Online Tuning of Parallelism Degree in Parallel Nesting Transactional Memory

Jingna Zeng*†, Paolo Romano†, João Barreto†, Luís Rodrigues†, Seif Haridi*
*KTH/Royal Institute of Technology, Sweden
†INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—This paper addresses the problem of self-tuning the parallelism degree in Transactional Memory (TM) systems that support parallel nesting (PN-TM). This problem has been long investigated for TMs not supporting nesting, but, to the best of our knowledge, has never been studied in the context of PN-TMs. Indeed, the problem complexity is inherently exacerbated in PN-TMs, since these require to identify the optimal parallelism degree not only for top-level transactions but also for nested sub-transactions. The increase of the problem dimensionality raises new challenges (e.g., increase of the search space, and proneness to suffer from local maxima), which are unsatisfactorily addressed by self-tuning solutions conceived for flat nesting TMs.**

We tackle these challenges by proposing AUTOPN, an online self-tuning system that combines model-driven learning techniques with localized search heuristics in order to pursue a twofold goal: i) enhance convergence speed by identifying the most promising region of the search space via model-driven techniques, while ii) increasing robustness against modeling errors, via a final local search phase aimed at refining the model's prediction. We further address the problem of tuning the duration of the monitoring windows used to collect feedback on the system's performance, by introducing novel, domain-specific, mechanisms aimed to strike an optimal trade-off between latency and accuracy of the self-tuning process.

We integrated AUTOPN with a state of the art PN-TM (JVSTM) and evaluated it via an extensive experimental study. The results of this study highlight that AUTOPN can achieve gains of up to 45× in terms of increased accuracy and 4× faster convergence speed, when compared with several on-line optimization techniques (gradient descent, simulated annealing and genetic algorithm), some of which were already successfully used in the context of flat nesting TMs.

I. Introduction

Transactional Memory (TM) has emerged as an attractive paradigm to tackle one of the key sources of the complexity of parallel programming, i.e., designing fine-grained locking schemes aimed to ensure correct and scalable synchronization among concurrent threads. By exploiting the familiar abstraction of atomic transactions as a first-class synchronization construct, TM only requires programmers to identify *which* portions of code are to be executed atomically. This allows for encapsulating the complexity of *how* to ensure atomicity, by relying on system libraries or middleware layers that conceal efficient concurrency control schemes, possibly implemented via dedicated hardware supports [1].

Whether based on pure software implementations (STM) or on hybrid designs that combine best-effort hardware support for TM with software fall-back paths, several works have shown that TM can deliver competitive performance with complex, ad-hoc designed fine-grained lock schemes in a wide range of workloads [2]–[4]. However, a particularly challenging scenario for TMs is when workloads contain a large portion of long-running transactions. Indeed, long-running transactions suffer from long windows of vulnerability (i.e., the interval of time during which the transaction can be subject to conflicts with other transactions and abort), which makes them prone to prohibitively high abort rates.

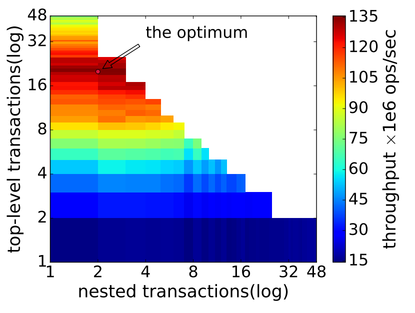
A possible path to tackle the performance issues caused by long transactions is to reduce the number of concurrent transactions, while dividing these fewer transactions into smaller, nested sub-transactions to run in parallel in the available idle cores — thus, reducing the execution time/vulnerability window of the original top-level transactions. This approach has received significant attention in the literature, with a number of proposals aimed to extend TM implementations to support parallel nesting with low overheads [5]–[10].

Unfortunately, though, fully exploiting the potential of PN-STM requires tackling a nontrivial problem that did not arise in TMs not supporting parallel nesting: identifying the right balance between inter-transaction and intra-transaction parallelism. Specifically, given a set n of available cores, in a PN-STM it is necessary to decide how many root (or top-level) transactions are allowed to be simultaneously active (t), and how many should be allocated to child (or nested) transactions (c) within each top-level transaction.

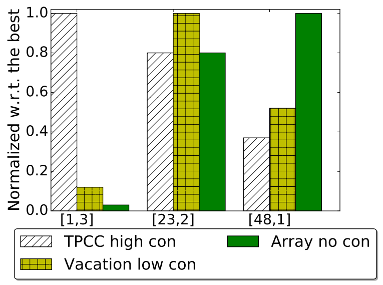
In CPU-intensive applications, as it is typically the case for TM environments, one should choose t and c such that the system is not oversubscribed, (i.e., $t \times c \leq n$). In platforms where t and c can be sufficiently high, there will be many possible configurations that avoid over-subscription: the search space grows in fact quadratically with respect to the problem of tuning the parallelism degree in TMs not supporting parallel nesting, which has been subject of intense study in the literature [11]–[15].

Identifying the optimal configuration in the resulting bi-dimensional search space is far from trivial. Among the different configurations, it is necessary to decide whether to favor higher root parallelism while reducing nested parallelism, or vice-versa. This depends on complex, workload-dependent interferences between root and child transactions that lead to contention and, therefore, conflicts; as well as on the overheads of spawning and maintaining nested transactions. Further, in contention-prone workloads, it is also not guaranteed that the optimal solution is the one that maximizes core utilization.

Fig. 1a illustrates this optimization problem by presenting the throughput of a porting of the TPC-C benchmark executing



(a) Throughput of a TPC-C porting.



(b) No-one-size-fits-all config exists.

Fig. 1. Performance of different configurations of inter- and intra-transaction concurrency in a PN-STM.

on a PN-STM environment (JVSTM [5]), when varying t and c. For this particular workload, configuration (20,2), i.e., 20 top-level transactions, each with 2 nested transactions, will generate the best throughput, where the total number of cores in the system is 48. Furthermore, the throughput of the best configuration is 9× higher than the throughput of the worst, which is configuration (1,1), and 2× to 3× higher than that of most of the remaining configurations. Also, the best configuration for a given workload may be the worst for a different application, as illustrated in Fig. 1b. Additionally, see Section VII-A, the static configuration that performs best on average across multiple workloads can be up to 3× away from optimum in some cases.

