# Machine Learning-based Thread-Parallelism Regulation in Software Transactional Memory

Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani
DIAG - Sapienza Università di Roma

Francesco Quaglia
DICII - Università di Roma "Tor Vergata"

**Abstract**

Transactional Memory (TM) stands as a powerful paradigm for manipulating shared data in concurrent applications. It avoids the drawbacks of coarse grain locking schemes, namely the potentially excessive limitation of concurrency, while jointly providing support for synchronization transparency to the programmers, which is achieved by embedding code-blocks accessing shared data within transactions. On the downside, excessive transaction aborts may arise in scenarios with non-negligible volumes of conflicting data accesses, which might significantly impair performance. TM needs therefore to resort to methods enabling applications to run with the maximum degree of transaction concurrency that still avoids thrashing. In this article we focus on Software TM (STM) implementations, and present a machine learning-based approach that enables the dynamic selection of the best suited number of threads to be kept alive along specific phases of the execution of STM applications, depending on (variations of) the shared data access pattern. Two key contributions are provided with our approach: (i) the identification of the well suited set of features allowing the instantiation of a reliable neural network-based performance model and (ii) the introduction of mechanisms enabling the reduction of the run-time overhead for sampling these features. We integrated a real implementation of our machine learning-based thread-parallelism regulation approach within the TinySTM open source package and present experimental data, based on the STAMP benchmark suite, which show the effectiveness of the presented thread-parallelism regulation policy in optimizing transaction throughput.

**Keywords:** concurrency, transactional memory, performance prediction, performance optimization

# 1 Introduction

Over the last decade multi-core systems have become mainstream computing platforms so that off-the-shelf desktop and laptop machines are nowadays equipped with multiple processors and/or CPU-cores. Consequently, great interest has came out on easy to manage abstractions able to simplify the job of developing concurrent applications accessing shared data.

While the traditional coarse-grained locking approach represents a simple management solution, it is prone to excessive reduction of concurrency. Thus it typically does not allow scalability on medium-to-high end multi-core systems. On the opposite side, fine grain locking is more complex to manage, and not suited for enabling adequate productivity by the programmers. Transactional Memory (TM) is a shared data management paradigm that allows the exploitation of parallelism—hence avoiding the drawbacks of coarse grain locking—while jointly providing synchronization transparency to the programmer, which is not allowed by hand-made fine grain locking.

TM allows the programmer to mark code blocks accessing shared data as transactions, whose atomicity and isolated execution is transparently ensured by hardware or software support. In the former case we talk about Hardware TM (HTM), a support offered by some modern processors families, such as the Intel Haswell [1]. It manages a transactional code block by having any of its updates temporarily buffered at the level of the caching system, and flushed out of the cache only upon a successful commit (e.g. when no other core has concurrently updated/read data that is written by the transaction). A core advantage of HTM is the minimal overhead for managing transactional data accesses, given the reliance on processor firmware. However, HTM implementations still suffer from significant limitations, such as the impossibility to manage transactions with data sets exceeding the cache capacity[1].

Software TM (STM) [2] is the counterpart software implementation of the TM paradigm. In STM, the management of transactional data accesses is purely demanded from a software layer, which encapsulates memory operations in such a way that any update is installed (made visible to other threads) in a controlled manner, e.g. upon the successful commit of a transaction. Various STM implementations exist, based on differentiated strategies for handling transaction concurrency [3, 4, 5, 6, 7]. Such an approach does not require any transaction-oriented hardware support and avoids the limitations of current HTM implementations. Particularly, by avoiding transaction aborts that are not directly related to conflicting data accesses (e.g. aborts caused by limited cache capacity).

However, the STM paradigm still requires to be complemented with methods and techniques aimed at avoiding excessive incidence of aborts (namely thrashing). They are caused by data conflicts across concurrent transactions, which is an issue that anyway affects also HTM. In fact, data conflicts may lead to unacceptably reduced operations/tasks throughput and to poor energy efficiency caused by the squash of non-committable transactional work.

In this article we address the problem of STM throughput optimization by introducing a Machine Learning (ML)-based approach [8]. We introduce a neural network-based model able to predict the variation of the transactions' throughput as a function of the number of threads used for running the application. The model is exploited at run-time by a thread-parallelism regulation system that dynamically selects the best suited thread concurrency level (via threads' suspension/resume), say the one avoiding both under-parallelism and over-parallelism in the current phase of the application execution. The former can prevent full exploitation of the available hardware resources, while the latter can originate the

[1]This problem is exacerbated by the fact that an HTM transaction is forced to abort also by conflicting cache-line accesses by multiple CPU-cores sharing the same cache hierarchy path.

aforementioned transaction trashing phenomena, just depending on the (dynamic) data access pattern by the application.

As for methodological contributions, our proposal does not only target the identification of a reference set of features to be taken into account while building the neural network-based throughput predictor. Rather, we also present a technique for dynamically shrinking the features' reference set, thus generating reduced subsets of actual features to by run-time sampled in order to correctly characterize the workload and fill the neural network-based predictor in input. This allows reducing the overhead incurred in by thread-parallelism regulation.

To evaluate the effectiveness of our proposal, we integrated the thread-parallelism regulation architecture within TinySTM [4], a popular open source STM layer written in C language, and we carried out an extensive experimental study based on the STAMP benchmark suite [11]. We compared our ML-based proposal with other literature techniques targeting the reduction of the incidence of aborts in STM. The experimental data show the higher effectiveness and robustness of our approach in optimizing the performance of STM applications in face of differentiated (and dynamic) workload profiles.

The reminder of this article is organized as follows. In Section 2 we discuss related work. A recap on the core ML method we use, namely neural networks, is provided in Section 3. The ML-based thread-parallelism regulation approach is presented in Section 4. Experimental results are reported in Section 5.

## 2 Related Work

An approach targeting the reduction of the incidence of aborts in TM applications is transaction scheduling. It is based on serializing transactions in order to avoid (highly likely) conflicting ones to execute concurrently. Solutions oriented to HTM are faced with the problem of identifying transaction' data access patterns, since memory accesses within transactions are not under the control of any software layer and, at the current state of HTM technology, cannot be directly audited. In [12], this problem is tackled via a probabilistic technique aimed at inferring potential conflicts, which is used to serialize transactions accordingly via locking mechanisms. The Adaptive Thread Scheduling (ATS) proposal in [13] exploits the notion of Contention Intensity (CI), which quantifies the incidence of aborts (as a simple—dynamically computed—ratio between aborted and executed transactions). If CI oversteps a given threshold value, then transactions are serialized via a lock-based mechanism. Although this approach has been proposed for (and evaluated with) STM applications, it can still be useful for HTM contexts since it does not rely on any data access pattern information.

Most of the transaction scheduling techniques specifically tailored for STM applications make use of information on data access patterns or conflicts' materialization, e.g., as gathered by the STM layer at run-time. The proposal in [14] assigns transactions to a same thread (thus sequentializing them) if they are recognized to access the same data portions, a target that is achieved by also exploiting indications provided by the application programmers. A similar technique of partitioning and serializing the transactions across different queues (managed by different threads) has been presented in [15], where the move of transactions across the queues is based on cross transactions' conflict**s** detected by the STM layer at run-time. A kind of specialization of this approach for the case of long running transactions, whose aborts may negatively impact resources' usage at a larger extent, has been presented in [16]. In [17] the authors present Shrink, a scheduler that serializes a transaction when a potential conflict with other running transactions is predicted. The prediction leverages on the estimation of expected read/write

sets, carried out on the basis of the history of processed transactions. To reduce the run-time overhead, Shrink activates the serialization mechanism only when the fraction of aborted transactions along an execution interval exceeds a given threshold. A different approach has been recently taken in [18], where the decision on whether to serialize transactions is based on an analytical performance model relying on Markon-chain formalisms. This model does not require knowledge on the data access pattern in order to be instantiated at run-time. Rather it only needs to be filled with simple parameters, such as the average time spent for executing aborted work, and the abort probability as observed when running with a single parallelism degree.

Compared to all the above proposals, we follow the orthogonal approach of thread-parallelism regulation, rather than transaction scheduling. Thus, we control the degree of concurrency across transactions by controlling (and dynamically adjusting) the overall level of thread-parallelism. One advantage from this approach is that thread suspension (in phases where the degree of parallelism needs to be shrunk for a while) can be operated by relying on different schemes, including those based on operating system blocking services. Instead, most of the literature transaction schedulers rely on busy waiting (e.g., via spin-locking) since the wait phase of a serialized transaction needs to be very short (especially for very fine grain applications), thus being typically not compatible with temporized waits supported by the operating system. Overall, as opposed to these approaches, thread-parallelism regulation has the potential to also provide improvements in energy efficiency. Further, except for the approach in [18], all the above proposals do not directly estimate (or model) the wasted time due to aborted transactions (vs the level of concurrency), while they only indirectly attempt to reduce the wasted time according to heuristics, e.g. threshold-based schemes. Rather, our ML-based performance model predicts such wasted time, and drives changes in the level of concurrency on the basis of the prediction.

As for literature studies targeting the determination of the well suited level of thread parallelism in TM applications, we find model based solutions, as well as heuristic approaches. In [19], an analytical modeling approach has been proposed to evaluate the performance of STM applications as a function of the number of concurrent threads and a set of workload profile parameters. Such an approach is targeted at building mathematical tools allowing the analysis of the effects of the contention management scheme on performance. This approach requires detailed knowledge on the conflict detection and management scheme used by the target STM, which is instead not required by the ML-based approach we propose.

The work in [20] presents an analytical model that takes as input a workload characterization of the application expressed in terms of transaction profiles, contention probability and hardware resources consumption. The model predicts the application execution time as function of the number of concurrent threads sustaining the application. However, the prediction is a representation of the average system behavior over the whole lifetime of the application. Hence, differently from our proposal, no ability to capture run-time variations (with consequent dynamic adaptation of the level of concurrency) is envisaged.

The proposal in [21] is targeted at evaluating scalability of STM systems. It relies on the usage of different types of functions (such as polynomial, rational and logarithmic functions) to estimate the performance of the application when considering different amounts of concurrent threads. The estimation process is based on measuring the speed-up of the application over a set of runs, each one executed with a different number of concurrent threads. Measurements are then used for calculating the parameters of a set of alternative interpolating functions, so as to select the best one that can be used to predict the speed-up of the application vs the number of threads. Differently from our proposal, this approach has the limitation of not taking into account the workload profile of the application. Hence the prediction may prove unreliable when the profile changes along different application execution phases.

The solutions in [22, 23] rely on the usage of parametric analytical expressions capturing the relation between thread-level parallelism and throughput of STM applications. The main objective of these works is to provide meta-models that can be promptly instantiated for thread concurrency adaptation with specific workloads. However, as shown by the results reported there in, the level of precision of these models needs to be refined via alternative (non-analytical) modeling strategies if extreme optimization of performance is targeted. In this work we take the orthogonal path of modeling the application performance (vs the degree of thread parallelism) via pure ML techniques.

Approaches based on hill climbing heuristics have been presented in [24, 25, 26], which dynamically increases or decrease the number of concurrent threads (or transactions) in STM applications. They determine whether the trend of increasing/decresing the concurrency level has positive effects on transactions' throughput, in which case the trend is maintained. Differently from our proposal, no direct attempt to capture the relation between the actual transaction profile and the achievable performance (depending on the level of parallelism) is done.

As for pure ML proposals, the work in [27] presents a classification-based approach for selecting the suited level of thread parallelism in HTM applications. The employed classification algorithms are based on input/output data specifically devised to account for HTM peculiarities (such as transaction aborts due to cache capacity), which are not proper of the STM context. In our proposal we provide a ML-based performance model which is specifically suited for STM, and which takes into account transaction data access profiles (which is not viable in HTM due to difficulties/costs for auditing transactional data accesses that are directly handled by the processor firmware). Also, we introduce an innovative approach for dynamically shrinking the set of features to be run-time sampled in order to re-evaluated the machine learning model and to regulate thread concurrency. This aspect is not addressed by the proposal in [27].

In [28] ML techniques are used to select the best performing conflict detection and management algorithm at run-time. Specifically, an application is profiled while running with different algorithm implementations, and the collected data are used by learning algorithms to decide the specific implementation to be used along the execution of the application. In [29], ML is used to select the most suitable thread mapping, i.e., the placement of application threads on different CPU-cores in order to get the best performance. The goals of both these works are different with respect to our one, since we focus on the regulation of the overall concurrency level while running STM applications. Ideally, such different techniques could be combined together.

Finally, in [30] the authors address the issue of multidimensional dynamic optimization of TM systems. This is achieved via the construction of a TM environment that makes different contention management policies (and implementations) coexist, while jointly controlling at run-time the level of concurrency. In this work the run-time decisions are taken on the basis of recommendation schemes and Bayesian optimization techniques. Our work differs from this proposal because we enable thread-parallelism regulation via ML by jointly exploiting a technique where the set of input features to be sampled an filled in input to the ML-based performance model is dynamicaly adapted to the actual workload.

# 3 Recap of Neural Networks

A Neural Network (NN) is a ML method [8] that can approximate various kinds of functions, including real-valued and discrete-valued ones. Inspired to the neural structure of the human brain, NN consists of a set of interconnected processing elements which cooperate to compute a specific function, so that, provided a given input, NN can be used to estimate the output of the function. Each processing element

calculates a (simpler) function, called transfer function. Different types of processing elements have been designed, each one calculating a specific transfer function. Commonly used ones are:

- **perceptron**—an element that takes a vector of real valued as input, calculates a linear combination of them and then outputs the value 1 if the result is greater that a given threshold, the value -1 otherwise. More formally, perceptron elements carry out the computation expressed by the following equation:

$$o(\vec{x}) = sign(\vec{w} \cdot \vec{x}) \tag{1}$$

  where $(\vec{x})$ is the perceptron input vector and $(\vec{w})$ is the weight vector.

- **linear unit**—an element that takes a vector of real valued as input, and outputs the weighted sum of the input plus a bias term. More formally:

$$o(\vec{x}) = (\vec{w} \cdot \vec{x}) \tag{2}$$

  where $(\vec{x})$ is the linear unit input vector and $(\vec{w})$ is the weight vector.

- **sigmoid unit**—it is similar to perceptron, but relies on a smoothed, differentiable threshold function. So the output of this element is a non-linear, differentiable function of its inputs. The sigmoid processing element computes its output as:

$$o(\vec{x}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x})}} \tag{3}$$

  where $(\vec{x})$ is the sigmoid unit input vector and $(\vec{w})$ is the weight vector.

In order to approximate a function $f$ via NN, a *learning algorithm* is used to compute the weights associated with the edges that connect the network processing elements. The learning algorithm exploits a set $\{(\mathbf{i}, \mathbf{o})\}$ of samples, called *training set*, where, for each sample $(\mathbf{i}, \mathbf{o})$, it is assumed that $\mathbf{o} = f(\mathbf{i}) + \delta$, where $\delta$ is a random variable (also said *noise*). Essentially, each sample provides the training algorithm with information on the relation between the input and the output of the function $f$ to be approximated.

