Multi-dimensional Self-tuning in Transactional Memory

Ricardo Neves

Abstract

The Transactional Memory (TM) paradigm promises to greatly simplify the development of concurrent applications. This led, over the years, to the creation of a plethora of TM implementations delivering wide ranges of performance across workloads. Yet, no universal TM implementation fits each and every workload. In fact, the best TM in a given workload can reveal to be disastrous for another one. This forces developers to face the complex task of tuning TM implementations, which significantly hampers the wide adoption of TMs.

This thesis addresses the challenge of automatically identifying the best TM implementation for a given workload. The proposed system, ProteusTM, hides behind the TM interface a large library of implementations. Under the hood, it leverages an innovative, multi-dimensional online optimization scheme, combining two popular machine learning techniques: Collaborative Filtering and Bayesian Optimization.

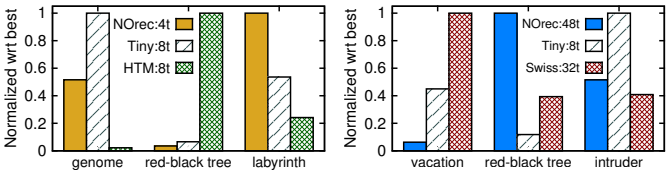
ProteusTM was extensively evaluated, obtaining average performance $<$ 3% from optimal, and gains up to 100$\times$ *over static alternatives.*

1. Introduction

The advent of multi-cores has brought parallel computing to the fore-front of software development. The Transactional Memory (TM) [25] abstraction is a prominent approach that promotes a simple idiom for synchronizing code: programmers specify only *what* should be done atomically (via serializable transactions), leaving to the TM the responsibility of implementing *how* to achieve it.

Over time, several works have provided evidence [42, 36, 40] on the effectiveness of TM to simplify the development and verification of concurrent programs. Recently, the relevance of TM was amplified by the standardization of constructs in popular languages (such as C/C++ [38]), and by the integration of hardware support in processors by Intel and IBM [50, 29].

The abstraction vs performance dilemma. Unfortunately, TM performance remains a controversial matter [7]: despite the large body of work in the area, the search for a "universal" TM with optimal performance across all workloads has been unsuccessful. Fig. 1 conveys experimental evidence of the strong sensitivity of TM to the workload characteristics. The energy efficiency (in Fig. 1a) and throughput (in Fig. 1b) of various TMs are reported in different architectures and benchmarks. The data is normalized with respect to the best performing configuration for the considered workload. Fig. 1 shows that, in two different architectures and metrics, the optimal TM configuration differs significantly for each workload. Furthermore, choosing wrong configurations can cripple performance by several orders of magnitude. Interestingly, some TMs used in these experiments were designed to tackle various workloads [21, 20], but configuring them properly is non-trivial and they still cannot perform well for all workloads.



(a) Throughput/Joule on a single-chip 8-core CPU (Machine A in Table 1).

(b) Throughput on a multi-chip 32-core CPU (Machine B in Table 1).

Figure 1: Performance heterogeneity in TM applications.

The problem is that the efficiency of existing TM implementations is strongly dependent on the workloads they face.

Given the vast TM design space, manually identifying optimal configurations is a daunting task. Overall, the complexity associated with tuning TM contradicts the motivation at its basis, i.e., to simplify the life of programmers, and represents a roadblock to the adoption of TM as a mainstream paradigm [32].

Contributions

The main contribution of this thesis is ProteusTM, the first TM with multi-dimensional self-tuning capabilities. ProteusTM contains two main modules:

• **PolyTM** is a polymorphic TM library that encapsulates state-of-the-art results from research in TM, and has the unique ability to transparently and dynamically adapt across multiple dimensions: (i) switch between different TM algorithms; (ii) reconfigure the internal parameters of a TM; (iii) adapt the number of threads concurrently generating transactions.