This is, to the best of our knowledge, the first paper to address the problem of self-tuning the parallelism degree in PN-TM and propose a novel self-tuning mechanism, called AUTOPN (Automatic Parallel-Nesting). Due to the large search space of parallel-nested configurations, the key feature of AUTOPN is its ability to quickly prune regions in the search space of that are unlikely to contain useful configurations. As we show later, AUTOPN dramatically outperforms popular search heuristics (like gradient descent, simulated annealing, or genetic algorithms [14], [16]–[18]), which, in contrast to AUTOPN, tend to get trapped in local optima or avoid local optima at the cost of a large number of random searches.

In contrast, AUTOPN employs a novel combination of model-driven and local-search techniques to find optimal configurations by only searching a small subset of the search space, accurately and reactively. Since AUTOPN requires no initial off-line training, it spares the cost of having to bootstrap an initial knowledge base representative of the target application and architectural environment.

Overall, this paper makes the following contributions:

1. For an effective pruning of the search space of parallel-nested configurations, AUTOPN leverages an innovative design of an online learning approach tailored to PN-TM by combining model-driven and local-search techniques. In our approach, the exploration is driven by a regression model based on the lightweight M5P decision tree algorithm [19], which uses the feedback gathered during previous explorations. The output of this regression model drives a Sequential Model-based Bayesian Optimization process [20], which exploits the Expected Improvement (EI) theory in order to: (1) identify the most promising configurations to explore and (2) determine when to stop exploring (stopping criterion). Finally, the model-driven exploration phase is complemented by a refinement phase, using a local search based on a simple hill-climbing strategy.

2. Since the above mechanism is highly sensitive to a proper tuning of the duration of the feedback-monitoring windows, we propose a novel, PN-TM-specific, adaptive sampling heuristics to address this key configuration issue. Our heuristics aim at an optimal trade-off between measurement accuracy, reactiveness and convergence speed.

3. We integrated AUTOPN with JVSTM [5], [21], a state of the art Java library implementing a lock-free multi-version PN-STM. This allowed us to experimentally evaluate the proposed solution using both synthetic and standard benchmarks (Vacation of the STAMP benchmark suite [22] and a porting of the TPC-C benchmark) and compared AUTOPN versus five different general purpose online self-tuning approaches. On average, AUTOPN reaches stability 4× faster than its counterparts and converges to solutions that are, on average, less than 1% away from optimum.

II. Related Work

A number of proposals [11], [15], [23] regarding self-tuning in the context of TM focus on the parallelism degree, i.e. the number of top-level concurrent threads, but, to the best of our knowledge, they all consider TM systems that do not support parallel nesting. Some of these solutions are based on analytical white box models that can be used to automate the decision. The work by Di Sanzo et al. [23], for instance, relies on an analytical model to predict the performance of various STM algorithms. Castro et al. [24] also proposed an analytical model of TM systems but focused on capturing the performance dynamics of hardware based implementations. These modeling tools could be used to estimate the performance of STM applications when using different numbers of top-level threads. However, they fall short in the context of PN-TMs, where it is crucial to identify the right balance between inter-transaction and intra-transaction parallelism.

Alternatively, other authors have proposed solutions based on black-box approaches, which resort to offline-trained machine learning techniques, such as artificial neural networks [25]. Rughetti et al. use machine-learning methods to adjust concurrency level in STM [13] as well as HTM [11]. In a follow-up work, these pure black-box techniques were combined with an analytical model to reduce the training time [12]. These works do not target PN-TM systems either. Additionally, since these solutions require offline training, they pose some practical challenges as they need to be pre-deployed on the target architecture in order to gather a training set that can be considered representative of the possible application dynamics on the target platform. We avoid this issue by relying purely on online learning techniques.

Some works propose solutions based on online learning approaches or customized heuristics. Didona et al. [14] use hill climbing to tune the concurrency level in STM. F2C2-STM adjusts the parallelism degree according to the target performance being profiled [26]. However, none of the above-mentioned proposals target PN-TMs. Nonetheless, we do evaluate in Section VII several online learning strategies. Some of these techniques have been successfully employed in TMs not supporting parallel nesting; our work highlights their limitations when employed in PN-TM systems.

Auto-PN builds on recent advances in the area of Sequential Model-based Bayesian Optimization (SMBO), which, in its turn, finds its roots in the statistics literature on experimental design [27] and has been recently extended to cope with the optimization of complex software systems, including databases [28], SAT solvers [20] and big data frameworks [29]. Our work makes novel contributions in this area by specializing SMBO to the domain of PN-TM systems by i) investigating the use of carefully selected, domain-specific set of initial points; ii) proposing the use of a modeling technique (bagged decision-trees, each trained over only 2 dimensions) that is lightweight enough to be used at run-time with limited overhead, iii) introducing domain-specific techniques to seek a trade-off between monitoring accuracy and system's reactivity; iv) combining SMBO and hill climbing, whereas the two approaches are normally considered as alternative.

Our work is also related to the broader literature in the area of self-tuning for TM, which has investigated the dynamic adaptation of orthogonal problems. These include: tuning the number of locks internally maintained by a STM implementation [30]; identifying the optimal retry strategy in presence of different abort types in HTM systems [16]; defining the thread mapping policy in STM systems deployed on multi-socket/NUMA machines [31]. There are also proposals for adapting between different synchronization systems, e.g., switching between HTM, STM and lock-based [32], or between different TM implementations [3], [33].

III. Problem Definition

In this section, we first present some background information on PN-STM systems and introduce the abstract model of a PN-STM system that is considered by AUTOPN. Using this model, we propose a mathematical formalization of the problem of self-tuning the parallelism degree in PN-TMs.

A. PN-TM: Background and System Model

Let us start by introducing some base terminology related to the nested transaction model [34]: A transaction is either top-level, which is managed as a conventional non-nested transaction, or is nested within a (parent) top-level transaction. This nesting generates trees of transactions, which are denoted using family relationship terminology, such as parent, child, ancestor, descendant, sibling, etc. In this model, only transactions with no active children can access data; such accesses are either reads or writes to memory.

In the TM domain, various nesting models have been considered, defining different admissible behaviors regarding the concurrent execution of nested transactions [6]. A first distinction can be seen between flat (or linear) and parallel nesting, where the former supports only a single thread of execution in a transactional nest (i.e., it disallows concurrency within a transaction) and the latter allows actual concurrency among a parent and its children transactions. Another distinction regards the semantics associated with the commits of a nested transaction, which can be made applied at the "top-level" (i.e., made visible to other nests) either immediately (open nesting model) or only upon commit of it top-level transaction (closed nesting model).