A learning algorithm usually works according to an iterative procedure, where it performs the following computation at each iteration step. For each sample $(\mathbf{i}, \mathbf{o})$ in the training set, it takes the output associated with the input $\mathbf{i}$ as computed by NN and then compares it with the output $\mathbf{o}$ kept by the sample, thus determining the error. Then, the algorithm modifies the weights associated with the edges interconnecting NN elements with the aim at minimizing the overall error for the whole training set. The iterative procedure can be stopped after a number of steps have been executed or when the error falls below a given threshold. Various learning algorithms have been proposed [9, 10]. The design of a learning algorithm can be tailored to the NN topology and to the specific type of computation NN is intended for. In order to approximate arbitrarily complex real valued functions, as we do in this study, a multi-layer non-acyclic sigmoid-based neural network can be used ([31, 32, 33]), which once fixed the set of elements and interconnection edges can be trained by relying on the commonly used Back-propagation algorithm [34]. Basically, this algorithm computes the weights by exploiting gradient descent in the attempt to minimize the mean squared error between NN computed outputs and the output values kept by training set samples.

# 4 Machine Learning-based Thread-Parallelism Regulation

## 4.1 Application and Performance Model

In our approach we assume that concurrent threads running the application can be paused or resumed, so that the number of concurrent threads can be dynamically changed to optimize the concurrency level along the application lifetime. We consider a very general scenario where the execution flow of any thread is allowed to alternate transactions and non-transactional code (*ntc*) blocks, which run outside of transactional context. The length of *ntc* blocks might be arbitrary, so that our performance model also captures cases where threads iteratively execute subsequent transactions with no (or minimal) non-transactional work in between them.

Any transaction starts with a *begin* operation and ends with a *commit* operation. When running a transaction, the thread can both (A) perform read/write accesses to shared data objects, and (B) execute operations that do not access shared data objects (e.g. they access local variables within the thread stack). Read/written shared data objects are included in the transaction read-set (write-set). If a conflict between two concurrent transactions occurs then one of the conflicting transactions is aborted and re-started. Right after the thread commits a transaction, some *ntc* block is allowed to start, which ends right before the execution of the *begin* operation of the subsequent transaction along the same thread.

A core challenge when predicting the performance of STM applications is the estimation of the expected transaction execution time, i.e. the elapsed time between the first execution of the *begin* operation and the successful *commit* operation of a transaction. This time interval is affected by the so-called *wasted time*, i.e. the time spent for aborted executions of the same transaction. Transactions' wasted time may, in its turn, depend on several factors, including workload profile (namely the length of transactions and the data access pattern) and run-time system parameters (e.g. number of concurrent threads and code processing speed). Further, both workload profile and run-time system parameters could change along the application lifetime, so that the (expected) wasted time could change over time too.

In order to build a performance prediction model to be used for dynamic thread-parallelism regulation and performance optimization, we exploit a function instantiated via NN, whose input parameters represent the set of input features we consider as relevant and representative of the relation between wasted time and workload profile/system settings. The function is as follows:

$$w_{time} = f(rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}, k) \tag{4}$$

where:

- $w_{time}$ is the average per-transaction wasted time;
- $rs_s$ is the average read-set size of transactions;
- $ws_s$ is the average write-set size of transactions;
- $rw_a$ (read-write conflict affinity) is an index providing an estimation of the likelihood that an object read by a transaction could also be written by another transaction;
- $ww_a$ (write-write conflict affinity) is an index providing an estimation of the likelihood that an object written by a transaction could also be written by another transaction;
- $t_{time}$ is the average execution time of a committed transaction run (i.e. the average execution time of a transaction run which is not aborted);

- $ntc_{time}$ is the average execution time of $ntc$ blocks, namely the average time a thread spends executing instructions between two subsequent transactions;
- $k$ is the number of concurrently running threads.

As a note, the input parameters $rw_a$ and $ww_a$ to the function $f$ represent probability values, hence being constrained in the interval $[0, 1]$.

For each input parameter to $f$, once keeping fixed the others, we can devise the following expectations:

- $rs_s$ and $ws_s$ can directly affect the abort probability. Particularly, the greater the number of shared objects a transaction reads (writes), the higher is the probability that other concurrent transactions update (at least) one of these objects, thus leading to higher likelihood of conflicts.
- $rw_a$ and $ww_a$ can directly affect the abort probability too. Particularly, the higher the probability to read/write some shared object, the higher the probability for a transaction to experience conflicts with concurrent writing transactions updating the same object, which leads to transaction aborts.
- $t_{time}$ can directly affect the wasted time, since long-running transaction instances are expected to give rise to larger wasted time than short-running ones, if aborted. Furthermore, longer transactions are expected to impact the abort probability also because of the longer length of so-called vulnerability windows [35], i.e. long-running transactions are expected to be more prone to be hit by some conflict. Overall, the average wasted time likely increases in scenarios where the average time for a successful transaction run is longer.
- $ntc_{time}$ can directly affect the abort probability, possibly leading to a reduction of transactions' wasted time. Particularly, the longer the execution time of $ntc$ blocks, the lower the likelihood that, while running some transaction along any thread, other concurrent threads also execute within transactional context.
- $k$ can directly affect the abort probability and the wasted time since transactions' concurrency just depends on thread level concurrency.

As we hinted, the above tendencies refer to the scenario where an individual input parameter to the function $f$ changes, with all the others being fixed. However, in real/complex workloads multiple input parameters to $f$ might simultaneously change over time. Hence, the objective of our NN based modeling approach is the one of providing a function, which we refer to as $f_{NN}$. It approximates $f$ and is able to capture complex effects on transactions' wasted time due to combined variations of all its input parameters. As we shall discuss in Section 4.3, the correlation and the variation of these parameters can be exploited in order to further improve the thread-parallelism regulation support operating at runtime, particularly by reducing its overhead. However, the approach we propose is intended to be a general one, so that it must be possible to instantiate the performance model independently of the actual parameters' correlation in the specific workload (or workload phase). To this end, we build $f_{NN}$ via the exploitation of a training set formed by $(\mathbf{i}, \mathbf{o})$ samples such that $\mathbf{i} = (rs_s^t, ws_s^t, rw_a^t, ww_a^t, t_{time}^t, ntc_{time}^t, k^t)$ and $\mathbf{o} = (w_{time}^t)$, where we use the apex $t$ to indicate that the involved quantities refer to training-phase data, rather than on-line data.

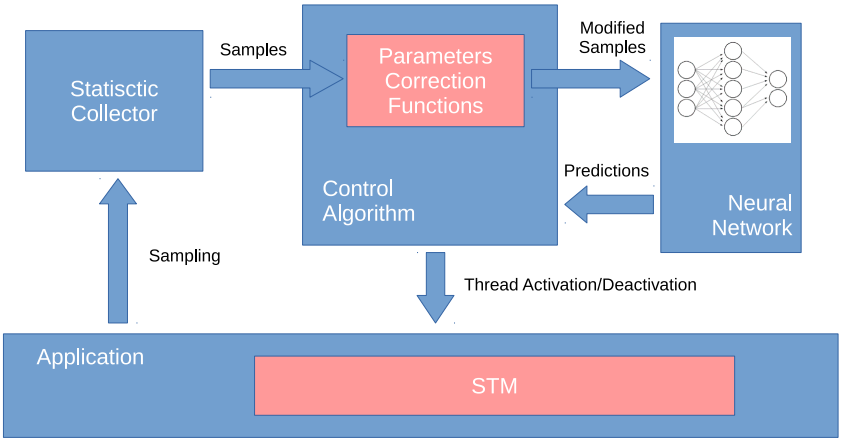


Figure 1: The thread-parallelism regulation architecture

## 4.2 The ML-based Thread-Parallelism Regulation Architecture

As schematized in Figure 1, our thread-parallelism regulation architecture relies on three building blocks, namely: a Statistics Collector (SC), a Neural Network (NN) and a Control Algorithm (CA). We consider the application lifetime as sliced into intervals, which we refer to as sampling intervals (or equivalently observation windows) where SC collects samples of the workload profile. At the end of each sampling interval, SC makes an estimation of the average values of the parameters: $rs_s$, $ws_s$, $t_{time}$ and $ntc_{time}$. Further, to estimate $rw_a$ and $ww_a$, SC initially estimates the probability distribution function of read and write operations over the whole set $S$ of shared data objects, in the form of histograms. After, $rw_a$ is estimated by calculating the dot product between the probability distribution functions of read and write operations, while $ww_a$ is estimated by calculating the dot product of the distribution function of write operations with itself. The estimated values of all the above parameters are assumed to be representative of the expected application behavior in the immediate future.

After, SC assembles the vector of estimates ($rs_s$, $ws_s$, $rw_a$, $ww_a$, $t_{time}$, $ntc_{time}$) and passes it to CA. Then, for each $k$ such that $1 \leq k \leq max_{threads}$, where $max_{threads}$ is the maximum admitted number of concurrent threads, CA generates the vector $\mathbf{v_k} = (rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}, k)$ and passes it as input to NN in order to compute the wasted time prediction $w_{time,k} = f_{NN}(\mathbf{v_k})$. Once the set of predictions $P = \{w_{time,k} | k \in [1, max_{threads}]\}$ is available, CA determines the number *opt* of concurrent threads which is expected to maximize the application throughput. Specifically, *opt* corresponds to the value of $k$ in the interval $[1, max_{threads}]$ for which the transactions' throughput, calculated as

$$thr_k = \frac{k}{w_{time,k} + t_{time} + ntc_{time}} \tag{5}$$

is maximized. Hence, during the subsequent sampling interval, CA keeps active *opt* threads. Note that in Equation 5 the quantity $(w_{time,k} + t_{time} + ntc_{time})$ corresponds to the predicted average time between the commit operations of two consecutive transactions along any thread when there are $k$ active threads.

Another point to note is that the vector ($rs_s$, $ws_s$, $rw_a$, $ww_a$, $t_{time}$, $ntc_{time}$)**,** produced by SC at the end of a sapling interval**,** is estimated on the basis of measurements collected while the application runs with a given number of concurrent threads, say the one that has been selected at the end of the last sampling interval.

Therefore, using this vector as it is does not account for the fact that, while carrying out NN based

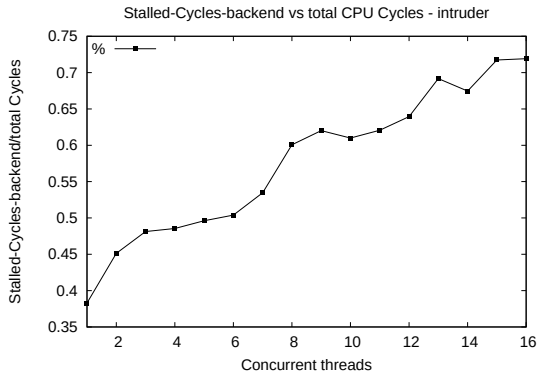


Figure 2: Stalled cycles back-end - Intruder benchmark

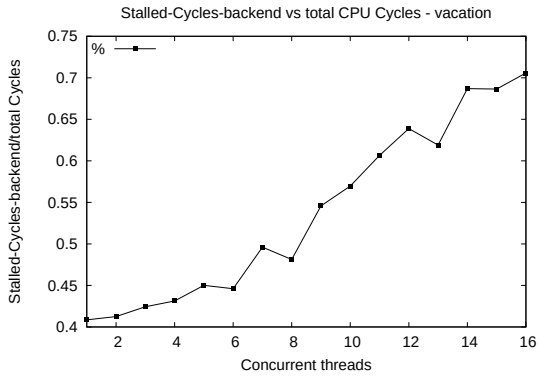


Figure 3: Stalled cycles back-end - Vacation benchmark

predictions via the $f_{NN}$ function receiving $\mathbf{v_k}$ in input, $t_{time}$ and $ntc_{time}$ may depend, in their turn, on the value of $k$. This dependence can arise because of different thread contention dynamics on system level resources (e.g. contention on system hardware) when changing the number of concurrent threads running the application. As an example, per-thread cache efficiency may change depending on the number of concurrent threads operating on a given shared-cache level, thus impacting the CPU time required for specific code blocks, either transactional or non-transactional. To cope with this issue, we further refine our thread-parallelism regulation approach by basing it on correction functions. Once estimated the value of $t_{time}$ and $ntc_{time}$ when running with $i$ threads (which we denote as $t_{time,i}$ and $ntc_{time,i}$ respectively), these functions allow predicting the corresponding values when supposing a different number of threads. This leads CA to fill NN with input vectors $\mathbf{v_k}$ where the average processing latency of a successful transaction and the average processing latency of a *ntc* block will account for the specific concurrency level $k$ targeted by the prediction. Also, the final throughput prediction carried out by CA for the different values of $k$ will be actuated via the formula:

$$thr_k = \frac{k}{w_{time,k}(t_{time,k}, ntc_{time,k}) + t_{time,k} + ntc_{time,k}} \tag{6}$$

Overall, the finally achieved performance model in Equation 6 determines the expected transaction wasted time when also considering contention on hardware resources (not only shared data) while varying the number of concurrent threads.

The correcting functions aimed at determining (predicting) the values $t_{time,k}$ and $ntc_{time,k}$, once know (or estimated via sampling) the values of these same parameters when running with thread concurrency level $i \neq k$, are instantiated in our approach by exploiting the same data set used to train NN. Specifically, via curve fitting, we determine the application specific function expressing the variation (vs the level of thread concurrency) of the number of clock-cycles the CPU-core spends waiting for data or instructions to come-in from RAM storage, which we refer to as *stall cycles*. The expectation is that the number of stall cycles scales (almost) linearly vs the number of concurrent threads used for running the application, with a pendency factor that is application specific. To support our claim, we report in Figure 2 and in Figure 3 data related to two different STM-based applications from the STAMP benchmark suite, namely Intruder and Vacation, while varying the number of threads running these benchmarks in the range between 1 and 16. More in detail, the plots show the variation of the ratio between the stall cycles and the total amount of cycles for running the applications while varying thread level concurrency. These

data have been gathered by running these benchmarks on top of TinySTM and a HP ProLiant server equipped with two AMD Opteron$^{TM}$6128 Series Processor, each one having eight CPU-cores (for a total of 16 cores), and 32 GB RAM, running Linux SUSE Enterprise 10.4 (kernel version 2.6). The reported statistics have been collected via the `perf` tool, which marks the stall cycles as `Stalled-Cycles-backend`. By the curves the close-to-liner scaling is fairly evident, hence, once determined the scaling curve (so the pendency factor) via regression, which we denote as $sc$, we have that:

$$t_{time,k} = t_{time,i} \times \frac{sc(k)}{sc(i)} \qquad ntc_{time,k} = ntc_{time,i} \times \frac{sc(k)}{sc(i)} \tag{7}$$

where:

- $t_{time,k}$ is the estimate of the expected CPU time (once known/estimated $t_{time,i}$) for a committed transaction if the application runs with level of thread concurrency set to $k$;
- $ntc_{time,k}$ is the estimate of the expected CPU time (once known/estimated $ntc_{t,i}$) for a non-transactional code block in case the application runs with level of thread concurrency set to $k$;
- $sc(k)$ (resp $sc(i)$) is the value of the correction function for level of thread concurrency set to $k$ (resp $i$).