• **RecTM** is in charge of determining the optimal TM configuration for an application. Its basic idea is to cast the problem of identifying such best configuration as a recommendation problem [41]. This allows RecTM to inherit two highly desirable properties of state of the art Recommender System (RS) algorithms: the ability to operate with very sparse training data, and to require only the monitoring of the Key Performance Indicator (KPI) to be optimized. This avoids intrusive instrumentation [44] and (possibly inaccurate) static code analysis [49] employed by other machine learning-based solutions.

While building ProteusTM, several challenges were solved:

▶ *Transparency and portability*: PolyTM encapsulates a wide variety of TM implementations, along with their corresponding tuning procedures. The key challenge here is to conceal these mechanisms without breaking the clean and simple abstraction of TM. Furthermore, one of the key design goals

of ProteusTM is to seamlessly integrate with existing TM applications, and to support different machine architectures.

This issue was tackled by integrating PolyTM in GCC, via the standard TM ABI [38], and by exposing to programmers standard C++ TM constructs. Not only this preserves the simplicity of the TM interface, but it also maximizes portability due to the widespread availability of GCC across architectures.

► *Minimizing the cost of adaptivity.* Supporting reconfiguration across multiple dimensions requires introducing some degree of synchronization, in order to ensure correctness during run-time adaptations. The challenge here is to ensure that the overheads to support adaptivity are kept small enough not to compromise the gains achievable via self-tuning.

This challenge was addressed by designing lightweight synchronization schemes that exploit compiler-aided, and asymmetric code instrumentation. The combination of these techniques allows PolyTM to achieve a negligible overhead of around 1% and a maximum overhead of 8%, even when considering the most performance sensitive TM implementations.

► *Applying Recommender Systems to the TM domain*: Decades of research have established RS as a powerful tool to perform prediction in various domains (e.g., music and news) [35, 13, 11]. The application of RS techniques to performance prediction of TM applications, however, raises unique challenges, which were not addressed by previous RS-based approaches to the optimization of systems' performance [15, 14]. One key issue here is that, in conventional RS domains (e.g., recommendations of movies), users express their preferences on a homogeneous scale (e.g., 0 to 5 stars). On the contrary, the absolute value of key performance indicators (KPIs) of TM applications can span very heterogeneous scales.

To cope with this issue, a novel normalization technique has been introduced, called rating distillation, which maps heterogeneous KPI values to scale-homogeneous ratings. This allows ProteusTM to leverage state-of-the-art RS algorithms even in the presence of TM applications whose KPIs' scales span across different orders of magnitude.

► *Large search space*: Although RS algorithms are designed to work with very sparse information, their accuracy can be strongly affected by the choice of the configurations [45] that are initially sampled to characterize a TM application. Deciding *which* and *how many* TM configurations to sample is a challenging task, as ProteusTM supports reconfiguration across multiple dimensions, resulting in a vast search space.

RecTM addresses this issue by relying on Bayesian optimization techniques [5] in order to steer the selection of the configurations included in the characterization of a TM application.

The rest of the paper is structured as follows. In §2 background on TM and CF is provided. Then, §3 overviews ProteusTM, which is detailed in §4 and §5. The evaluation follows in §6, and conclusions in §7.

2. Background

Next, background on TM is provided, as well as, an overview of prominent Collaborative Filtering techniques for Recommender Systems.

2.1. Transactional Memory

The TM programming model relies on the abstraction of *atomic blocks* to demarcate which portions of code of a concurrent application must execute as atomic transactions. The TM implementation guarantees serializable transactions, by aborting transactions that perform unsafe operations and automatically re-executing them until completion.

Many design and configuration choices have high impact on performance. Next, it will be discussed their associated trade-offs that are self-tuned by ProteusTM.

TM implementations. The TM abstraction has been implemented in software (STM), hardware (HTM), or combinations thereof (Hybrid TM). A wide variety of STMs have been proposed [24]. STMs pose no restrictions on the number of memory accesses of a transaction, but they require costly code instrumentation to track transactional operations. HTMs do not need instrumentation, but they are best-effort [50, 17]: only transactions whose memory footprint fits in the processor's cache can be executed; otherwise they incur a *capacity abort* and resort to a fall-back synchronization. The fall-back is typically a global lock [50], or an STM [9].