The self-tuning scheme presented in this work is based on a black-box approach and, therefore, makes no assumptions on the commit semantics of nested transactions. As a matter of fact, AUTOPN is currently integrated with a PN-STM supporting the closed nesting model (JVSTM [5]), but it could be seamless applied also to open nested PN-STMs. Indeed, we consider an abstract model of a PN-STM, which assumes the following policy regarding how threads are allocated to top-level and nested transactions: top-level transactions are executed by a first set of threads, T; child transactions (of any family) share a disjoint set of threads P. Further, we assume that the PN-STM can alter the cardinality of sets T and P at run-time. Finally, we assume that the PN-STM system can be queried to obtain the measurement of some target key performance indicator (e.g., throughput) achieved in the current system's configuration.

This abstract model maps to various possible implementations, although, in typical PN-STM systems [5], [8], top-level transactions are often executed directly by application-level threads, whereas child transactions are executed by a shared thread pool that is under the direct control of the PN-STM run-time (in order to minimize the overheads associated with the activation of child transactions).

B. Problem Formulation

In general, in a PN-TM program, each of top-level transaction may generate arbitrary deep trees of nested transactions, where each nested transaction may spawn a different number of child transactions. The shape of a transaction tree can also be dependent on the transaction type and on its inputs.

In the light of these considerations, one may be tempted to define the problem of tuning the degree of parallelism in PN-STM systems as aimed to identify the optimal (performance-wise) number of concurrent child transactions that should be executed for each parent transaction, at any depth of the tree, and for each different transaction type and input class. Although one may be allured by the generality of such a formulation, it is also easy to see that the corresponding problem's dimensionality is theoretically unbounded, and may thus make the problem intractable in practice.

In this paper, we take a pragmatical approach, which leverages on practical considerations to reduce the problem's dimensionality and allow its tractability in realistic settings.

Observation 1: Transaction trees are normally shallow. The first observation that we make is that the PN-TM workloads considered so far in the literature [5], [8], [9] tend to generate very shallow, although potentially quite fat, trees. We argue that this is due to the fact that, in order to actually benefit from the use of parallel nesting, the tasks to be parallelized have to be sufficiently coarse in order to outweigh the overheads incurred for synchronizing the parallel execution child transactions. However, as we descend at greater depths in the tree, the task granularity tends to decrease exponentially, and, as such, also the probability of being able to effectively parallelize them.

Observation 2: Oversubscription should be avoided. A second pragmatical consideration is that in CPU-intensive workloads, such as those typically targeted by PN-TM, it is pointless to oversubscribe the available physical cores (or hardware threads, if simultaneous multi-threading is supported), i.e., it is desirable to ensure that the number of concurrent threads never exceeds the number n of physical cores.

Observation 3: Allocation of resources to top-level transactions should be fair. Finally, in most of the workloads that we are aware of, both in the flat and parallel nesting TM model [22], [35], the top-level transactions are executed with the same probability by any of the top-level threads in T. Hence, it is in general desirable to ensure a fair allocation of resources to top-level threads, i.e., each top-level thread should have the possibility to activate the same number of concurrent nested transactions. Achieving this goal, while still avoiding oversubscription, implies that each top-level thread can execute concurrently at most $n/|T|$ nested transactions.

Overall, the above considerations lead us to propose a more pragmatic problem formulation that considers a two dimensional search space $S = \{t \times c : t, c \in \mathbb{Z}^+ \wedge t \times c \leq n\}$, where t denotes the number of concurrent top-level transactions, c the number of concurrent nested transactions in each transaction tree, and n the total number of cores in the system. Denoting with $f : S \to \mathbb{R}$ the unknown function that maps the space of admissible configurations to their performance (quantified via some target Key Performance Indicator, i.e., KPI), the problem consists in identifying the configuration $opt \in S$ that optimizes f, i.e., maximizes or minimizes it depending on the target KPI. In the following, we will consider the problem of maximizing throughput (committed transactions per second), as target KPI.

IV. Architectural Overview of autoPN

Figure 2 illustrates the high-level architecture of autoPN. autoPN has been integrated with JVSTM [5], a Java-based multi-versioned PN-STM supporting closed nesting. JVSTM was extended with three new modules, namely, the *optimizer*, the *actuator*, and the *monitor*.

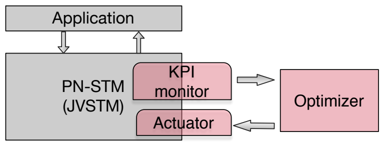


Fig. 2. High level architecture of autoPN

The *optimizer*, described in Section V, aims to identify the optimal degree of parallelism for a given application workload. This component drives the exploration of the solution space by gathering feedback on the quality of the explored configurations via the KPI monitor, and requesting adjustments of the current degree of parallelism to the actuator. We based the implementation of the optimizer on Weka, an open source machine learning workbench in Java [36].

The *actuator*, described in Section VI, is the module in charge of steering the dynamic reconfiguration process of the degree of parallelism of JVSTM-based applications.

The *monitor*, described in Section VI, is responsible for gathering on-line measurements of relevant key performance indicators (KPIs) of the PN-STM system. As already mentioned, in the following we will focus on the problem of maximizing throughput, although autoPN could be used to optimize different metrics (e.g., latency or abort rate). The key challenge in the design of this component consists in gathering measurements in a both accurate and timely way, so as to maximize not only the accuracy but also the reactivity of the self-tuning process.

V. Optimizer

Figure 3 illustrates the self-tuning process that is coordinated by the optimizer module. The optimizer leverages two complementary methodological approaches: i) an initial model-driven exploration phase, which relies on the Sequential Model-based Bayesian Optimization (SMBO) framework [37] and on decision tree-based regression models [19]; ii) a final localized search based on hill-climbing. The rationale underlying this approach is to take the best of the two approaches (i and ii) in order to compensate for each other's drawbacks.

By steering the initial exploration using model-driven techniques, autoPN aims at building a global view of the unknown function, f, that maps the configuration space (S) to system's performance (e.g., throughput). This approach allows to quickly discard large regions of the search space that are unlikely to contain high-quality solutions. Model-based techniques, however, are inherently approximated (being based on approximate models built based on partial knowledge of S) and, although they are normally effective in identifying at broad strokes the quality of macro-regions of the search space, they tend to suffer from *long-sightedness*, i.e., they have low accuracy in predicting the quality of solutions in nearby search regions. This makes model-based techniques prone to quickly identify solutions that are in the proximity of global maxima, but fail to detect higher quality solutions in their proximity.