### 4.3 Reducing the Sampling Overhead: Dynamic Feature Selection

In this section we discuss an improvement aimed at reducing the run-time overhead for sampling the workload profile. The sampling overhead is caused by the need for tracing the application execution and producing the set of statistically computed values $\{rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}\}$ to be passed from SC to CA. We remark that this set includes all the features we have identified as relevant (say the reference ones) for characterizing the application execution profile in our ML-based approach.

We now define an approach where, depending on the current execution profile of the application (which may change over time), the set of features to be sampled can be dynamically shrunk, or enlarged again towards the maximum cardinality. In all the execution phases where the parallelism regulator works with a shrunk features' set, the run-time overhead for the sampling process is reduced, with additional benefits on performance—beyond the ones achievable by dynamically controlling thread-level parallelism to the well suited value. More in detail, the samples for estimating the features in the above reference set need to be taken by performing work along the critical path of the thread that is currently running a transaction. Hence, avoiding the sampling of a few features removes that work. As an example, if the $rw_a$ feature needs to be estimated, then it implies that when a thread executes an access to transactional data, some meta-data need to be updated along that same thread to reflect such an access onto, e.g., histograms used to finally estimate $rw_a$. Details on how this process is actually carried out in our implementation are provided in Section 5.1.

The idea of dynamically shrinking the features' set is based on noting that:

**A:** the values of some feature may show small variance along a given execution interval, and/or

**B:** the values of some feature may be statistically correlated (also including the case of negative correlation) to the values of other features along a given execution interval.

Particularly, we can expect that (significant) variations of $w_{time}$, if any, do not depend on any feature exhibiting small variance along the current observation interval. On the other hand, in case of correlation

across a (sub)set of different features, the impact of variations of these values on $w_{time}$ is expected to be assessable by observing the variation of any individual feature in that (sub)set, an approach known as Correlation-based Feature Selection (CFS) [36] in the ML literature. If the above scenarios occur, we can build an estimating function for $w_{time}$ which, compared to the $f$ function in Equation 4, relies on a reduced number of input parameters.

For the reference set $\{rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}\}$, it comes natural to think about the following expectations for the correlation of subsets of the input features:

- the size of the transaction read-set/write-set may be correlated to the transaction execution time. In fact, the number of read/write operations executed by the transaction contributes to the actual transaction execution time. If this reveals true, $t_{time}$ and $rs_s$ (or alternatively $ws_s$) can be excluded;
- read-write and write-write conflict affinities may exhibit correlation. In fact, these two indexes are both affected by the distribution of write operations executed by transactions. If this reveals true, $rw_a$ or $ww_a$ can be excluded.

Further, depending on the actual application logic, generic sub-sets of the reference features might result correlated along a given execution interval, even if they are not naturally identified as potentially correlated with each other. Also, still depending on the application logic, any of the features in the reference set $\{rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}\}$ may exhibit small variance along a given execution interval, thus being candidate to be excluded from sampling.

To determine at what extent such an expectation materializes, and to observe whether the scenarios in points **A** and **B** can anyhow materialize independently of the initial expectation, we have performed a set of experiments still relying on the STAMP benchmark suite, and report in Table 1 data related to the observed correlation among the different features for all the applications from STAMP. All data refer to serial execution of the STAMP applications, carried out on the same multi-core platform that we have described in Section 4.2.

We note that serial execution is adequate for the purpose of this experimental study, which is only tailored to determine workload features that are essentially independent of the degree of execution parallelism. Specifically, given that correlation and variance are computed over feature-samples, each one representing an average value (over a set of individual samples taken along a sampling interval entailing 4000 transactions in this experiment), for the only parameters that can be potentially affected by hardware contention (e.g. bus-contention) in case of actual parallelization, namely $t_{time}$ and $ntc_{time}$, the corresponding spikes (if any) would be made irrelevant by the aggregation of the individual samples within the window related average.

Data in Table 1 confirms that our rationale can find justification in the actual behavior of STM-based applications, considering the execution patterns provided by the STAMP benchmark suite as a reliable/reasonable representation of typical applications' dynamics. In fact, we can observe that the correlation between $rw_a$ and $ww_a$ is higher than 0.8 for 4 out of 8 applications of the STAMP suite, the correlation between $rs_s$ and $t_{time}$ is higher than 0.8 for 3 out of 8 applications, the correlation between $ws_s$ and $t_{time}$ is higher than 0.8 for 2 out of 8 applications. Further, as reported in Table 2, we observed a reduced variance for $rw_a$ and/or $ww_a$ for many of the applications, and reduced variance for $rs_s$ and/or $ws_s$ in a few cases.

We have also carried out an experimental study aimed at assessing at what extent shrinking the set of input features to be sampled would allow reducing the overhead. This study is still based on the STAMP

**Ssca2**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,259 | 1 | | - | - | - |
| $rs_s$ | -0,166 | 0,190 | 1 | - | - | - |
| $ws_s$ | -0,166 | 0,190 | 1 | 1 | - | - |
| $rw_a$ | -0,024 | -0,638 | -0,136 | -0,136 | 1 | - |
| $ww_a$ | -0,001 | -0,629 | -0,210 | -0,210 | 0,992 | 1 |

**Yada**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,705 | 1 | - | - | - | - |
| $rs_s$ | 0,860 | 0,619 | 1 | - | - | - |
| $ws_s$ | 0,828 | 0,617 | 0,946 | 1 | - | - |
| $rw_a$ | -0,417 | -0,183 | -0,508 | -0,552 | 1 | - |
| $ww_a$ | -0,400 | -0,173 | -0,491 | -0,542 | 0,999 | 1 |

**Intruder**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,781 | 1 | - | - | - | - |
| $rs_s$ | 0,914 | 0,940 | 1 | - | - | - |
| $ws_s$ | 0,577 | 0,924 | 0,848 | 1 | - | - |
| $rw_a$ | 0,516 | -0,377 | -0,540 | -0,330 | 1 | - |
| $ww_a$ | 0,023 | -0,350 | -0,269 | -0,559 | 0,322 | 1 |

**Vacation**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,989 | 1 | - | - | - | - |
| $rs_s$ | 0,507 | 0,520 | 1 | - | - | - |
| $ws_s$ | 0,345 | 0,315 | -0,487 | 1 | - | - |
| $rw_a$ | -0,167 | -0,179 | 0,811 | 0,657 | 1 | - |
| $ww_a$ | -0,572 | -0,535 | 0,262 | -0,954 | -0,483 | 1 |

**Genome**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,012 | 1 | - | - | - | - |
| $rs_s$ | 0,352 | 0,902 | 1 | - | - | - |
| $ws_s$ | -0,742 | 0,492 | 0,158 | 1 | - | - |
| $rw_a$ | -0,584 | -0,202 | -0,397 | -0,422 | 1 | - |
| $ww_a$ | 0,040 | -0,027 | -0,009 | -0,049 | 0,064 | 1 |

**Labyrinth**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,993 | 1 | - | - | - | - |
| $rs_s$ | 0,992 | 0,991 | 1 | - | - | - |
| $ws_s$ | 0,992 | 0,992 | 0,999 | 1 | - | - |
| $rw_a$ | -0,521 | -0,500 | -0,495 | -0,492 | 1 | - |
| $ww_a$ | -0,332 | -0,277 | -0,273 | -0,267 | 0,714 | 1 |

**Kmeans**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,141 | 1 | - | - | - | - |
| $rs_s$ | 0,434 | 0,194 | 1 | - | - | - |
| $ws_s$ | -0,524 | 0,106 | 0,481 | 1 | - | - |
| $rw_a$ | -0,245 | -0,729 | 0,072 | 0,177 | 1 | - |
| $ww_a$ | -0,072 | -0,723 | 0,090 | 0,008 | 0,968 | 1 |

**Bayes**

| | $t_{time}$ | $ntc_{time}$ | $rs_s$ | $ws_s$ | $rw_a$ | $ww_a$ |
|---|---|---|---|---|---|---|
| $t_{time}$ | 1 | - | - | - | - | - |
| $ntc_{time}$ | 0,141 | 1 | - | - | - | - |
| $rs_s$ | 0,434 | 0,194 | 1 | - | - | - |
| $ws_s$ | -0,524 | 0,106 | 0,481 | 1 | - | - |
| $rw_a$ | -0,245 | -0,729 | 0,072 | 0,177 | 1 | - |
| $ww_a$ | -0,072 | -0,723 | 0,090 | 0,007 | 0,968 | 1 |

Table 1: Features' correlation for STAMP applications

suite, run on top of TinySTM. We have experimentally evaluated the sampling overhead while varying (a) the number of concurrent threads (between 1 and 16), and (b) the set of selected input features. Given that the reported data are related to performance aspects, this time it is worth providing some details on how the sampling process has been implemented. We used the CPU-time sampling facilities natively offered by TinySTM to determine $w_{time}$, $t_{time}$ and $ntc_{time}$. In addition, the evaluation of $rs_s$ and $ws_s$ as been based on instrumenting TinySTM code by including counters within the transactional context meta-data. Further, in order to compute the access distribution of read/write operations for determining $rw_a$ and $ww_a$, we added a read counter and a write counter to each element of the transaction lock array in TinySTM. At the end of the commit phase of a successfully committing transaction, the read/write counter for each lock associated with an item in the read/write set of the transaction is incremented.

In our experiments, the sampling overhead has been evaluated as the additional time required to complete the execution of the benchmark application when sampling is active along any thread, compared to the time required for executing the application with no active sampling. The platform used for the experiments is still the HP ProLiant multi-core machine presented in detail in Section 4.2. Sampling overhead data for all the applications from STAMP are reported in Figures 4-11. For completeness, we include graphs for both relative and absolute overhead values.

One important observation we can make when analyzing the results is that, once fixed the set of

| | **Ssca2** | **Intruder** | **Genome** | **Kmeans** |
|---|---|---|---|---|
| $t_t$ | $1,14 \cdot 10^5$ | $1,24 \cdot 10^7$ | $6,05 \cdot 10^7$ | $5,07 \cdot 10^5$ |
| $ntc_t$ | $2,27 \cdot 10^4$ | $1,19 \cdot 10^7$ | $1,09 \cdot 10^6$ | $1,65 \cdot 10^6$ |
| $rs_s$ | $1,5 \cdot 10^{-3}$ | 142 | 945 | 0,134 |
| $ws_s$ | $1,3 \cdot 10^{-3}$ | 4,311 | 0,958 | 7,61 |
| $rw_a$ | $2,87 \cdot 10^{-10}$ | $1,25 \cdot 10^{-5}$ | $7,46 \cdot 10^{-10}$ | $3,34 \cdot 10^{-4}$ |
| $ww_a$ | $1,03 \cdot 10^{-10}$ | $1,17 \cdot 10^{-3}$ | $4,95 \cdot 10^{-4}$ | $4,16 \cdot 10^{-4}$ |
| | | | | |
| | **Yada** | **Vacation** | **Labyrinth** | **Bayes** |
| $t_t$ | $7,01 \cdot 10^6$ | $8,82 \cdot 10^6$ | $3,24 \cdot 10^{12}$ | $5,07 \cdot 10^5$ |
| $ntc_t$ | $7,38 \cdot 10^4$ | $2,57 \cdot 10^5$ | $1,04 \cdot 10^7$ | $1,65 \cdot 10^6$ |
| $rs_s$ | 33 | 770 | 70 | 0,134 |
| $ws_s$ | 1,914 | 7,5 | 173 | 7,614 |
| $rw_a$ | $4,60 \cdot 10^{-6}$ | $4,16 \cdot 10^{-13}$ | $4,02 \cdot 10^{-7}$ | $3,34 \cdot 10^{-4}$ |
| $ww_a$ | $2,44 \cdot 10^{-5}$ | $1,76 \cdot 10^{-10}$ | $2,04 \cdot 10^{-7}$ | $4,16 \cdot 10^{-4}$ |

Table 2: Features' variance for STAMP applications

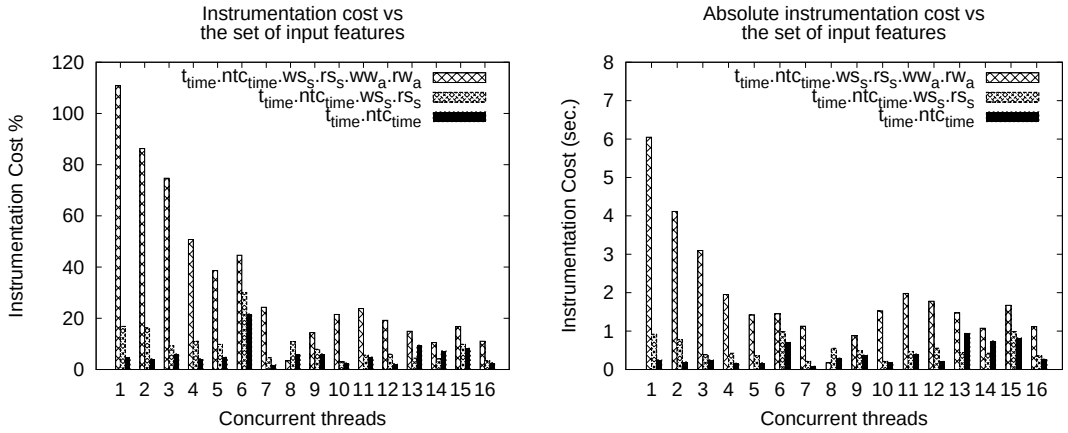


Figure 4: Sampling overhead for Intruder

input features to sample, for several of the tested benchmarks the percentage overhead caused by the instrumenting code used to take samples tends to scale down while the number of concurrent threads is increased. This behavior is related to a kind of throttling effect that manifests when any active thread is involved in the sampling process, since threads issue transactions with reduced rate because of delays associated with sampling activities along thread execution. In particular, when the degree of concurrency is increased, the throttling effect tends to reduce the number of transaction aborts, which tends to reduce the overhead observed when running with the sampling process active, compared to the case of no active sampling. This is a typical behavior for parallel computing applications entailing optimistic processing and rollback actions [37], as is the case of STM applications. This phenomenon is not observed for reduced values of the number of threads, which leads to reduced contention levels, hence to reduced abort probability, for which non-significative positive effects can be achieved thanks to throttled executions.