Degree of parallelism. The number of concurrently active threads is another parameter with a potentially strong impact on TM performance: a low thread count may lead to sub-utilizing available processing power; a high one, conversely, may induce excessive contention and lead to thrashing [18].

2.2. Collaborative Filtering in Recommender Systems

A Recommender System (RS) seeks to predict the rating that a user would give to an item. These ratings can be exploited to recommend items of interest to users [35]. This work focuses on Collaborative Filtering (CF) [45], a prominent prediction technique used in a RS. To infer the rating of a ⟨user, item⟩ pair, CF techniques exploit the preferences expressed by other users, and ratings by the user on different items. Ratings are stored in a *Utility Matrix* (UM): rows represent users and columns represent items. Typically, a UM is very sparse, as a user rates a small subset of the items. A CF algorithm reconstructs the full UM, from its sparse representation, by filling empty cells with ratings close to the ones that the users would give.

K-Nearest Neighbors (KNN) and Matrix Factorization (MF) are popular CF techniques [41]. KNN uses a *similarity function* to express the affinity of two rows or columns: a recommendation for a pair ⟨u,i⟩ is computed with a weighted average of the ratings of the most similar users to u (and/or on the most similar items to i) [41]. MF, instead, maps users and items to a latent factors space of dimensionality d. Each dimension represents a hidden similarity concept: in the movies' domain,

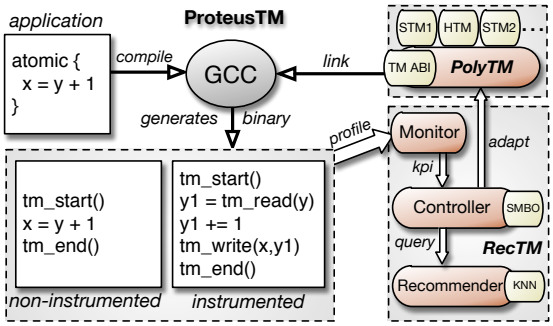


Figure 2: Architecture of the ProteusTM system.

a similarity concept may be how much a user likes drama movies, or how much a movie belongs to the drama category. To compute recommendations, MF infers two matrices P and Q, which represent, respectively, users and items in the aforementioned d-dimensional space. The product of P and Q is a matrix R that is similar to a given UM A, i.e., $Q^T P = R \sim A$, containing also predictions for the missing ratings in A [41].

3. ProteusTM in a Nutshell

In essence, ProteusTM applies Collaborative Filtering (CF) to the problem of identifying the best TM configuration that maximizes a user-defined Key Performance Indicator (KPI): e.g., throughput or consumed energy. ProteusTM aims to maximize the efficiency of TM applications by orchestrating a number of TM algorithms and the dynamic reconfiguration of their parameters. Let us now overview the architecture of ProteusTM, depicted in Fig. 2, which enables its self-tuning capabilities. More details shall be provided in the corresponding sections.

• **PolyTM** §4: consists of a Polymorphic TM library comprising various TM implementations. It allows for switching among TMs and reconfigure several of their internal parameters. It exposes transactional operators via an implementation of the standard TM ABI [38] (supported by GCC [28]).

• **RecTM** §5: is responsible for identifying the best configuration for PolyTM depending on the current workload. It is composed, on its turn, by the following sub-modules:

1. Recommender §5.1: a RS that acts as a performance predictor and supports different CF algorithms. It receives the KPIs of explored configurations from the Controller, and returns ratings (i.e., predicted KPIs) for unexplored ones.

2. Controller §5.2: selects the configurations to be used and triggers their adaptation in PolyTM. It queries the Recommender with the KPI values from the Monitor, obtaining estimates for the ratings of unexplored TM configurations.

3. Monitor §5.3: this module collects the target KPI to (i) give feedback to the Controller about the quality of the current configuration and (ii) detect changes in the workload, so as to trigger a new optimization phase in the Controller.