Local search heuristics, conversely, can suffer from *short-sightedness*, in the sense that they get trapped easily in local

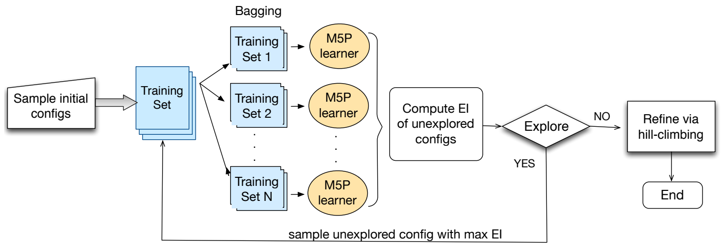


Fig. 3. Illustrating the self-tuning process coordinated by the optimizer.

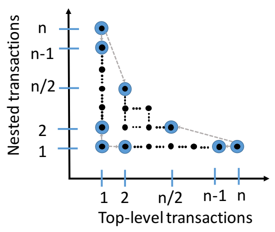


Fig. 4. Biased sampling strategy of the initially explored configurations.

maxima, and, as we will show in Section VII, tend to exhibit poor accuracy when used alone. Conversely, in AUTOPN, we use a localized search, based on hill-climbing, only after having concluded the model-driven optimization phase. This way, we activate the local search starting from a high-quality configuration, which we strive to further refine via a final hill-climbing phase that addresses long-sightedness of the model.

A. *Initial sampling*

Any black-box model-driven approach requires a collection of initial samples of the configuration space aimed to construct an initial knowledge base/training set to build the model. The most common approach to determine the initial training set consists in using a randomized, uniform sampling policy. The key advantage of this approach is its simplicity and generality.

In AUTOPN, we depart from this conventional design and exploit domain knowledge to define a biased sampling strategy that aims at promoting the construction of models able to capture the global trends of f based on a small number of configurations. The key intuition is to force the deterministic exploration of 9 configurations that lie on the three boundary regions of S illustrated in Figure 4. The rationale underlying this strategy is to assess the workload's sensitivity to small variations of the inter-/intra-transaction parallelism in proximity of three "pivot" configurations, which correspond to three extreme settings of inter-/intra-transaction concurrency, namely: i) using only a thread to execute top-level transactions and disabling parallel nesting, i.e., (1,1); ii) allocating all hardware resources to top-level transactions and disabling parallel nesting, i.e., (n,1), iii) running top level transactions sequentially and allocating available cores to execute nested transactions concurrently, i.e., (1,n).

In Sec. VII we will show that this biased, domain-specific, sampling strategy allows for building initial training sets of higher quality than generic approaches based on uniform random sampling.

B. *SMBO-driven Optimization*

SMBO is a strategy for optimizing an unknown function $f : D \rightarrow \mathbb{R}$, whose estimation can only be obtained through the observation of sampled values. It operates as follows: (i) evaluate the target function f at n initial points $x_1 \dots x_n$ and create a training set S with the resulting $\langle x_i, f(x_i)\rangle$ pairs; (ii) fit a probabilistic model M over S; (iii) use an *acquisition function* $a(M, S) \rightarrow D$ to determine the next point x_m; $iv)$ evaluate the function at x_m and accordingly update M; $v)$ repeat steps (ii) to $iv)$ until a stopping criterion is satisfied.

Acquisition function. SMBO can be coupled with different acquisition functions, and we use Expected Improvement (EI) [37].

EI selects the next point to sample based on the predicted gain with respect to the currently known optimal configuration, while keeping into account the possible uncertainty in that prediction. More formally, considering without loss of generality a maximization problem, let D_e be the set of evaluation points collected so far, D_u the set of possible points to evaluate in D and $x_{max} = \arg\max_{x \in D_e} f(x)$. Then the positive improvement function I over $f(x_{max})$ associated with sampling a point x is $I(x) = max\{f(x) - f(x_{max}), 0\}$. Since f has not been evaluated on x, $I(x)$ is not known *a priori*; however, one can use the model M, trained over past observations, to predict the expected value for the positive improvement:

$$EI(x) = \mathbb{E}[I(x)] = \int_{f(x_{max})}^{\infty} (c - f_{x_{max}}) p_M(c|x) dc$$

where $p_M(c|x)$ denotes the probability density function that the model M associates with possible outcomes of the evaluation of f at point x [37]. This formulation associates a high EI value either with points that are predicted by the model to have a high mean value or with points for which the model is uncertain about (i.e., they have high predicted variance).

By using the EI as acquisition function for SMBO, one achieves the effect of balancing exploitation and exploration: on the one hand, the model's confidence is exploited to sample the function at points that are likely to correspond to maxima; on the other hand, by exploring regions for which the model is uncertain, one provides the model with valuable information and iteratively shrink uncertainty zones.

Computing $\mathbf{p}_M(\mathbf{c}|\mathbf{x})$. Like in other previous works that applied SMBO [37], in order to ensure tractability, we assume $\mathrm{p}_M(\mathrm{c}|\mathrm{x})$ to have a Gaussian distribution $\sim N(\mu_x, \sigma_x^2)$. This allows computing EI in closed form:

$$E[I(x)] = (\mu_x - f_{max})\Phi(\frac{\mu_x - f_{max}}{\sigma_x}) + \sigma\phi(\frac{\mu_x - f_{max}}{\sigma_x}) \quad (1)$$

where Φ and ϕ represent, respectively, the probability density function and cumulative distribution function of a standard Normal distribution.

In order to estimate μ_x, σ_x^2 in Eq. 1, AUTOPN relies on an ensemble (i.e., a set) of black-box regressors, based on the M5P algorithm [19] (described next). More in detail, the AUTOPN builds a *bagging* ensemble [37] of k M5P-based learners, each trained with a random subset (obtained via uniform sampling with replacement) of the whole training set. μ_x and σ_x^2 are computed, respectively, as the average and variance of the predictions of the ensemble of learners. We use a set of 10 bagged learners in AUTOPN, which we found to be sufficiently large to generate sufficient model diversity while incurring negligible overheads (§ VII-E).

Model construction. As mentioned, AUTOPN relies on M5P-based models to predict the performance of unexplored configurations. The M5P algorithm allows for constructing decision-tree based regressors that maintain on their leaves a multivariate linear model, allowing for approximating arbitrary functions by means of piece-wise linear models.

One may include a broad range of different metrics (or *features*) in the training set that is fed to the M5P regressor (e.g., abort rate, reads vs. writes ratio, etc). The larger the number of metrics, the larger the potential to provide the model with additional information and enhance its predictive power; larger feature spaces, though, require exponentially larger data sets (the, so-called, curse of dimensionality problem [37]) and accordingly larger computational costs.