The scale down of the percentage overhead with higher numbers of concurrent threads also appears when sampling the full reference set of input features. As an example, the results in Figure 11 (left) show how for Yada the percentage overhead when sampling the whole reference set of features $\{rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}\}$ is of the order of about 80% when running with up to 4 threads, while it decreases to about 15% when running with 16 threads. However, the most significant reduction of

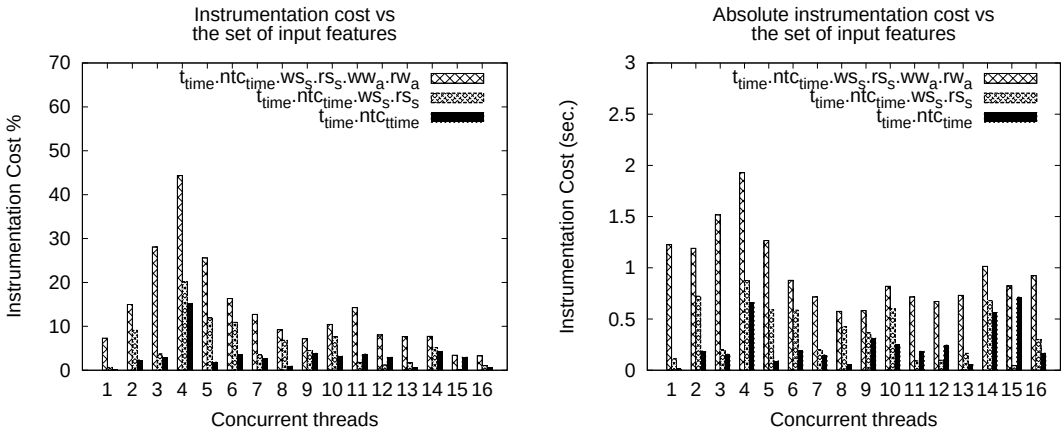


Figure 5: Sampling overhead for Bayes

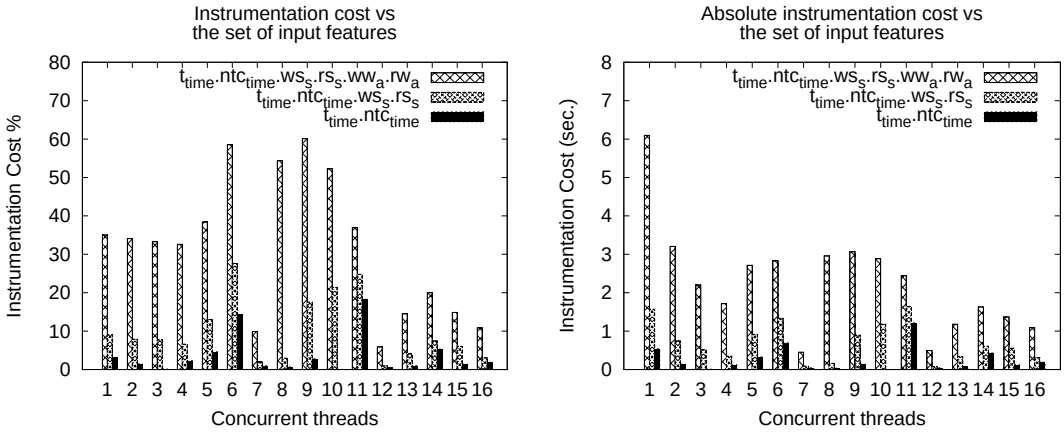


Figure 6: Sampling overhead for Genome

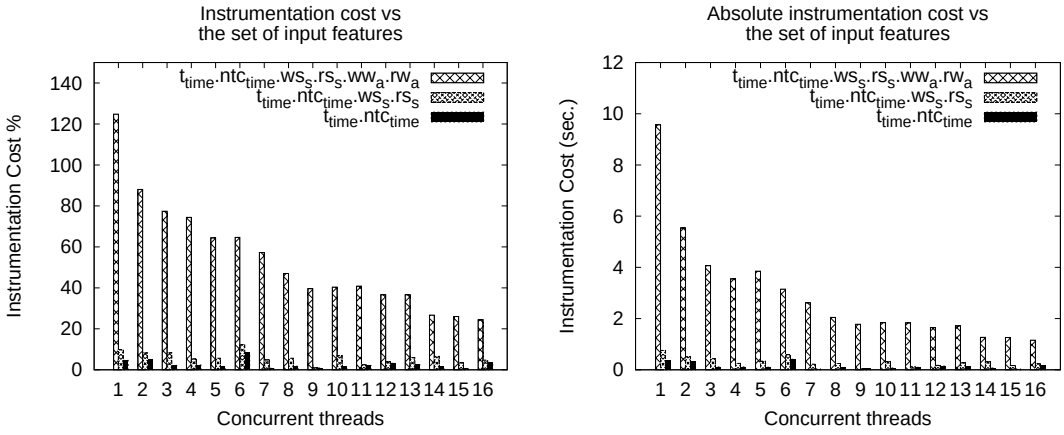


Figure 7: Sampling overhead for Ssca2

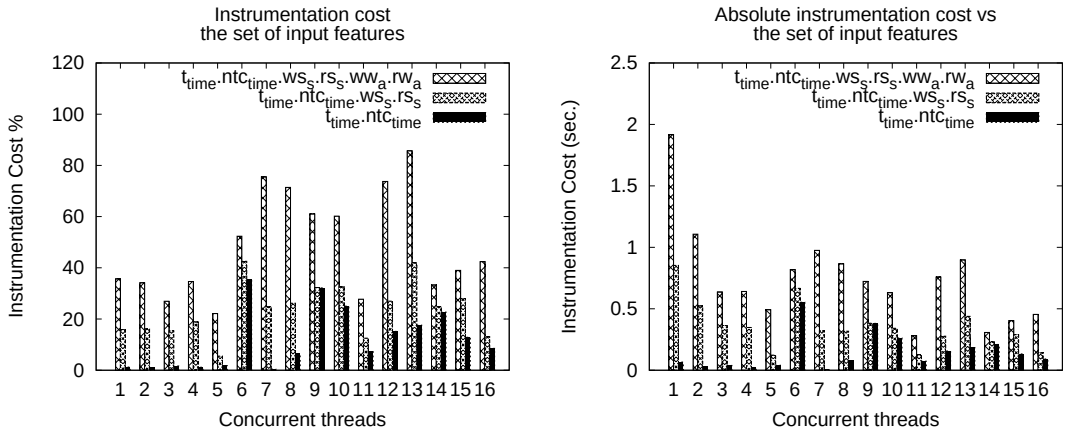


Figure 8: Sampling overhead for Vacation

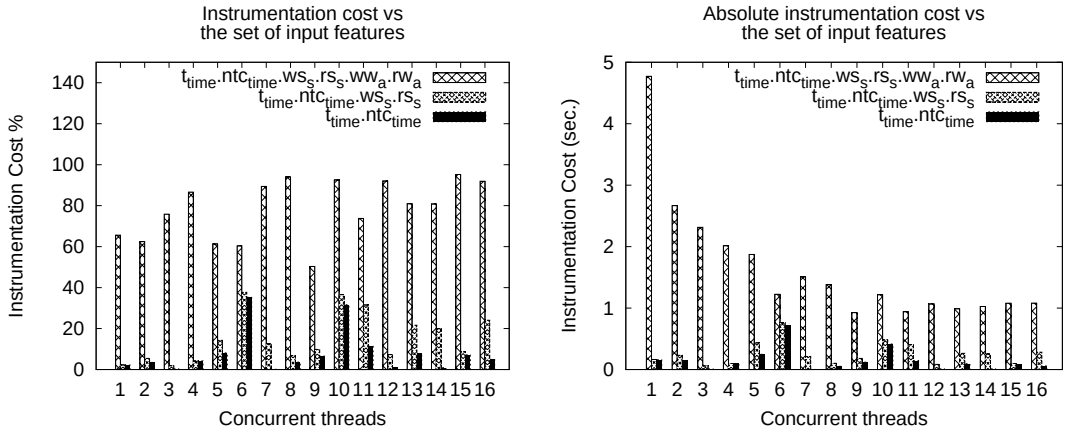


Figure 9: Sampling overhead for Kmeans

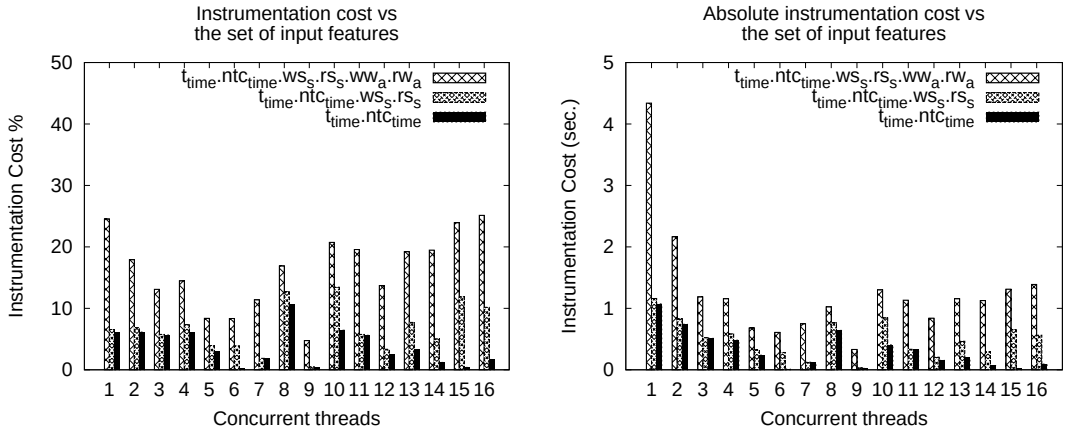


Figure 10: Sampling overhead for Labyrinth

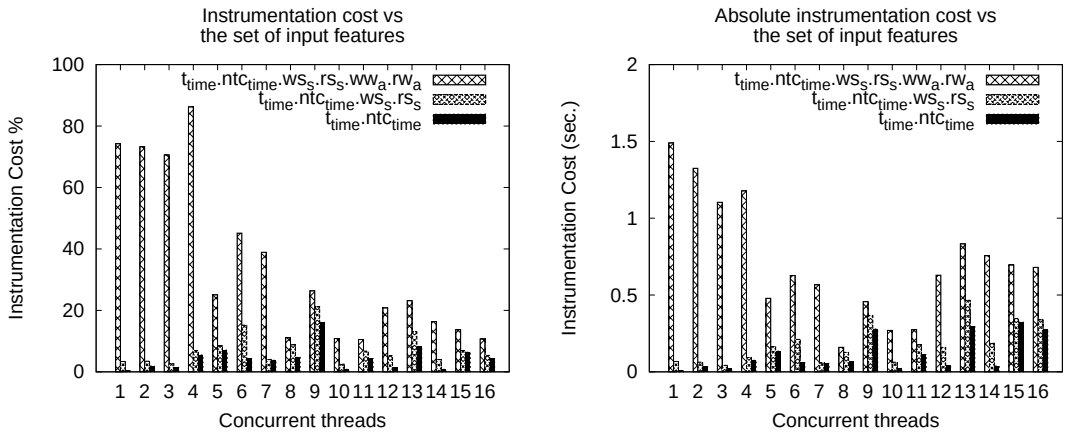


Figure 11: Sampling overhead for Yada

the overhead is observed in scenarios where the set of input features for which sampling is active has been shrunk. For example, as shown for the Intruder benchmark in the left graph of Figure 4, when shrinking the monitored features from 6 to 4 or 2, we get up to 90% reduction of the overhead for lower values of the number of concurrent threads (namely up to 6). This is highly relevant when considering that the optimal degree of parallelism for Intruder has been shown to be around 5-6 when running on the hardware platform used in this study (we refer the reader to Section 5 for actual performance data showing this tendency). In other words, the optimal parallelism degree can be achieved with a number of concurrent threads that does not allow the overhead due to sampling to be negligible if no optimized scheme for shrinking the set of input features to be sampled is provided.

#### 4.3.1 Shrinking vs Enlarging the Set of Input Features

As pointed out, some features of the reference set may be discarded at run-time because of low variance and/or high correlation. However, given that the application execution profile may vary over time, variance and correlation of excluded features may change so that they become again relevant to estimate $w_{time}$. As an example, two generic features $x$ and $y$ which exhibited correlation in some time in the past, such that one of them was excluded from the set of input features, may successively start behaving in an uncorrelated manner. Hence, the excluded feature should be re-included within the set.

Detecting this type of scenarios, in order to support the dynamic enlarging of the feature-set, cannot be based on run-time input features analysis (e.g. correlation analysis), since the feature that was excluded from the sampling process shows a behavior which is currently unknown. Hence, no fresh statistical data for that feature are available to detect whether variance and/or correlation with other features have changed.

To overcome this problem, our proposal relies on evaluating the accuracy of the wasted-time prediction by the ML-based performance model on the basis of the real wasted time (as observed at run-time at the end of a sampling interval). If the accuracy is detected to be low, the input feature set is enlarged towards the maximum in order to recover to a good workload characterization scenario. The index we have selected for determining the quality of the prediction is the *weighted root mean square error* (WRMS) of the predicted wasted time vs the corresponding real (measured) value.

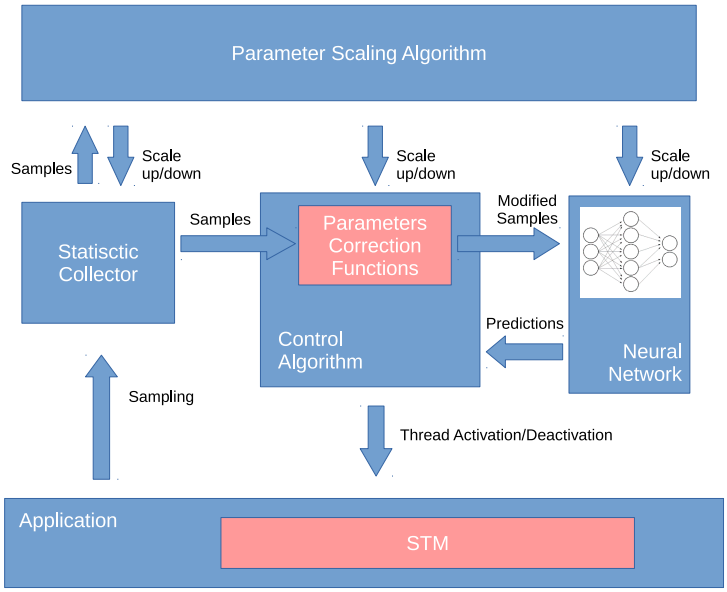


Figure 12: The thread-parallelism regulation architecture entailing dynamic feature selection capabilities

#### 4.3.2 The Dynamic Feature Selection-based Thread-Parallelism Regulation Architecture

To support dynamic selection of the relevant features, our approach relies on a set of NN instances (not a unique instance as instead it occurs in the baseline architecture presented in Section 4.2), each one able to manage a different feature-set and properly trained with that set. These NN instances can be trained in parallel (e.g. during the early phase of application processing). Then a so-called Parameter-Scaling-Algorithm (PSA) implemented within an additional module integrated in the architecture is exploited for dynamically scaling-up/down the set of features (also referred to as parameters in the final architecture) to be taken into account for concurrency regulation along the sub-sequent execution window. Thus PSA is aimed at determining the NN instance to be used in relation to the selected set of input features. The schematization of the architecture entailing dynamic feature selection capabilities is shown in Figure 12.