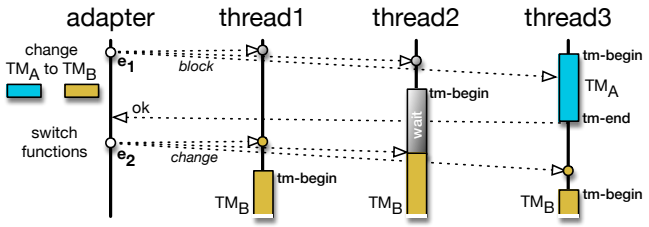


Figure 3: Switching TM algorithm safely in PolyTM.

4. PolyTM: a Polymorphic TM Library

The PolyTM library encompasses a wide variety of TM implementations. It interacts with compilers, like GCC, via the standard TM ABI [28]. Each atomic block, written by the programmer using standard C/C++ constructs [38], is compiled into calls to the various modules of ProteusTM.

For every atomic block, GCC inserts a call to *tm_begin* and *tm_end*, which is direct to PolyTM. Also, two code paths are generated: a non-instrumented path, and a second one in which reads and writes to memory are instrumented with calls to PolyTM. The latter allows the code to arbitrate reads and writes, besides the begin and commit of transactions.

Behind the TM ABI interface, it is implemented in PolyTM several TM algorithms, and run-time support to switch among them: 4 STMs [10, 16, 21, 20], 2 HybridTMs [9, 37], and 2 HTMs [50, 1].

PolyTM collects the commits and aborts at each thread, and the energy consumed by the system. It also uses a dedicated *adapter thread* to change the TM configuration.

In the following, the mechanisms used by PolyTM to support run-time configuration changes are described.

4.1. Switching Between TM Algorithms

The different TM implementations are hidden under a common interface defined in PolyTM. Each thread uses a set of function pointers to this interface to process transaction operations. To switch between TMs, a thread switches the function pointers to a different implementation.

Running concurrent transactions with different TMs is not safe in general [49, 34]. So, PolyTM enforces an invariant: a thread may run a transaction in mode TM_A only if no other thread is executing a transaction in mode TM_B. The problem is illustrated in Fig. 3: at time e_1 the adapter thread tries to change the TM mode; if thread 2 immediately applied the change, it could run mode TM_B concurrently with thread 1 in TM_A. The above invariant guarantees correctness by forcing thread 2 to wait until e_2 to change to TM_B.

The invariant is enforced via an implementation based on the following three steps: (i) adapt parallelism degree (i.e., number of threads) from its current value, say P, to 0; (ii) change TM back-end; (iii) adapt parallelism degree back to P.

4.2. Adapting the Parallelism Degree

The maximum number of active threads is adapted using the synchronization scheme described in Algorithm 1.

Algorithm 1 Changing the parallelism degree in PolyTM.

```
1:  const int RUN ← 1 , BLOCK ← 1 ≪ 32
2:  padded var int threadState[MAX_THREADS] ← { 0 }
3:  function disable-thread(int t)                    ▷ adapter thread
4:    int val ← fetch-and-add(threadState[t], BLOCK)
5:    while (val & RUN) val ← threadState[t]
6:  function enable-thread(int t)                     ▷ adapter thread
7:    threadState[t] ← RUN
8:    signal(t)                          ▷ wakes up thread t (locking omitted)
9:  function tm-start(int t)                          ▷ application thread
10:   int val ← fetch-and-add(threadState[t], RUN)
11:   if (val & BLOCK)
12:     fetch-and-sub(threadState[t], RUN)
13:     cond-wait(t)                    ▷ checks it is still blocked after locking
14:     ▷ ...omitting logic for tm-start...
15: function tm-end(int t)                            ▷ application thread
16:   ▷ ...omitting logic for tm-end...
17:   fetch-and-sub(threadState[t], RUN)
```

Each application thread synchronizes with the adapter thread via a (padded) state variable. When executing a transaction for the first time, a thread is registered in PolyTM. This is simplified in the algorithm by assuming a maximum number of threads, although PolyTM supports an arbitrary number.