In AUTOPN we address this trade-off by using a minimalistic feature space defined solely over $t \times c$, i.e., the training set fed to M5P is composed of tuples $\langle t, c\rangle \rightarrow KPI(t, c)$. This choice is based on the following rationale:
1. In AUTOPN, we want to be able to use models based on the knowledge of a very small, online explored, number of configurations. Furthermore, the SMBO phase can still be corrected by the final hill-climbing-based search. These two observations allow us to use simpler models, which can be trained with fewer data and at a reduced computation cost, in the SMBO phase.
2. In AUTOPN, we wish to update (re-train) the ensemble of M5P learners, and accordingly query them to obtain, in an online fashion, updated predictions of unknown configurations; ideally, this should occur whenever new samples become available. Decision tree algorithms are known for their relatively high computational efficiency (compared, e.g., to popular regressors like ANN [25]). Yet, in order to minimize the costs related to training/querying online the models, it is clearly desirable to use simple models.

As a consequence of this design decision, AUTOPN builds a different model for each workload, and it does not attempt to detect similarities with previously optimized workloads (unlike other self-tuning schemes do [11]). Besides reducing model complexity and instrumentation overheads, this choice has the key pragmatical advantage of allowing AUTOPN to operate fully online, which avoids the complexity and cost of gathering, offline, a training-set containing workloads representative of the target application and platform.

Stopping criterion. The EI framework provides us with a natural framework also to construct a model-driven criterion to stop exploration and settle with the best configuration encountered so far. More in detail, AUTOPN concludes the SMBO optimization phase as soon as the EI falls below a threshold (typical values are 1%-10%). We evaluate the effectiveness of this policy in § VII-C.

Dynamic workloads. The presented design assumes that, throughout the optimization process, the application's workload does not vary, so that the KPIs of the explored configurations can be compared in a meaningful way. We argue that this assumption is realistic, since, as we will show in Section VII, AUTOPN normally requires exploring only a very small number of configurations. Indeed, this assumption holds true for all the benchmarks we considered in Section VII, which are representative of realistic PN-TM applications.

Yet, AUTOPN can easily be extended to cope with dynamically shifting workloads (again, under the assumption that the workloads vary slow enough to ensure convergence of the optimization process), by coupling it with a change detector (e.g., based on the CUSUM algorithm [38]). This would allow for identifying statistically relevant alteration of the workload characteristics (e.g., sudden throughput changes) and, accordingly, activate a new self-tuning process.

VI. KPI MONITOR AND ACTUATOR

KPI Monitor. As mentioned, the key challenge addressed by the KPI monitor is to strike a trade-off between accurate and responsive measurements. In order to achieve good measurement's accuracy, the general practice in the TM self-tuning domain is to use conservative measurements intervals, statically defined on the basis of either a fixed time period or having collected a given number of relevant events, i.e., commits of top-level transactions. None of these solutions are robust and generic, though. Policies based on monitoring windows of fixed time, e.g., [26], require a careful tuning that is strongly workload-dependent. For instance, the throughput of different TM applications can easily vary by 6 or more orders of magnitudes, from a few to millions of committed transactions per second. Using conservative values, in order to ensure good accuracy as with low-throughput workloads, can severely hinder the reactivity of the whole self-tuning process with high-throughput workloads, especially if the applications being optimized last for relatively short periods. The use of aggressive values may, conversely, lead to feeding the optimizer with erroneous information, and hinder its predictive capabilities.

Policies based on gathering a fixed number of commits, e.g., [14], conversely, are vulnerable when the system adopts a "bad" (performance-wise) configuration, e.g., where transactions starve due to excessively high contention levels, failing to commit or committing at a very slow rate. We shall illustrate the limitations of these static policies in Section VII-D.

In AUTOPN we address this problem by using an adaptive policy that can automatically adjust the monitoring frequency in a robust and workload independent way. This adaptive policy is based on two complementary mechanisms.

The first mechanism is based on the idea estimating the statistical uncertainty associated with the current throughput measurement on the basis of the coefficient of variation (CV). More precisely, we evaluate throughput upon each commit event since the beginning of the monitoring window (time $t = 0$). Denoting as $time(i)$ the time elapsed since the beginning of the measurement window and the occurrence of the i-th commit, the throughput upon the i-th commit, $\mathcal{T}(i)$ is simply $i/time(i)$. We then use as an estimate of the accuracy of the measurement after i commits the CV of $\mathcal{T}(i)$, i.e., $CV(\mathcal{T}(i)) = \frac{std\_dev(\{\mathcal{T}_1,\ldots,\mathcal{T}_i\})}{avg(\{\mathcal{T}_1,\ldots,\mathcal{T}_i\})}$. Typical CV values used in engineering to express high confidence span in the range [1%,10%]. As we will see in Section VII-D, 10% represents a robust value in the context of PN-TM systems.

The second mechanism is based on an adaptive time-out mechanism and is aimed at avoiding that the monitoring system gets stuck for an arbitrarily long time in a "bad" configuration. The intuition here is to use the throughput of the (1,1) configuration, corresponding to a single-threaded/sequential configuration, as a way to automatically establish a (workload-dependent) threshold that we then use to decide whether to time out the monitoring interval. Denoting with $\mathcal{T}(1,1)$ the throughput of the (1,1) configuration (which, recall, is always included in the initially sampled configurations), we can estimate the average time to experience a commit event in the (1,1) configuration as $1/\mathcal{T}(1,1)$. Waiting longer than such a time interval without witnessing any commit event suggests that the current one is likely to be a low-quality configuration — especially considering that PN-TM workloads are expected to scale, so the throughput in the (1,1) configuration is typically much lower than in the optimal configuration. Given that the quality of this configuration is very likely to be very far away from optimum, we argue that it is indeed pointless, for self-tuning purposes, to spend the further time to achieve high accuracy in its measurement.

Based on these domain-specific insights, we use $1/\mathcal{T}(1,1)$ as conservative time-out value, after which we terminate the measuring interval, even if the CV-based estimator cannot confirm the measurement's stability, yet.

Actuator. The actuator relies on two complementary mechanisms to dynamically adapt the parallelism degree of a PN-STM application. On the one hand, we seek to achieve total transparency for applications (e.g., legacy ones), in order to allow their optimization without requiring any modification to their source-code. This is achieved by intercepting the calls to begin and commit/abort transactions (both top-level and nested) and ensuring, via the use of semaphores, that the number of concurrent top-level transactions/nested transactions per tree is at any point in time less than allowed by the current configuration. On the other hand, we allow applications to take advantage of the knowledge on their optimal degree of inter-/intra-concurrency by exposing this information via an ad-hoc API. This information can be used, for instance, to optimize the different data partitioning schemes.