To select the best suited NN instance, and hence the sub-set of features that can be considered as reliably representative of the workload actual behavior, PSA periodically (i.e. at the end of the observation window) performs the following tasks: (1) It evaluates the quality of the prediction by the currently in use instance of NN (representative of the currently in use set of features) via the estimation of WRMS; (2) It analyzes the statistics related to the currently monitored features, to determine variance and correlation.

If the calculation of WRMS in point 1 gives rise to a value exceeding a specific threshold, then PSA enlarges the set of input features to the full set of features, which from now on we will refer to as $maxSet = \{rs_s, ws_s, rw_a, ww_a, t_{time}, ntc_{time}\}$. It then issues commands to SC, CA and NN in order to trigger their internal reconfiguration, leading to work with $maxSet$. This means that any query from CA during the subsequent observation window is answered by the NN instance trained over $maxSet$. On the other hand, if the WRMS value computed in point 1 does not exceed the threshold value, the analysis in point 2 is exploited to determine whether the currently used (sub-)set of features can be shrunk (and hence to determine whether a scale-down of the set of sampled parameters can be actuated). In particular, if the variance observed for a given feature is lower than a given threshold, the feature

is discarded from the set to be exploited in the next observation window. Then, for each couple of not discarded features, PSA calculates their correlation and, if it oversteps another threshold, one of them is discarded too. The non-discarded features form the optimized (shrunk) set of parameters to be exploited for concurrency regulation in the subsequent observation period, which we refer to as *minSet*. Then, similarly to what done before, PSA issues configuration commands to SC, CA and NN in order to trigger them for operating with *minSet*.

# 5 Experimental Assessment

In this section we present the results of an experimental study aimed at assessing the effectiveness of our ML-based thread-parallelism regulation architecture. We still relied on applications from the STAMP benchmark suite. As hinted, we implemented our architecture by integrating it with TinySTM, and we run all the experiments on top of the same HP ProLiant multi-core machine we have described in Section 4.2. The relevant details of our implementation are provided in Section 5.1 ([2]). We present the results for all the STAMP benchmarks, which show very differentiated workloads. They span from low to high percentage of time spent executing transactions (vs non-transactional code blocks), and from low to high incidence of data contention, according to differentiated transaction profiles. Further, we run each STAMP application with different configurations of its input parameters, which represent an additional factor affecting the workload profile and, consequently, the optimal level of thread-parallelism. As an additional note, each individual application, in any specific configuration of its input parameters, can generate transactions with varying execution profiles (e.g., non minimal variance of their execution time), as explicitly assessed in [11]. This is an important aspect since our proposal is based on a core performance model—the one in Equation 5—which exploits average values of the involved parameters. Therefore testing our solution with multiple workloads, each one entailing variance of the execution profile of transactions, increases the level of robustness of the experimental analysis.

## 5.1 Implementation Details

In our implementation we exploited TinySTM version 1.0 for Unix systems. We used the stopwatch facilities natively offered by TinySTM to determine $w_{time}$, $t_{time}$ and $ntc_{time}$ and we instrumented TinySTM code to take samples to evaluate $rs_s$ and $ws_s$, in a similar scheme to the one used for the experimentation in Section 4.3. In order to compute the access distribution of read/write operations, which determine $ww_a$ and $rw_a$, we added a read counter and a write counter for each element of the lock vector. At the end of the commit phase of a successfully committing transaction, the read/write counter for each lock associated with an item in the read/write set of the transaction is incremented. Clearly, along sampling intervals where the set of features to be sampled is shrunk, the code block that is used to take samples of discarded features is not activated along that thread, which occurs by simply checking boolean conditions. Such conditions are restored to the value "true" each time the corresponding feature is reinserted in the set of the ones to be sampled. This occurs (if needed) at the beginning of a new sampling interval, according to the specifications provided in Section 4.3.2.

Along any sampling interval, samples produced by all the threads are gathered by a single thread, which we name *master* thread. This choice permits not to affect system scalability as thread synchroniza-

[2]The source code is available for free download at the URL `http://www.dis.uniroma1.it/~ hpdcs/AML-STM.zip`.

tion mechanisms are avoided at all within the added sample collection/aggregation modules ([3]). Actually, the master thread is randomly selected among the active threads at the beginning of each sampling interval. At the end of each sampling interval, the master thread calculates the aggregated statistics (including feature variance and correlation) and then exploits the NN instance to calculate (along its own critical path) the number *opt* of concurrent threads which is expected to maximize the throughput according the approach depicted in Section 4. Finally, it keeps active *opt* threads out of the maximum number of $max_{thread}$ threads during the next sampling interval.

In our implementation, the activation/deactivation of threads is based on a shared array with $max_{thread}$ elements. The master thread sets to 1 (0) the elements associated with the threads which have to be deactivated (activated). The slave threads (namely the remaining $max_{thread} - 1$ threads) check their corresponding value before executing a new transaction, by trapping into a polling phase while the value is 1. In this way, the application level software is never interrupted because of a thread deactivation, thus leading any transaction to safely run up to its commit/rollback point.

Finally, our implementation of the NN instances embedded within our thread-parallelism regulation architecture consists of acyclic feed-forward full connected networks [8] that have been coded by leveraging on FANN open source libraries (version 2.2.0) [38].

## 5.2 Evaluation Methodology

We evaluated the effectiveness of our ML-based thread-parallelism regulation architecture starting from its baseline configuration relying on fixed set of input features to the NN-based performance model. Then we focused on assessing the optimization based on dynamic feature selection.

For any STAMP application, we initially performed an off-line sampling process to build the training sets for all the NN instances we exploited with either fixed or dynamic set of input features. The samples used to populate the training sets have been collected in a set of runs where, for each run, we randomly selected the configuration of the input parameters within the parameter domains of each application. We refer to these configurations as *training configurations*. We identified the domain of each input parameter on the basis of the values suggested by the authors of STAMP (see [11] for details). For each run of an application, we randomly selected the number of concurrent threads between 1 and 16. We remark that the machine we used in our experimental study is equipped with 16 CPU-cores, and literature results show that it is generally not convenient to use more threads than the available CPU-cores in the underlying machine for running STM applications [39], which is the reason why we focused on the interval 1-16.

Once built the NN instances, we evaluated the ML-based thread-parallelism regulation architecture using various configurations of input parameters for each application. We refer to these configurations as *test configurations*. To assess the effectiveness of our proposal over a wide range of workload profiles, we also used test configurations whose values of the application input parameters correspond to extreme values of the domains of all the parameters. To clarify how we selected training and test configurations, in Figure 13 we show an example for the case of an application with three input parameters, which we name **a**, **b** and **c**. The cube we show represents the overall domain of these three parameters. Hence, the vertices of the cube represent the configurations defined by the extreme values of each input parameter's domain. As depicted in the figure, training configurations differ from test configurations, thus making our experimental study more robust. In particularly, the set of selected test configurations include the ones

[3]We recall that the study on sampling overhead in Section 4.3 was only aimed at assessing the additional cost spent along the critical path of threads while gathering samples, not for aggregating samples into a coherent stream.

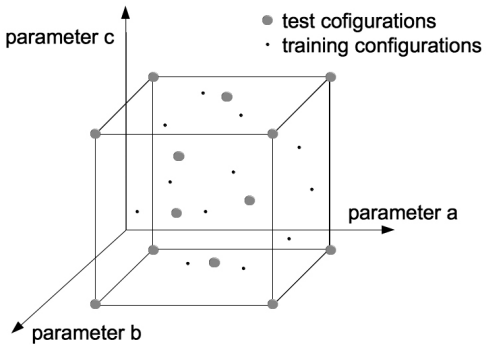


Figure 13: Example of the selection of training and test configurations for an application with three input parameters

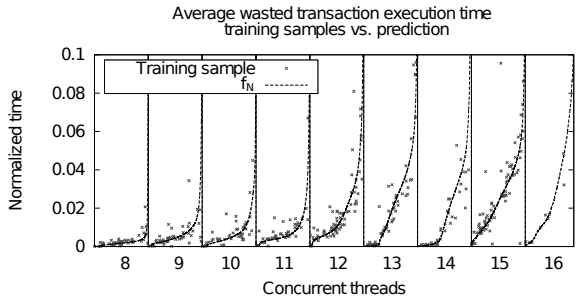


Figure 14: Average wasted transaction execution time: training set vs. predicted.

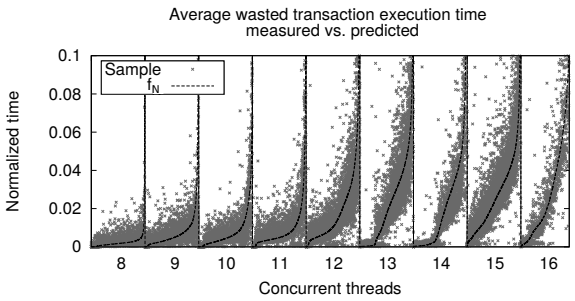


Figure 15: Average wasted transaction execution time: measured vs. predicted.

associated with the vertices of the cube, which represent border cases with respect to the configurations used to train the NN instances.

## 5.3 Details on the Training Process

For each STAMP application, we trained the associated NN instances using 800 samples randomly selected over a set of 10.000 samples that we collected during the execution of 64 runs of the application, carried out with different input configurations of the application, as explained above. Each sample averages the values collected over a sampling interval whose duration was determined by the execution of 4000 subsequent committed transactions along a thread. To limit the number of outliers, we filtered out samples stemming from sampling intervals in which more than the 99% of the transaction runs have been aborted. Note that, when moving towards such an abort probability, the transaction response time grows very fast and may exhibit very high variability. Reasonably, we can assume that when the transaction abort probability reaches 99% the throughput is never optimal, thus these samples can be straightforwardly discarded without affecting the reliability of the construction of the ML-based model at the core of our architecture.

To train the NN instances we used a back-propagation algorithm [34, 40, 41]. We observed that a number of hidden layers equal to one was a good thread-off between prediction accuracy and learning time. The number of hidden nodes for which the NN instances provided the best approximations was

between 4 and 16, depending on the application. Further, during our experiments, we observed that values between 0.0 and 0.2 were good for the learning coefficient and the momentum, respectively. The iterations of the back-propagation algorithm (across all the scenarios) have been no more than 2500, and the algorithm execution time was less than 10 seconds on a desktop machine equipped with an Intel®Core$^{TM}$2 Duo P8700 and 8 GB RAM. On the other hand, the on-line computation by the NN instances has been observed to take the order of a few microseconds on the target machine we used for the final performance tests.

In Figure 14 we provide some graphical details about the results of the off-line training process. The results refer to the Intruder application (from STAMP) and to the case of *maxSet* input features, which we use as a reference example. The data points represent the normalized wasted transaction execution time over the training samples collected by running the application with a number of threads varied in the interval between 8 and 16. In the same figure, the dotted line represents the wasted transaction execution time calculated by the NN instance (i.e the output values by $f_{NN}$), showing how it interpolates the values of the training samples. In Figure 15, we show data points representing the wasted transaction execution time over a larger set of samples that have not been used in the training process. As we can see, even for these samples, the trained NN instance shows good interpolation capability. Particularly, we note that with, e.g., 16 threads there are few samples in the training set (see Figure 14). In spite of this, Figure 15 shows that the trained NN instance well represents the average value of the samples that are out of the training set, even for the case of 16 threads.

To finally quantify the accuracy of the NN based performance model, we have measured the provided WRMS with the different NN instances across different configurations (e.g. number of threads used to run the applications) and we have consistently achieved values of the order of 5%. We recall that WRMS is a core parameter we exploit for dynamically shrinking/enlarging the feature set to be sampled as discussed in Section 4.3.1.

## 5.4 Experimental Results with Fixed Set of Input Features

In this section we present experimental results for assessing our ML-based thread-parallelism regulation architecture, referred to as ML-TPR, in the scenario with fixed set of input features (i.e. *maxSet*). We compare ML-TPR with the native implementation of TinySTM (which is devoid of scheduling support), and with three different scheduling techniques proposed in literature that have been integrated within TinySTM, thus sharing the same code base: (a) Adaptive Thread Scheduling [13], which we refer to as ATS-STM, (b) Shrink [17], we refer to as Shrink-STM, and (c) Probe-STM. ATS and Shrink have been already described when discussing related work (see Section 2). Probe-STM is an implementation of the hill climbing-based concurrency regulation approach exploited in [24, 26, 42] (see again Section 2 for indications on this approach). We note that in literature there not exist other techniques that use ML with the specific goal of regulating thread-level parallelism. In fact, the ML-based approach presented in [28] targets the orthogonal problem of mapping threads to different processors and CPU-cores. We selected the above techniques in our comparative analysis since they are representative of literature alternative approaches to ML (hence to our proposal) and target either thread-parallelism regulation or transaction scheduling, thus covering a relatively wide set of techniques for STM optimization. They have also been used as reference approaches in comparison studies on STM optimization (e.g. [24, 26]).

A for the configuration parameters of ATS and Shrink, we set their values according to the suggestions provided by the authors in [13] and [17], respectively.

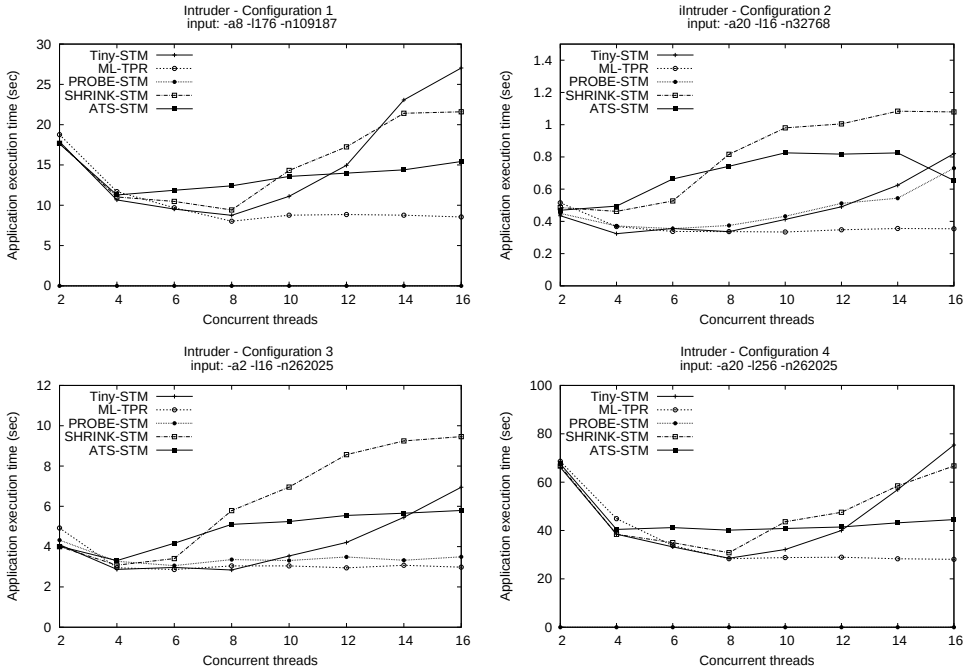


Figure 16: Execution time results with Intruder

In the experiments we varied the number of threads allowed to run the application, between 2 and 16, the latter value corresponding to the number of physical CPU-cores in the underlying architecture. Since the native version of TinySTM does not use any scheduling policy (possibly blocking threads), while ML-TPR and the other scheduling techniques can temporarily block threads, for these techniques the number of threads reported on the x-axis is an upper bound of the number of threads that actually run along the application execution (say $max_{threads}$ for ML-TPR).