Upon starting a transaction, a thread t sets the lowest bit in its state variable (line 10), whereas the adapter thread sets the highest bit of t's state variable when it wants to disable t (line 4). These writes are performed atomically together with returning the state of t. Then, both adapter thread and t can reason on who wins (a potential race): if t sees only the lowest bit set, it is allowed to proceed and executes the transaction; otherwise, it must wait for the adapter to change the mode (line 13). The adapter inversely checks that only the highest bit is set, or else waits for t to unset the lowest bit (line 5) — because t was already executing a transaction.

Also, a conditional variable is used, and associated with each thread t, for t to wait on, in the case it is disabled. The details of its management have been omitted, for simplicity of presentation.

PolyTM guarantees that a reconfiguration always terminates: a thread eventually commits a pending transaction, or else aborts and checks whether it was disabled — assuming finite atomic blocks. Hence, the duration of a reconfiguration depends on the longest running transaction. This, however, does not impair the efficiency of PolyTM's reconfiguration: in-memory transactions are generally very fast (given that they do not entail I/O) [48, 33].

In addition, the success of a reconfiguration does not rely on threads to eventually call into ProteusTM. This is crucial to cope with applications whose threads may wait for events (e.g., client requests) and do not run atomic blocks often.

5. RecTM: a Recommender System for TM

RecTM optimizes PolyTM via a *black-box* methodology that relies on a novel combination of off-line and on-line learning. In short, it operates according to the work-flow of Algorithm 2: (i) build a *training set* by profiling the KPI of an initial set of applications in the encompassed TM configurations (line 1); (ii) instantiate a CF-based performance predictor based on the training set obtained off-line in (i) (lines 2 and 3); (iii) upon deploying a new application or detecting a change of the workload, profile on-line the application over a small set of explored configurations (lines 4 and 5); iv) recommend a configuration for the workload (line 6).

In the following, the building blocks of RecTM will be detailed.

5.1. Recommender: Using Collaborative Filtering

RecTM casts the identification of the optimal TM configuration for a workload into a recommendation problem, which it tackles using Collaborative Filtering (CF) [45].

A key challenge to successfully apply CF in predicting the performance of TM applications, is that CF assumes the ratings in a predetermined scale (e.g., a preference from 0 to 10). The absolute KPI values produced by different TM applications, instead, can span orders of magnitude (e.g., from millions [6] to few txs/sec [22]). Further, KPI values of specific configurations provide no indication on the max/min KPI that the application can obtain, impairing their normalization.

Recommender tackles this issue with an innovative technique, which let us address it as *rating distillation*. This function maps KPI values of diverse TM applications onto a rating scale that can be fruitfully exploited by CF to identify correlations among the performance trends of heterogeneous applications.

The Rating Heterogeneity Problem. Ratings are stored in a Utility Matrix (UM) A, of which each row u represents a workload and each column i is a TM configuration: $A_{u,i}$ is the rating of configuration i for workload u (i.e., in the chosen domain, it expresses the performance of i in u for a given KPI metric). To illustrate the problem, let us populate the UM directly with sampled KPI values (e.g., throughput): $\begin{pmatrix} 1 & 2 & 3 \\ 30 & 20 & 10 \\ 100 & 200 & ? \end{pmatrix}$, which contains information on applications A_1 and A_2 profiled with configurations C_1, C_2 and C_3 and A_3 profiled only at C_1 and C_2. Let us assume that C_i is an application running with a given TM and i threads. From the matrix, it is possible to infer that A_1 can scale, as its performance increases linearly with the number of threads; A_2 does not, since its performance, though higher in absolute value than A_1's, decreases as the number of threads grows. Assume that it is required to predict the rating for $A_{3,3}$. Note that A_3 exhibits the same linear trends of A_1: for this reason, a likely value for $A_{3,3}$ would be 300. Next, it will be shown why well-known CF techniques can be misled because of the heterogeneity of the ratings' scales in the UM.

The Need for Normalization. The most common similarity functions in KNN CF are the Euclidean, Cosine and Pearson [41]. The first cannot be applied to heterogeneous ratings. The other two are scale-insensitive, so they are able to identify C_1 as similar to C_3. However, they would yield an incorrect

Algorithm 2 RecTM work-flow

1: Off-line performance profiling of an initial training set of applications.
2: Rating distillation and construction of the Utility Matrix (Section 5.1).
3: Selection of CF algorithm and setting of its hyper-parameters).
4: Upon the arrival of a new workload (Section 5.3):
5: Sample the workload on a small set of initial configurations (Section 5.2).
6: Recommend the optimal configuration (Section 5.1).

prediction in absolute value, as it will lie on C_1's scale, which is different from C_3's.