VII. Experimental Study

In this section, we evaluate experimentally AUTOPN, seeking answers to the following key questions: i) how does the optimization process employed by AUTOPN fare compared with the alternative, online learning approaches (Section VII-B)?; ii) how relevant are the domain-specific optimizations integrated in the design of AUTOPN? In particular, how effective is the proposed biased sampling strategy for the initial configurations (Section VII-C) and the adaptive policy used by the KPI monitor (Section VII-D)?; iii) what is the overhead caused by AUTOPN's self-tuning process and to what extent does it interfere with the performance of TM applications (Section VII-E)?

All the experimental data was gathered by deploying AUTOPN on a machine equipped with four AMD Opteron 6168 processors (48 cores total), 128GB of RAM, Linux 2.6.32 and an OpenJDK 64-bit Server 1.7.0 (build 19.0-b03) JVM.

A. Benchmarks and baseline algorithms.

In the following, we will consider 10 workloads, generated via 3 different benchmarks, which are representative of different application domains and have heterogeneous characteristics. In particular, we considered two well-known benchmarks, TPC-C and Vacation of the STAMP benchmark's suite [22], which were adopted, by previous works, to operate in a PN-STM environment [5]. We use these benchmarks to generate 3 workloads each, characterized by low/medium/high degrees of contention. We further developed a synthetic micro-benchmarks, called Array, in which we use nested transactions to parallelize the access of top-level transactions to a large, shared array of integers. We use Array to generate 4 workloads, in which transactions scan the entire array and change respectively none, 0.01%, 50% and 90% of the array's elements. It should be noted that the best average configuration over all the workloads (i.e., 24 top level and 2 nested transactions) has an average Distance From Optimum of 21.8%, its 90-th percentile is 2.56× worse than optimum and, in the worst case (Array high contention) 3.22× slower.

We consider the following five baseline algorithms: i) random search, which selects a configuration uniformly at random; ii) grid search, which explores the bi-dimensional search space by progressively sweeping first c (child transactions) and then t (top-level transactions); iii) a plain hill-climbing (HC) algorithm that starts from a randomly selected point; iv) Simulated Annealing (SA) [17], a probabilistic algorithm inspired by the thermodynamics of metals, which extends the hill-climbing algorithm by forcing it to pick a sub-optimal neighbour with a probability that decays over time, analogously to how electrons' speeds decay over time in a metal after an initial heating; v) Genetic Algorithms (GA), a family of search heuristics inspired by biological evolution in nature. GA encodes candidate solutions (i.e., configurations in our case) via bit strings, denoted as *chromosomes*. Solutions are selected via an *evolution process*, which identifies the best solutions (called elites) in the current generation, and probabilistically applies mutations (random flips of chromosomes' bits) and crossovers (swapping segments of the chromosome's between elites).

SA and, in particular, GA, require tuning a relatively large number of meta-parameters in order to be properly used: the cooling rate and initial temperature, for SA, the population size, chromosome encoding function, the rate of elitism,

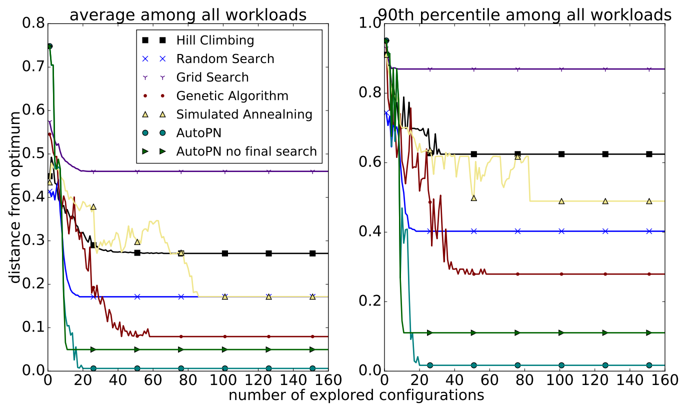


Fig. 5. Evaluating the effectiveness of AUTOPN's optimization scheme.

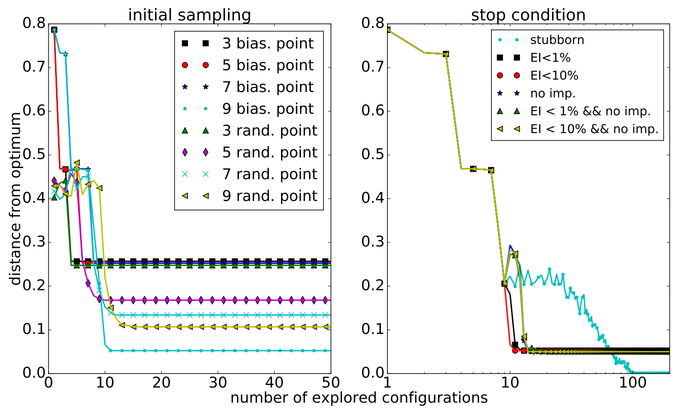


Fig. 6. Initial sampling and stop condition in AUTOPN.

crossover, and mutation, among others, for GA. In order to ensure the proper configuration of these meta-parameters, we used 10-fold cross-validation combined with grid-search to compare, offline, the performance of these methods when using different settings of these meta-parameters and identify their most robust parametrization across the whole set of workloads.

For the random and grid search heuristics, we stop exploration when the last 5 explorations do not improve more than 10%. This is to provide a fair comparison with AUTOPN when using EI less than 10% as a stopping criterion.

B. Comparison with the baselines

Figure 5 reports the accuracy, evaluated in terms of distance from optimum (expressed in %), over time, of the baselines mentioned above, AUTOPN and a variant of AUTOPN that skips the final, hill-climbing-based local search.

In this experiment, we feed the optimizers with off-line collected traces, obtained by evaluating exhaustively every configuration in the solution space (which encompasses 198 configurations, given that we use as target system a machine equipped with 48 cores). Each configuration was tested 10 times, using runs lasting 10 minutes. This choice allows us to decouple the problem of obtaining timely, but accurate, KPI measurements, which we will address in Section VII-D, from that of building effective optimization policies, which we can compare using fair and reproducible inputs.