For each application, we report results achieved with four different configurations of the input parameters, including two configurations corresponding to the vertices of the cube (see Section 5.2) where ML-TPR achieved the worst and the best performance with respect to TinySTM. Each reported throughput value is computed as the average over 20 runs. Although not explicitly reported for readability of the plots, we observed that the 95% confidence interval across samples was within the 10% of the average throughput value.

### 5.4.1 Results

In Figure 16, we present the results for Intruder. In all the configurations, the execution time with TinySTM decreases while increasing the number of concurrent threads up to 4-6. However, beyond these parallelism levels, it drastically increases. Conversely, ML-TPR avoids such a drastic performance loss. In more details, when the number of concurrent threads is less than the optimum value for TinySTM (e.g. 8 threads for *Configuration 1*), ML-TPR achieves similar performance to TinySTM (or slightly worse, just due to the overhead for sampling the application at run-time). At worst, namely with *Configuration 3* and 2 concurrent threads, the execution time with ML-TPR is 18% higher than the one of TinySTM. However, as soon as the level of allowed thread concurrency increases, the performance provided by TinySTM constantly degrades, while ML-TPR gives rise to execution times that are always close to the

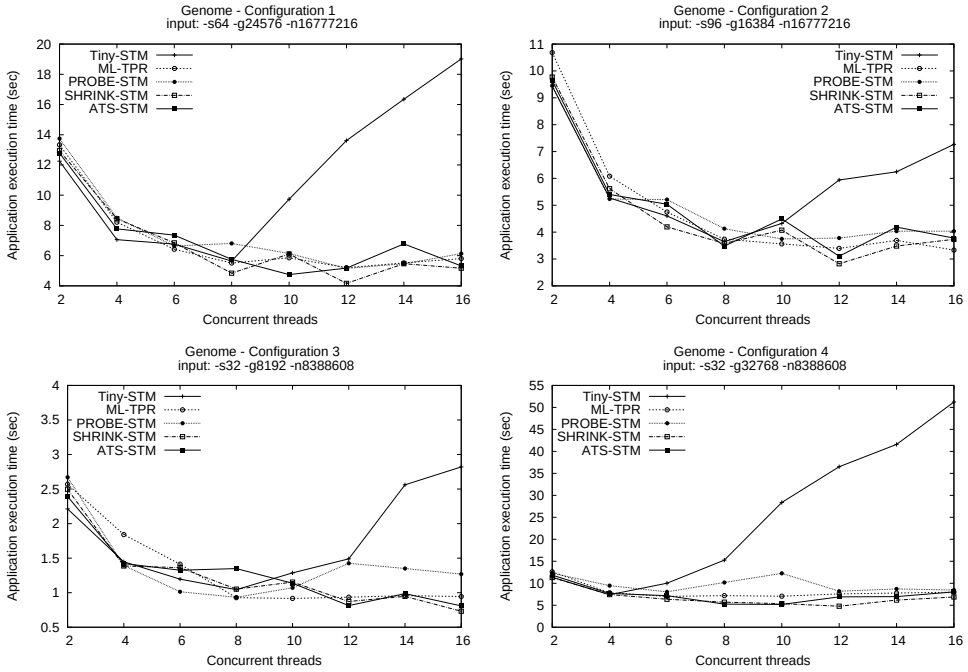


Figure 17: Execution time results with Genome

minimum value (i.e. the peak performance) achieved by TinySTM. Further, ML-TPR provides very good performance also in the two configurations of the application input corresponding to the vertices of the cube, say *Configuration 3* and *Configuration 4*. As mentioned, this is important in terms of robustness of our approach, because of the distance between these configurations and those that have been used for the training process of the NN at the core ML-TPR. In general, in all the configurations, as soon as there are 8 or more concurrent threads, the performance achievable by TinySTM rapidly degrades (up to a factor 2.8x). This phenomenon is avoided by ML-TPR, which prevents over-parallelism in all the scenarios where the maximum number of admitted threads exceeds the optimal level of parallelism for that specific workload profile. Compared to the other techniques, ML-TPR clearly performs better. Indeed, these techniques do not efficiently contrast the performance loss while the number of concurrent threads increases. Probe-STM performs similarly to ML-TPR with *Configuration 3*, but it is less efficient in the other scenarios.

Execution time results with Genome are shown in Figure 17. With *Configurations 1, 2* and *4* (the latter being a vertex configuration of the cube), the execution times with ML-TPR up to 4-6 threads are slightly worse than the ones achieved with TinySTM. At worst, i.e. for *Configuration 3* (which is the other vertex configuration of the cube), the difference is of the order of 19%. With more threads, while the performance with TinySTM degrades, ML-TPR again effectively contrasts the performance loss caused by transaction aborts and retries, and ensures an execution time comparable to, or better than, the minimum one provided by TinySTM. Overall, the overhead induced by ML-TPR pays off any time we enable the application to run with no under-parallelism, which is anyway a suboptimal configuration to be avoided. As for the other scheduling techniques, we observe that this time all of them scale quite well when enabling up to 16 concurrent threads, just like ML-TPR does.

In Figure 18 we show the results with Kmeans. Similarly to what happens with other benchmark

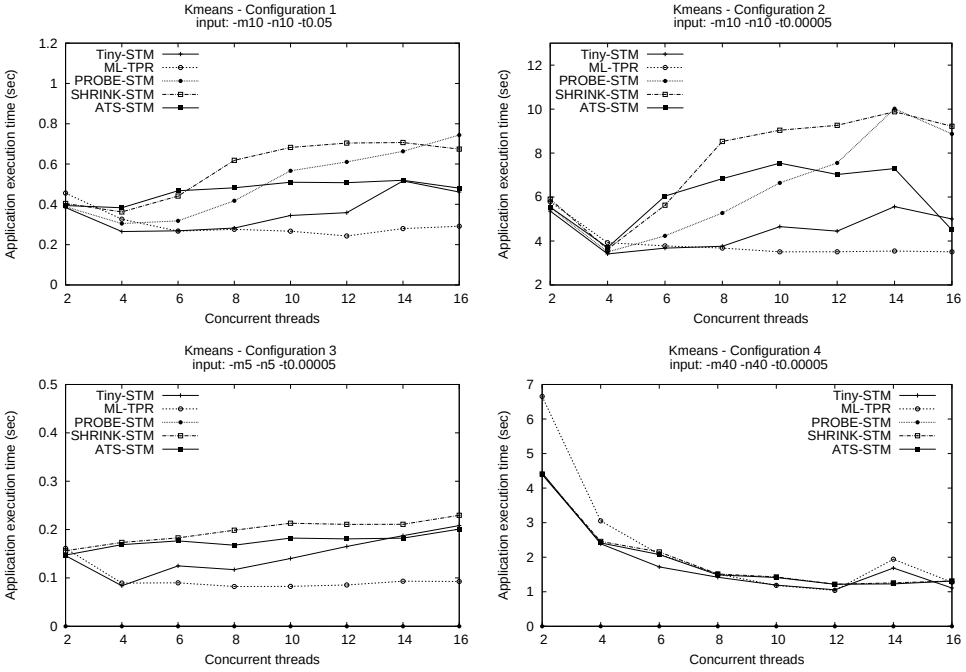


Figure 18: Execution time results with Kmeans

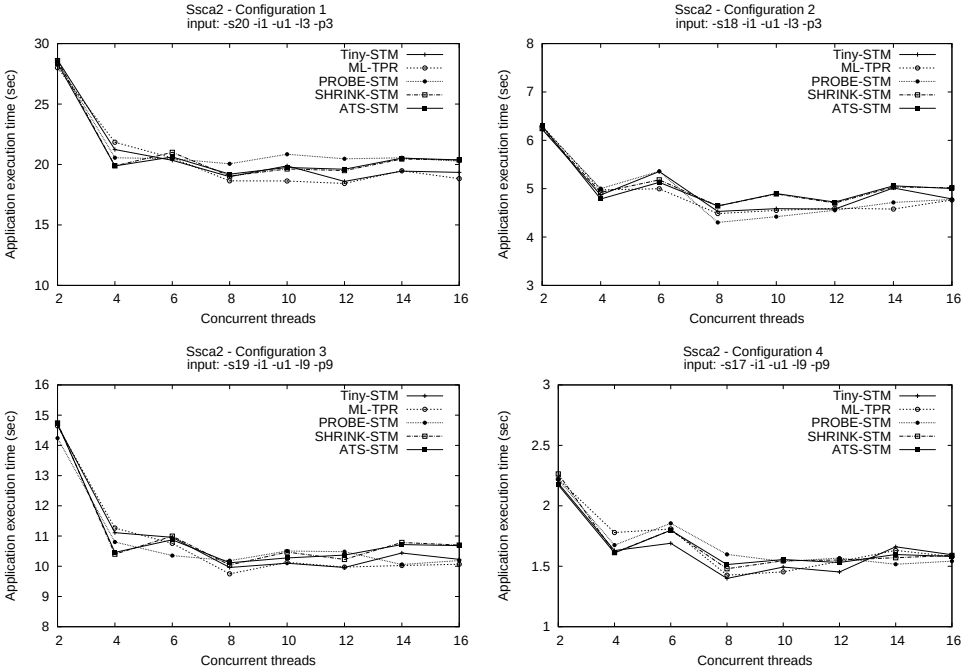


Figure 19: Execution time results with Ssca2

applications, ML-TPR ensures relevant performance benefits over all the other tested solutions (in all the configurations) when the number of enabled concurrent threads is greater than the value giving rise to the peak performance of TinySTM. The only exception is *Configuration 4*, namely a vertex configuration of the cube, where TinySTM and all the other techniques scale well up to 16 concurrent treads. With this configuration, ML-TPR shows a relevant overhead at lower thread counts. With 2 concurrent threads, the difference in the execution time is about 30% unfavorable to ML-TPR. However, as pointed out before, the overhead caused by ML-TPR robustly pays off at parallelism levels that are close to (or beyond) the optimal one, even for such border configuration of the input.

In Figure 19 we show the results with Ssca2. In all the tests we executed with this benchmark application, no performance degradation phenomenon has been noted while increasing the number of enabled threads up to 16. The execution time for this application with more than 6 concurrent threads is essentially flat, meaning that performance is mostly bounded by factors aside of data conflicts. Anyway, ML-TPR provides performance essentially similar to the one provided by the other techniques, showing how the overhead by ML-TPR steps down with execution profiles originating less intense accesses to shared data.

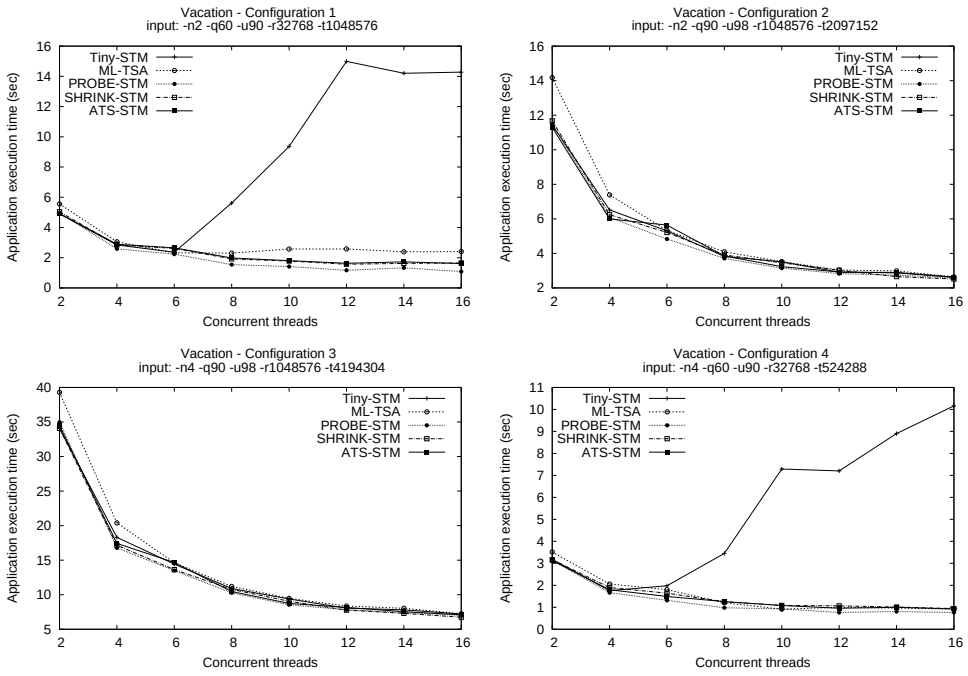


Figure 20: Execution time results with Vacation

The results with Vacation shown in Figure 20 are quite similar to the ones with Ssca2. The only difference is the limited scalability of the baseline TinySTM setting (entailing no scheduling support) with both *Configuration 1* and *Configuration 4*, where the execution time starts increasing after 4-6 enabled concurrent threads. Conversely, all the scheduling techniques scale well up to 16 threads, showing minimal differences in the execution times. Again, some overhead is manifested by ML-TPR in scenarios of under-parallelism, such as with 2 threads in *Configuration 2* and *Configuration 3*, an issue that again disappears as soon as we approach the optimal level of parallelism.