A solution to these problems is to normalize the entries in UM. An effective normalization function should fulfill two requirements: (i) to transform entries in the UM so that similarities among heterogeneous applications can be mined and (ii) to enable the application of conventional CF techniques.

Next, it is described how ProteusTM normalizes ratings to meet the two aforementioned requirements and, thus, enables CF to optimize TM applications.

Normalization in the Recommender. The rating distillation used by the Recommender uses a mapping function that, for any workload w in the UM, ensures: (i) the ratio between the performance of two configurations c_i, c_j is preserved in the rating space, i.e., $\frac{kpi_{w,c_i}}{kpi_{w,c_j}} = \frac{r_{w,c_i}}{r_{w,c_j}}$; and (ii) the ratings of the corresponding configurations, $r_{w,c}$, are distributed (assuming a maximization problem) in the range [0,M_w], so as to minimize the index of dispersion of M_w: $D(M_w) = var(M_w)/mean(M_w)$.

Property (i) ensures that the information about the relative distances of two configurations is correctly encoded in the rating spaces. Property (ii) aligns the scales that express the ratings of each workload w to use similar upper bounds M_w, which are tightly distributed around their mean value.

The rating of each row is obtained by normalizing its KPI with respect to a column C*$\in C_M$, so to minimize the index of dispersion among the resulting maximum ratings in the normalized domain.

5.2. Controller: Explorations Driven by Bayesian Models

The Controller uses Sequential Model-based Bayesian Optimization (SMBO) [27] to drive the on-line profiling of incoming workloads, to quickly identify optimal TM configurations.

SMBO is a strategy for optimizing an unknown function $f : D \rightarrow \mathbb{R}$, whose estimation can only be obtained through (possibly noisy) observation of sampled values. It operates as follows: (i) evaluate the target function f at n initial points $x_1 \ldots x_n$ 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. Controller uses as acquisition function the criterion of *Expected Improvement (EI)* [30], which selects the next point to sample based on the gain that is expected with respect to the currently known optimal configuration. More formally, considering without loss of generality a minimization 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 $x_{min} = \arg\min_{x \in D_u} f(x)$. Then the positive improvement function I over $f(x_{min})$ associated with sampling a point x is $I_{x_{min}}(x) = max\{f(x_{min} - f(x), 0\}$. Since f has not been evaluated on x, $I(x)$ is not known *a priori*; however, thanks to the predictive model M fitted over past observations, it is possible to obtain the expected value for the positive improvement: $EI_{y(x_{min})}(x) = \mathbb{E}[I_{y(x_{min})}(x)] = \int_{-\infty}^{y(x_{min})} (f_{x_{min}} - c) p_M(c|x) dc$. Here, $p_M(c|x)$ is the probability density function that the model M associates to possible outcomes of the evaluation of f at point x [30].

Machine ID	Processor / Number of cores / RAM	HTM	RAPL
Machine A	1 Intel Haswell Xeon E3-1275 3.5GHz / 4 (8 hyper-threads) / 32 GB	Yes	Yes
Machine B	4 AMD Opteron 6172 2.1 Ghz / 48 / 32 GB	No	No

Table 1: Machines used in the experimental test-bed.

Computing $p_M(c|x)$. The Controller computes $p_M(c|x)$ with an ensemble of CF predictors, and obtains predictive mean μ_x and variance σ_x^2 of $p(c|x)$ as frequentist estimates over the output of its individual predictors evaluated at x. It then models $p_M(c|x)$ as a Gaussian distribution $\sim N(\mu_x, \sigma_x^2)$. Assuming a Normal distribution for $p(c|x)$ is frequently done in SMBO [27] and other optimization techniques [39] to ensure tractability. Given a Gaussian distribution for $p_M(c|x)$, $EI_{y(x_{min})}(x)$ can be computed in closed form as $EI_{y(x_{min})}(x) = \sigma_x[u\Phi(u) + \phi(u)]$, where $u = \frac{y(x_{min}) - \mu_x}{\sigma_x}$ and Φ and ϕ represent, respectively, the probability density function and cumulative distribution function of a standard Normal distribution [30].