The plots in Figure 5 report the average (left) and 90-th percentile of the distance from optimum across all the workloads. To account for the non-determinism of the optimization process, we repeat each workload 10 times for all the optimization algorithms. The plots clearly highlight that purely local search heuristics, like HC, fall easily in local optima when faced with PN-STM workloads, and are even worse than simple random search. This result is relevant, considering that hill-climbing has been used with success in the past to optimize STM systems that do not support parallel nesting. The use of random deviations from the local gradient, used by SA, only mildly ameliorates the problem. Among the considered baselines, GA is definitely the best performing one, converging eventually at around 8% from global optimum. Yet, we observe that GA is also quite disappointing in terms of convergence speed, exploring on average around 30% of the whole search space before stabilizing. We argue that, compared to HC and SA, GA promotes a broader search in the solution space, which eventually pays off, but that is also "data greedy", probably due to its preponderant random nature.

These experimental results clearly highlight the superiority of AUTOPN's optimizer vs all the baseline solutions, in terms of both convergence speed and quality of the final configuration: on average, AUTOPN achieves 1% distance from optimum, while exploring 3× fewer configurations than the best baseline (GA).

Finally, this plot allows us to quantify the accuracy gains stemming from the use of the final refinement phase: with as few as a handful additional explorations, the final localized search reduces the distance from optimum from 5% to 1% on average, and from 10% to 2% on the 90-th percentile; a remarkable increase, in relative terms.

C. Initial sampling strategy and stop condition

We now focus on assessing the efficacy of the mechanisms employed by AUTOPN's optimizer to i) build the initial knowledge base for the SMBO-driven optimization, and ii) establish the completion of the SMBO process.

We start by comparing the accuracy achievable by the SMBO phase when using as initial sampling policy either i) 3, 5, 7 and 9 configurations selected uniformly at random, or ii) the proposed biased scheme, but selecting a smaller number of configurations, namely 3, 5, 7[1]. We disable the hill-climbing based search phase, to focus solely on the SMBO phase, and use AUTOPN's default stopping policy (EI<10%).

The *Initial Sampling* in Figure 6 shows two main trends: i) at parity of explored configurations, the biased sampling policy achieves, on average, better accuracy than a uniform random sampling scheme but only when it includes all the 9 points at the boundary of the solution space; ii) there is a major

[1]When using 3 configurations, we only include the three pivots, i.e., {(1,1),(n,1),(1,n)}. With 5 configurations, we also include {(n-1,1),(1,n-1)}, and with 7 also {(2,1),(1,2)}

boost in accuracy when passing from 7 to 9 configurations in the biased sampling policy. Overall, these results confirm the effectiveness of AUTOPN's biased sampling scheme, and suggest that renouncing to sample any of the 9 boundary configurations tends to make it much less effective.

In the *Stop Condition* of Figure 6, we evaluate the stop condition logic employed by AUTOPN, when using different threshold values for the EI (1% and 10%), a heuristic (*no-improvement*) that stops exploration if the observed performance has not improved over the last K steps (we use K=5 as an illustration), hybrid heuristics resulting from the combination of EI and no-improvement, as well as an stopping condition (*stubborn*) that blocks exploration only when the optimal configuration has been found. The stubborn condition is, in fact, an ideal stop condition that cannot be implemented in practice, since the optimum is not known a priori.

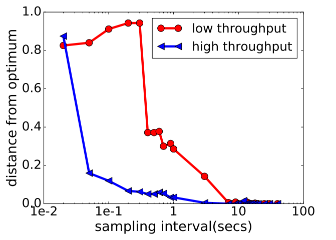
The analysis of the results for stubborn reveals an interesting fact: it is much more effective to complete the SMBO phase as soon as we have identified good enough solutions, which is what EI does, than striving to achieve perfect accuracy, like stubborn does. This suggests that model-based techniques, due to their inherently approximate nature, are effective in identifying approximate solutions, but tend to blunder when they are forced to operate at a resolution that is beyond their actual reach. Local search techniques are much more efficient at this, as seen in Figure 5. The plot also confirms the superiority of the EI-based stopping condition versus the simpler no-improvement heuristic as well as versus the more sophisticated hybrid schemes.

D. KPI monitoring

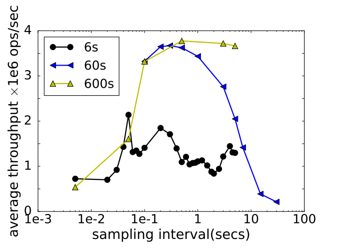
We now move to evaluate the KPI monitoring technique used by AUTOPN. We start by considering two experiments that highlight the relevance and complexity of the problem.

In Figure 7a we consider a live deployment of AUTOPN, in which we execute two workloads of the Array micro benchmark, generating, respectively, low and high throughput rates. We consider a simple strategy that uses a monitoring window of a statically configured duration, which we vary on the x-axis across three orders of magnitude (from 20 msecs to 40 secs), giving AUTOPN enough time to fully complete its optimization process and report the distance from the optimum of the final solution it identified. The plot clearly shows that different workloads require different tunings: values as low as 0.1 seconds can be used for the high-throughput workloads to achieve 10% accuracy, whereas 30× larger intervals must be used to achieve similar accuracy with the other workload.

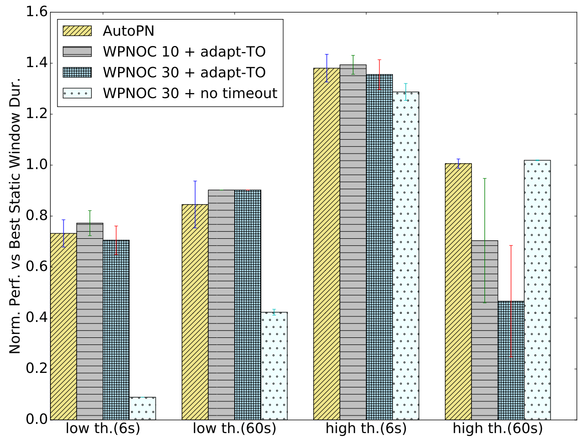
Figure 7b considers a more challenging, yet realistic scenario, of short-running applications: in this case, the faster is the KPI monitor in providing accurate feedback to the optimizer, the smaller the time spent exploring suboptimal configurations and the larger the average throughput for the run (reported on the y-axis). In this case, using overly conservative values can, as expectable, cripple performance severely, thus, amplifying the complexity of tuning the duration of the measurement window.



(a) Tuning of static time-based schemes: impact on the final solution.



(b) Tuning of static time-based schemes: impact on avg. throughput.