In Figure 21 we show the results with Labyrinth. This benchmark application clearly demonstrates the

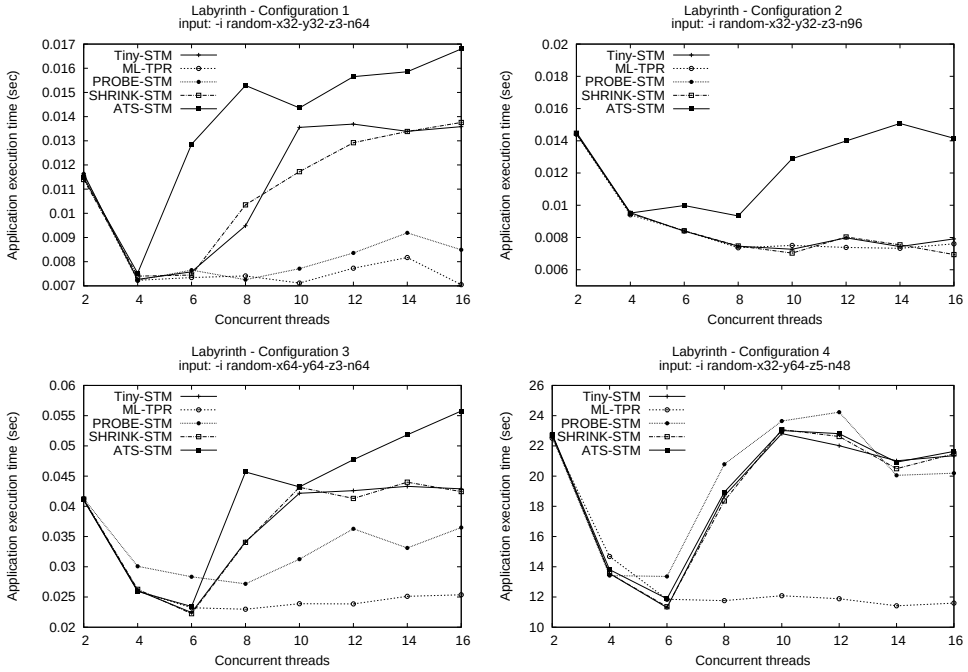


Figure 21: Execution time results with Labyrinth

limitations of the techniques we compare with ML-TPR. In fact, the workload generated by Labyrinth in 3 out of 4 configurations of its input parameters leads these techniques to fail in terms of actual reduction of the incidence of transaction aborts on performance. *Configuration 2* is the only one where the different solutions show similar performance, except ATS-STM, which gives rise to performance decrease as soon as the allowed number of threads steps over 4. However, we note that *Configuration 2* is such that scalability is guaranteed even by the baseline TinySTM setting, not employing any scheduling support for performance optimization. On the other hand, for the other configurations, leading to dynamics much more affected by data conflicts and consequent transaction aborts, ML-TPR is the only solution constantly delivering the peak performance. The data also show limitations of Shrink-STM, which performs better than TinySTM limited to *Configuration 1*. Probe-STM performs better than TinySTM for any number of concurrent threads limited to *Configuration 1*, and *Configuration 4* with more than 6 enabled concurrent threads.

Very similar considerations can be made for the case of Yada, whose results are shown in Figure 22. With this benchmark application, Probe-STM performs better than ML-TPR with reduced thread-parallelism, still due to the higher overhead by ML-TPR. However, just like the other literature solutions, Probe-STM does not allow resilience to performance degradation when running with amounts of threads leading to over-parallelism (say more than 8 threads).

The results with Bayes, which are shown in Figure 23, lead to additional interesting outcomes. In particular, this application shows very high variability of the execution time as soon as we allow executions with more than 4-6 concurrent threads. This is especially evident with *Configuration 1* and *Configuration 2*. However, in spite of this behaviour, demonstrating high variability of the incidence of data conflicts, ML-TPR effectively erases the effects of such variability and ensures the peak performance as soon as we allow executions with a number of threads sufficiently great to avoid under-parallelism. The only

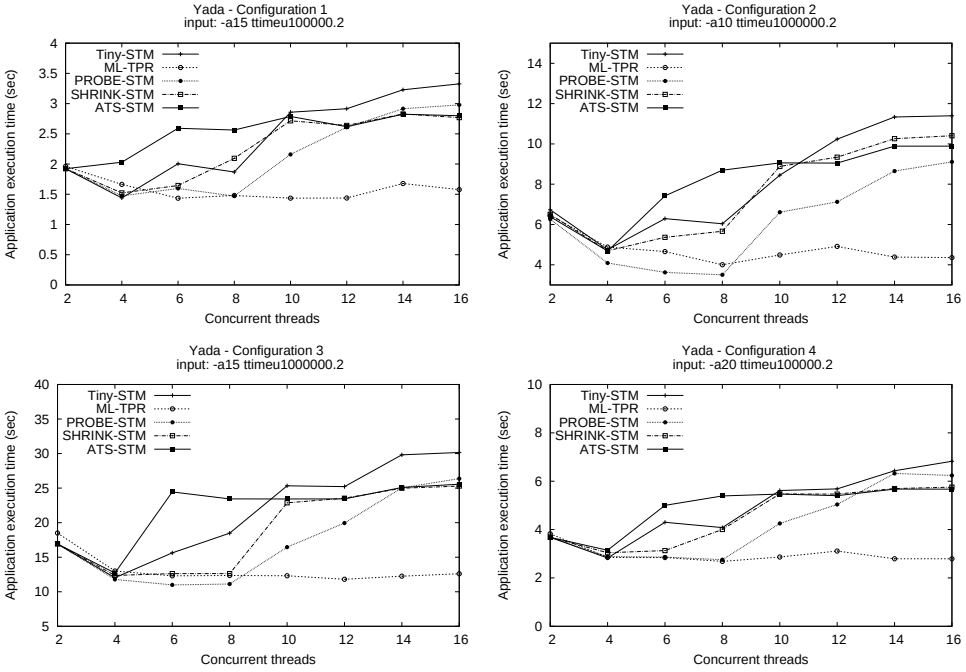


Figure 22: Execution time results with Yada

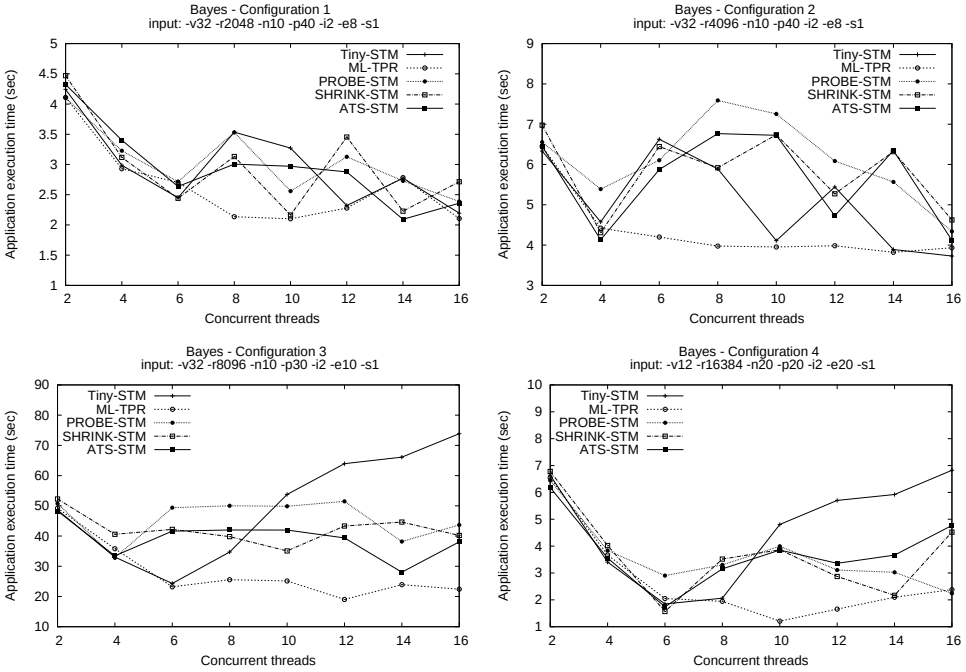


Figure 23: Execution time results with Bayes

exception is with *Configuration 1*, for which the performance trend provided by ML-TPR still looks not stable.

### 5.4.2 Results Assessment

By the experimental results we presented so far, we can draw the following main conclusions:

- ML-TPR is able to consistently avoid thrashing and performance degradation phenomena any time the applications are enabled to run with amounts of threads that stand beyond the optimal level of parallelism, as determined by varying the level of thread-parallelism in a baseline setting not including any scheduling support, like TinySTM. Also, except for rare configurations, the actual performance delivered by ML-TPR corresponds to the peak performance, still as determined via TinySTM while varying the number of threads. This is important also because the baseline TinySTM setting includes no overhead for running tasks that are functional to performance optimization. As shown by the experimental outcomes, the overhead in ML-TPR for handling these tasks plays a negative role mostly limited to scenarios of under-parallelism, which are typically corner cases when considering the scale up in the core count on modern platforms. In any case, such an issue is directly tackled by the dynamic feature selection approach that we evaluate in the next section.

  We also note that a static optimization approach where an application, once fixed its input configuration, is observed while running with different parallelism levels to identify the best one, could work well limited to that specific input configuration. In any case, such a static approach would result unsuited when the workload profile changes because of a different input configuration. Indeed, to cope with this problem, one would need some method to identify the different phases where the workload profile actually changes and to keep track of the most suited number of threads for each execution phase. This would require profiling the application for whichever admissible input configuration, which is clearly prohibitive. This problem is avoided by our approach.

- With almost all the tested benchmark applications and configurations, ML-TPR performs better than all the tested techniques based on lightweight heuristic approaches. The data show that the effectiveness of heuristic-based techniques may vary depending on the workload profile. Conversely, our ML-based approach shows a more stable behavior, in terms of ability to optimize performance by regulating the level of threads concurrency in face of differentiated workload profiles. One core aspect limiting those heuristics is that they aim at "guessing" the optimal scheduling decisions with no full exploitation of knowledge on the specific application profile. Hence they may lead anyhow to suboptimal outcomes. Further, for what concerns exploration-based approaches like Probe-STM, they may require non-minimal steps to converge to the optimal solution, or even fall in local throughput maxima. Contrarily, our thread-level parallelism regulator can quickly reacts to workload variations, given that the ML-based performance model can immediately identify the optimal number of threads once observed the current workload profile. Thus, it allows to directly "jump" to the optimal parallelism degree for the current phase of the workload profile. Further, Probe-STM needs to continuously perturb the current parallelism level to discover whether the application has changed its profile, thus requiring to be run with a different number of threads for a while. Hence, Probe-STM pays anyhow a cost even in scenarios of extremely stable workload. In conclusion, although heuristic techniques are quite simple to be implemented, they may provide uncertain results, as shown by our experimental analysis.

## 5.5 Experimental Results with Dynamic Feature Selection

This section is devoted to evaluate the additional benefits achievable via dynamic feature selection of the input features to the ML-based model at the core of our approach, as presented in Section 4.3. We name the architecture entailing dynamic feature selection capabilities as ML-TPR-DFS.

We evaluated the performance improvement by ML-TPR-DFS over ML-TPR by replicating all the experiments we presented in Section 5.4. The experimental outcomes have shown that all the benchmark applications with at least two couples of correlated features (see Table 1) take advantage by ML-TPR-DFS[4]. Indeed, we observed performance improvements for Intruder, Labyrinth, Vacation and Yada. Further, concerning benchmark applications with a single highly correlated couple of input features, ML-TPR-DFS provided performance improvement for Ssca2. For brevity, we only report the results for these benchmark applications, with the configuration of the input parameters that gave rise to the worst case performance of ML-TPR, i.e. the one for which ML-TPR performed worse than TinySTM with various levels of thread-parallelism due to its overhead (see Section 5.4). For all the other STAMP applications, i.e. Genome, Kmeans and Bayes, the execution times with ML-TPR-DFS and ML-TPR were essentially the same, thus we omit reporting them.

By the plots we report in Figure 24 we see how ML-TPR-DFS allows removing most of the overhead that led ML-TPR to perform worse that TinySTM in a few configurations. Indeed, the plots show that the execution times with ML-TPR-DFS at low concurrency are clearly lower than the ones achieved with ML-TPR. In particular, they are similar to (or lower than) the execution times observed with TinySTM. Additionally, for all the benchmark applications appearing in this test-set, with the exception of Vacation, ML-TPR-DFS reduces the execution times also at higher concurrency levels (say up to 16 concurrent threads) which points to the achievement of a better balance between overhead and effectiveness by concurrency regulation, on a wider scale of enabled parallelism levels.

As an additional relevant point, in our experimental study we noted that all the applications of the STAMP suite are characterized by quite static or phase-based workload profiles, with (very) few changes of the workload during their execution. However, one major strength of ML-TPR-DFS lies in its potential for detecting dynamic changes of the application profile and adapting the ML-based performance model (and its weight) consequently. On the other hand, more variable workloads would represent suited test cases to stress such capabilities, e.g., by inducing frequent and/or continuous shrinking/enlarging of the set of selected input features to the ML-based performance model.

To cope with this aspect we generated a highly dynamic workload by modifying Vacation. Essentially, this application emulates a travel reservation system, where customers can reserve flights, rooms and cars. The fraction of transactions accessing each one of the three types of items is fixed over time. This is representative of scenarios where the popularity of the different types of items does not change over time. We modified this feature in order to emulate scenarios where the item popularity can periodically (and rapidly) change. We note that such variations in the workload profile are likely to happen in real e-booking or e-commerce systems. Indeed, the volume of accesses to specific sub-sets of items is typically non-static. Rather, it is likely to react to, e.g., new product launches or promotional sales. To simulate a highly dynamic workload, we modified Vacation in such a way that the fraction of transactions accessing car items changes over time according to the curve depicted in Figure 25, and the remaining fraction of transactions is equally split in two sets accessing flight items and room items, respectively.

To provide the readers with insights on the capabilities of ML-TPR-DFS in scenarios with highly

[4]We found that a value equal to 0.85 as correlation threshold (i.e. the minimum correlation value for a couple of input features such that one feature is discarded - see Section 4.3.2) was adequate for all applications of our experimental study.

