirkham, and M. Lujan, "A study of a transactional parallel routing algorithm," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 388–398.

[28] F. Rubin, "The lee path connection algorithm," *IEEE Trans. Comput.*, vol. 23, no. 9, pp. 907–914, Sep. 1974.

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

[30] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 315–324, March 2007.

[31] P. K. Janert, *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[32] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[33] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[34] 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 Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO. IEEE Computer Society, Sep. 2013.

[35] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Analytical/ml mixed approach for concurrency regulation in software transactional memory," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 81–91.

[36] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," Georgia Institute of Technology, School of Electrical and Computer Engineering, Tech. Rep., 02 2007.

[37] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," University of Rochester, Department of Computer Science, Tech. Rep. 893, March 2006.

[38] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," *SIGPLAN Not.*, vol. 44, pp. 155–165, June 2009.

[39] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 124–134.

[40] T. Heber, D. Hendler, and A. Suissa, "On the impact of serializing contention management on stm performance," *J. Parallel Distrib. Comput.*, vol. 72, no. 6, pp. 739–750, Jun. 2012.

[41] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott, "Implementing and exploiting inevitability in software transactional memory," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 59–66.

[42] P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, no. 4, pp. 831–872, Sep. 1993.

[43] H. Rito and J. Cachopo, "Flashbackstm: Improving stm performance by remembering the past," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, H. Kasahara and K. Kimura, Eds. Springer Berlin Heidelberg, 2013, vol. 7760, pp. 266–267.

[44] I. Valença, T. Lucas, T. Ludermir, and M. Valença, "Selecting variables with search algorithms and neural networks to improve the process of time series forecasting," *Int. J. Hybrid Intell. Syst.*, vol. 8, no. 3, pp. 129–141, Aug. 2011.

[45] 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.).

[46] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 145–156.

[47] R. Ennals and R. Ennals, "Software transactional memory should not be obstruction-free," Technical Report IRC-TR–06–052, Intel Research Cambridge Tech Report, Jan 2006, Tech. Rep., 2006.

[48] D. Rughetti, P. Di Sanzo, and A. Pellegrini, "Adaptive transactional memories: Performance and energy consumption tradeoffs," in *Network Cloud Computing and Applications (NCCA), 2014 IEEE 3rd Symposium on*. IEEE Computer Society, Feb 2014, pp. 105–112.

[49] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 156–167.

[50] ——, "Bloom filter guided transaction scheduling," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 75–86.

[51] D. Choi, S. H. Kim, and W. W. Ro, "Conflict avoidance scheduling using grouping list for transactional memory," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 547–556.

[52] M. Ansari, B. Khan, M. Luján, C. Kotselidis, C. Kirkham, and I. Watson, "Improving performance by reducing aborts in hardware transactional memory," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 35–49.

[53] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15. New York, NY, USA: ACM, 2015, pp. 224–233.

**Pierangelo Di Sanzo** received a M.S. degree and a Ph.D. degree in Computer Engineering from Sapienza University of Rome. He was an associate researcher at the Italian National Interuniversity Consortium for Informatics, and currently he is a postdoctoral researcher at the Department of Computer, Control, and Management Engineering, at Sapienza University of Rome. His research interests lie in the area of concurrent programming and transactional systems, with special interest on performance analysis, modelling, optimization and energy efficiency.