(c) Comparing with static schemes.

Fig. 7. Evaluating the effectiveness of AUTOPN's monitoring scheme.

Finally, in Figure 7c, we contrast the performance of the adaptive monitoring policy employed in AUTOPN with: i) two variants which, instead of using the CV-based policy to detect measurement's stability, wait to track, resp. 10 and 30, commits (WPNOC10, WPNOC30), but that use AUTOPN's adaptive timeout policy (adapt-TO); iii) a policy that only waits to track 30 commits (WPNOC30). On the x-axis, we vary the workloads and their duration; on the y-axis we report the distance from the optimum of the configuration identified by AUTOPN, normalized w.r.t. to the best configuration identified using an (optimally tuned) monitoring scheme based on static measurements intervals. By the plot, we observe that AUTOPN's adaptive policy is, overall, the one to deliver the most consistent results across the considered workloads.

E. Overhead assessment

Finally, we quantify the overhead of the proposed optimization scheme. In order to isolate the self-tuning costs, we enable monitoring (using the proposed adaptive mechanism) and request the tuning algorithm to update and query its ensemble of models (based on trace-driven feedback). In the meanwhile, we inhibit the actuator from applying any configuration change. Thus, we pay the costs of self-tuning without benefiting from it. We consider an Array workload that generates no contention and scales up to all the available cores and configure the system to operate since the start in the optimal configuration. This way, we can estimate an upper bound on the overhead of AUTOPN's optimization scheme. Our experiments reported a negligible drop in throughput that is, on average, less than 2%, confirming the practicality of the proposed solution.

VIII. Conclusion and Future Work

This paper addresses, for the first time in the literature, the problem of optimizing the degree of inter- and intra-transaction parallelism in PN-TMs. We tackle this problem by proposing AUTOPN, an on-line self-tuning system that combines model-driven learning and localized search heuristics to achieve the best of the two approaches. We evaluate AUTOPN via an exhaustive experimental study, showing that AUTOPN reaches stability 4× faster than its counterparts, converging to solutions that are, on average, less than 1% away from optimum.

While AUTOPN focuses on homogeneous workloads, an interest research question that could be addressed by future work is how to tackle workloads comprised of heterogeneous types of parallel-nested transactions. Given its black box nature, we argue that would be relatively straightforward to extend AUTOPN to support this problem of higher-dimensionality, by modeling the search space as a set of distinct (t_k, c_k) pairs for each type of top-level transaction, k. It is unclear, though, whether its efficiency would still remain acceptable when faced with such a larger search space.

Another research line suggested by our work is how to incorporate information on the noisiness of sampled data (e.g., measured in terms of coefficient of variation) in the modeling phase. This is in contrast with the current AUTOPN's approach, in which data is fed to the model only after having ensured that the corresponding measurement is statistically meaningful (or irrelevant for the optimization's purposes).

Acknowledgments

This work was partially supported by Erasmus Mundus Joint Doctorate Program (2012-0030), Swedish Foundation for Strategic Research, and Portuguese funds through FCT via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014. We wish to acknowledge Diego Didona and José Simões for their help in the initial phase of this project, and Niklas Ekström for helpful discussions.

References

[1] R. Yoo, C. J Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *SC*, 2013.

[2] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, 1997.

[3] Y. Lev, M. Moir, and D. Nussbaum, "Phtm: Phased transactional memory," in *TRANSACT*, 2007.

[4] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, 2006.

[5] N. Diegues and J. Cachopo, "Practical parallel nesting for software transactional memory," in *Distributed Computing*. Springer, 2013.

[6] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: model and architecture sketches," *Science of Computer Programming*, 2006.

[7] J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka, "Leveraging parallel nesting in transactional memory," in *ACM Sigplan Notices*. ACM, 2010.

[8] H. Volos, A. Welc, A. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy, "Nepaltm: design and implementation of nested parallelism for transactional memory systems," in *ECOOP*, 2009.

[9] W. Baek and C. Kozyrakis, "NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory," in *PACT*, 2009.

[10] R. Kumar and K. Vidyasankar, "Hparstm: A hierarchy-based stm protocol for supporting nested parallelism," in *TRANSACT*, 2011.

[11] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, "Automatic tuning of the parallelism degree in hardware transactional memory." in *Euro-Par*, 2014.

[12] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Analytical/ml mixed approach for concurrency regulation in software transactional memory," in *CCGrid*, 2014.

[13] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *MASCOTS*, 2012.

[14] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," *Computing*, 2015.

[15] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Adaptive concurrency control for transactional memory," in *MULTIPROG*, 2008.

[16] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *ICAC*, 2014.

[17] C. R. Hwang, "Simulated annealing: theory and applications," *Acta Applicandae Mathematicae*, 1988.

[18] K. F. Man, K. S. Tang, and S. Kwong, "Genetic algorithms: concepts and applications," *IEEE transactions on Industrial Electronics*, 1996.

[19] Y. Wang and I. H. Witten, "Induction of model trees for predicting continuous classes," 1996.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration." *LION*, 2011.

[21] S. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *ACM SIGPLAN Notices*, 2011.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[23] P. D. Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano, "On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking," *Performance Evaluation*, 2012.

[24] D. Castro, P. Romano, D. Didona, and W. Zwaenepoel, "An analytical model of hardware transactional memory," in *MASCOTS*, 2017.

[25] T. M. Mitchell, *Machine Learning*, 1st ed. McGraw-Hill, 1997.

[26] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for stms," in *IPDPS*, 2014.

[27] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. of Global Optimization*, Dec. 1998.

[28] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proceedings of the VLDB Endowment*, 2009.

[29] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017.

[30] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP*, 2008.

[31] M. Castro, L. F. W. Góes, and J.-F. Méhaut, "Adaptive thread mapping strategies for transactional memory applications," *Journal of Parallel and Distributed Computing*, 2014.

[32] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, "Adaptive integration of hardware and software lock elision techniques," in *SPAA*, 2014.

[33] D. Didona, N. Diegues, A. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "Proteustm: Abstraction meets performance in transactional memory," in *ASPLOS*, 2016.

[34] J. E. B. Moss, "Nested transactions: An approach to reliable distributed computing." Tech. Rep., 1981.

[35] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: A benchmark for software transactional memory," in *EuroSys*, 2007.

[36] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD*, 2009.

[37] E. Brochu, V. M. Cora, and N. D. Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010.

[38] M. Basseville, I. V. Nikiforov, *Detection of abrupt changes: theory and application*. Prentice Hall Englewood Cliffs, 1993.