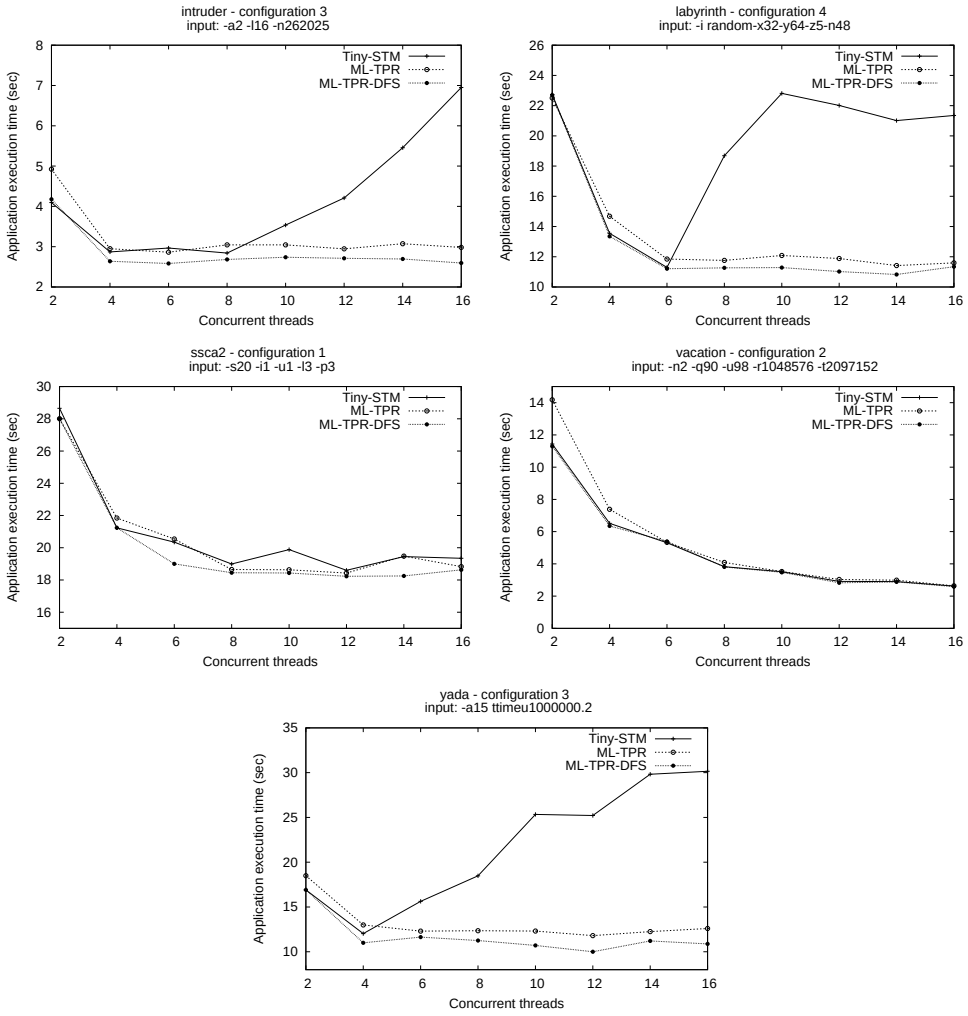


Figure 24: Execution time results with Dynamic Feature Selection

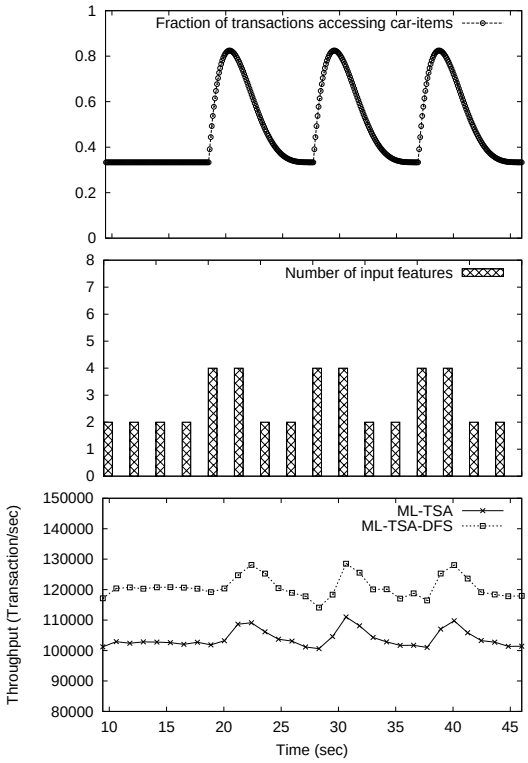


Figure 25: Parameters and throughput variation over time for the modified version of Vacation

variable workloads, we also show (see the middle plot in Figure 25) the variation of the number of input features to the ML-based model which are selected by ML-TPR-DFS in reaction to variability of the modified Vacation workload. The results refer to an execution with $max_{threads}$ set to the value 8. We note that, whenever the mix of transactions does not change over time (such as up to the 17th second of the execution, or between the 22th and the 27th seconds), only two features ($t_{time}$ and $ntc_{time}$) are selected as representative. Conversely, whenever the mix of transactions rapidly changes (e.g. in the interval between the 17th and the 22nd seconds or between the 27th and 32nd seconds), which leads to increase the variance and/or un-correlation of some input features, the number of features grows to 4 (including $t_{time}$, $ntc_{time}$, $ws_s$, $rs_s$). The throughput achieved with both ML-TPR and ML-TPR-DFS is shown in the bottom of Figure 25, and we see that ML-TPR-DFS achieves a remarkable performance improvement with respect to ML-TPR.

To compare the execution times of ML-TPR-DFS with both ML-TPR and TinySTM, we show the results of an experiment where we run the modified version of Vacations using *Configuration 1*, but with a larger number (about 3x) of transactions to induce more fluctuations of the workload. The achieved results are reported in Figure 26, which show that ML-TPR is able to avoid the performance loss of TinySTM observed with more than 6 enabled concurrent threads. However, with fewer threads the difference in performance with TinySTM is non-minimal. Further, with more than 6 concurrent threads, the execution times with ML-TPR do not equate the minimum execution time provided by TinySTM (i.e. the peak performance we observe with 6 concurrent threads). Conversely, ML-TPR-DFS ensures execution times equal to or significantly less than TinySTM for any level of enabled thread concurrency.

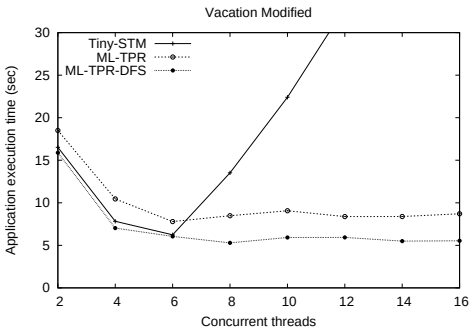


Figure 26: Execution time results with the modified version of Vacation

### 5.6 Additional Considerations on the Presented Approach

In our experimentation, we also explored the possibility to train a single NN by relying on synthetic workloads with a variety of transaction profiles, with the aim at using the trained NN with other applications, such as the ones from STAMP. This would in principle avoid the cost of application specific training. However, the achieved experimental results have shown that the throughput prediction error in this scenario is subject to large variations when moving from an application to another. Also, in some cases the error is subject to large fluctuations along the execution of a same application. This suggests that generating a kind of "meta-workload" to be exploited for training a single NN still able to capture all possible dependencies between the input and the output parameters for generic applications may be a very complex, if not unfeasible, process. Another aspect that makes the use of a single pre-trained NN problematic is related to the actual platform on top of which the final application to be controlled (in terms of thread-level parallelism) is deployed. When moving an application from a machine to another with a different (hardware) configuration, the optimal thread-level parallelism may change. This can be due to various factors, such as the presence of different cache levels, and their size, or the type of the memory architecture (UMA vs. NUMA). In conclusion, our study demonstrates that training a NN for each single application deployed on a specific machine is a reasonable and effective approach for thread-level parallelism regulation.

Clearly, the need for application specific training is the intrinsic price of our approach. On the other hand, we note that the collection of samples to build the training set could be carried out also when the application has already been activated in its production environment (if not possible in a pre-production phase). As an example, it could be carried out along the initial lifetime of the real application operations. This solution might benefit from incremental building of the training set as naturally provided by the spontaneous evolution of the workload profile along time.

## 6 Conclusions

In this article we have tackled the problem of optimizing thread-level parallelism in Software Transactional Memory applications. This has been done via the introduction of a machine learning-based performance prediction approach that enables the selection of the best suited number of threads to be employed for running the application along its lifetime, or along phases of its execution. Our proposal also explores the path of reducing the run-time overhead of the machine learning-based performance predictor, which

is done via the introduction of a technique that allows the dynamic shrinking/enlarging of the set of input features to be sampled at run-time in order to characterize the application execution profile. An ample experimental study is provided, based on data collected running all the applications from the STAMP benchmark suite on top of a 16-core machine. The study has been targeted at comparing the performance level provided by our solution, under the different workloads of the STAMP suite, against various alternative approaches taken from literature. As a further step ahead, it would be interesting to investigate solutions for on-line (incremental) training and instantiation of the neural networks at the core of our approach. In our present study we rely on an off-line training phase, where we identify in-advance to the actual operational phase of the applications the suitable values for some parameters, such as the number of hidden nodes of the neural networks. On-line training would require some methods to automatize the estimation of the suitable values of the above parameters, to be carried out at run-time. This approach will also need to include some method to determine when to automatically stop the incremental training phase. We plan to investigate this topic as future work along our research path.

# References


[1] Intel, INTEL Core™ Processors,
www.intel.com/content/www/us/en/processors/core/5th-gen-core-processor-family.html.

[2] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing, ACM, New York, NY, USA, 1995, pp. 204–213.

[3] D. Dice, O. Shalev, N. Shavit, Transactional Locking II, in: Proceedings of the 20th International Symposium on Distributed Computing, ACM, New York, NY, USA, 2006, pp. 194–208.

[4] P. Felber, C. Fetzer, T. Riegel, Dynamic performance tuning of word-based software transactional memory, in: Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, 2008, pp. 237–246.

[5] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, SIGARCH Comput. Archit. News 21 (2) (1993) 289–300.

[6] M. F. Spear, L. Dalessandro, V. J. Marathe, M. L. Scott, A comprehensive strategy for contention management in software transactional memory, in: Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, 2009, pp. 141–150.

[7] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, M. Olszewski, Anatomy of a scalable software transactional memory, in: Proceedings 4th ACM SIGPLAN Workshop on Transactional Computing, 2009.

[8] T. M. Mitchell, Machine Learning, McGraw-Hill, New York, 1997.

[9] L. Zhang, P. Suganthan, A survey of randomized algorithms for training neural networks, Inf. Sci. 364 (C) (2016) 146–155. doi:10.1016/j.ins.2016.01.039.
URL http://dx.doi.org/10.1016/j.ins.2016.01.039

[10] P. Baldi, K. Hornik, Learning in linear neural networks: a survey, IEEE Trans. Neural Networks 6 (1995) 837–858.

[11] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: Proceedings of the IEEE International Symposium on Workload Characterization, IEEE Computer Society, Seattle, WA, USA, 2008, pp. 35–46.

[12] N. Diegues, P. Romano, S. Garbatov, Seer: Probabilistic scheduling for hardware transactional memory, in: Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015, 2015, pp. 224–233.

[13] R. M. Yoo, H.-H. S. Lee, Adaptive transaction scheduling for transactional memory systems, in: Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures, ACM, New York, NY, USA, 2008, pp. 169–178.

[14] S. Dolev, D. Hendler, 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, ACM, New York, NY, USA, 2008, pp. 125–134.

[15] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, I. Watson, Advanced concurrency control for transactional memory using transaction commit rate, in: Proceedings of the 14th international Euro-Par Conference on Parallel Processing, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 719–728.

[16] H. Rito, J. Cachopo, ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory, Springer International Publishing, Cham, 2014, pp. 150–161.

[17] A. Dragojević, R. Guerraoui, A. V. Singh, V. Singh, Preventing versus curing: avoiding conflicts in transactional memories, in: Proceedings of the 28th ACM symposium on Principles of Distributed Computing, ACM, New York, NY, USA, 2009, pp. 7–16.

[18] P. di Sanzo, M. Sannicandro, B. Ciciani, F. Quaglia, Markov chain-based adaptive scheduling in software transactional memory, in: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, 2016, pp. 373–382.

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

[20] Z. He, B. Hong, Modeling the run-time behavior of transactional memory, in: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, Washington, DC, USA, 2010, pp. 307–315.

[21] A. Dragojević, R. Guerraoui, Predicting the scalability of an stm: A pragmatic approach, in: Presented at: 5th ACM SIGPLAN Workshop on Transactional Computing, 2010.

[22] P. di Sanzo, F. D. Re, D. Rughetti, B. Ciciani, F. Quaglia, Regulating concurrency in software transactional memory: An effective model-based approach, in: 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, Philadelphia, PA, USA, September 9-13, 2013, 2013, pp. 31–40.

[23] D. Rughetti, P. di Sanzo, B. Ciciani, F. Quaglia, Analytical/ml mixed approach for concurrency regulation in software transactional memory, in: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014, 2014, pp. 81–91.

[24] K. Chan, K. Tin Lam, 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.

[25] D. Didona, P. Felber, D. Harmanci, P. Romano, J. Schenker, Identifying the optimal level of parallelism in transactional memory applications, Computing 97 (9) (2015) 939–959.

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

[27] D. Rughetti, P. Romano, F. Quaglia, B. Ciciani, Automatic tuning of the parallelism degree in hardware transactional memory, in: Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings, 2014, pp. 475–486.

[28] Q. Wang, S. Kulkarni, J. V. Cavazos, M. Spear, Towards applying machine learning to adaptive transactional memory, in: Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing, 2011.

[29] M. Castro, L. F. W. Góes, and J.-F. Méhaut, Adaptive thread mapping strategies for transactional memory applications, J. Parallel Distrib. Comput., vol. 74, no. 9, pp. 2845–2859.

[30] D. Didona, N. Diegues, A. Kermarrec, R. Guerraoui, R. Neves, P. Romano, Proteustm: Abstraction meets performance in transactional memory, in: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016, 2016, pp. 757–771.

[31] G. Cybenko, Approximation by superpositions of a sigmoidal function, Mathematics of Control, Signals, and Systems 2 (1989) 303–314.

[32] K. Funahashi, On the approximate realization of continuous mappings by neural networks, Neural Netw. 2 (3) (1989) 183–192.

[33] D. Hush, B. Horne, Progress in supervised neural networks, Signal Processing Magazine, IEEE 10 (1993) 8 – 39.

[34] A. Bryson, Y. Ho, Applied Optimal Control: Optimization, Estimation, and Control, Halsted Press book', Taylor & Francis, 1975.

[35] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, G. Muller, Scheduling support for transactional memory contention management, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010, 2010, pp. 79–90.

[36] I. Valença, T. Lucas, T. Ludermir, M. Valença, Selecting variables with search algorithms and neural networks to improve the process of time series forecasting, Int. J. Hybrid Intell. Syst. 8 (3) (2011) 129–141.

[37] D. R. Jefferson, Virtual time, ACM Trans. Program. Lang. Syst. 7 (3) (1985) 404–425.

[38] Fast artificial neural network. `http://leenissen.dk/fann/wp`

[39] R. Ennals, Software transactional memory should not be obstruction-free, Tech. rep., Intel Research Cambridge Tech Report (Jan 2006).

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

[41] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, in: J. A. Anderson, E. Rosenfeld (Eds.), Neurocomputing: foundations of research, MIT Press, Cambridge, MA, USA, 1988, pp. 696–699.

[42] D. Didona, P. Felber, D. Harmanci, P. Romano, J. Schenker, Identifying the optimal level of parallelism in transactional memory systems, in: Proc. International Conference on Networked Systems, NETYS, Springer, 2013.