Stopping Criterion. As discussed, SMBO requires the definition of a predicate to stop exploring new configurations.

Controller uses a stopping criterion that seeks a balance between exploration and exploitation by relying on the notion of EI: it uses the estimated likelihood that additional explorations may lead to better configurations.

5.3. Monitor: Lightweight Behavior Change Detection

The Monitor periodically gathers KPIs from PolyTM. These are used for two tasks: (i) while profiling a new workload, they are fed to the Controller, providing feedback about the quality of the current configuration; (ii) at steady-state, they are used to detect a workload change. The Monitor implements the Adaptive CUSUM algorithm to detect, in a lightweight and robust way, deviations of the current KPI from the mean value observed in recent time windows [2]. This allows the Monitor to detect both abrupt and smooth changes and to promptly trigger a new profiling phase in Controller. Note that environmental changes (e.g., inter-process contention or VM migration) are indistinguishable from workload changes from the perspective of the implemented behavior change detection.

Machine ID	TM Backend	# threads	HTM Abort Budget	HTM Capacity Abort Policy
Machine A	STMs and TSX [50]	1,2,3,4, 5,6,7,8	1,2,4, 8,16,20	Set budget to 0; decrease budget by 1; halve budget
Machine B	STMs	1,2,4,6, 8,16,32,48	N/A	N/A

Table 2: Parameters tuned by ProteusTM. STMs are TinySTM [21], SwissTM [20], NORec [10] and TL2 [16].

6. Evaluation

This section provides an extensive validation of ProteusTM. Section 6.1 introduces the testbed, applications, and accuracy metrics used. In Section 6.2 the overhead incurred by PolyTM to provide self-tuning capabilities is assessed. In Section 6.3, the effectiveness of RecTM's components is evaluated in separate.

6.1. Experimental Test-Bed

ProteusTM was deployed in two machines with different characteristics (described in Table 1) and used a wide variety of standard TM benchmarks. Over 300 workloads were considered, which are representative of heterogeneous applications, from highly to poorly scalable, from HTM to STM friendly [19]. Moreover, three KPIs were tested: execution time, throughput and EDP (Energy Delay Product, a popular energy efficiency metric [26]). The energy consumption was measured via RAPL [12] (available on Machine A).

The system optimizes the KPI by tuning the four dimensions listed in Table 2. Overall, a total of 130 TM configurations are considered for Machine A and 32 for Machine B.

Evaluation metrics. The performance of ProteusTM is evaluated by considering 2 accuracy metrics: Mean Average Percentage Error (MAPE) and Mean Distance From Optimum (MDFO).

Noting $r_{u,i}$ the real value of the target KPI for workload u when running with i as configuration, $\widehat{r}_{u,i}$ the corresponding prediction of the Recommender, and S the set of testing $\langle u,i\rangle$ pairs, **MAPE** is defined as: $\sum_{\langle u,i\rangle\in S} |r_{u,i}-\widehat{r}_{u,i}|/r_{u,i}$.

Noting with i^*_u the optimal configuration for workload u and with $\widehat{i^*_u}$ the best configuration identified by the Recommender, the **MDFO** for u is computed as: $\sum_{\langle u,\cdot\rangle\in S} |r_{u,i^*_u} - r_{u,\widehat{i^*_u}}|/r_{u,i^*_u}$.

MAPE reflects how well the CF learner predicts performance for an application. In contrast, MDFO captures the quality of final recommendations output by the Recommender.

6.2. Overhead Analysis and Reconfiguration Latency

The overhead of PolyTM, is now assessed i.e., the inherent steady-state cost of supporting adaptation. Let us proceed to the performance comparison of a bare TM implementation T with that achieved by PolyTM using T without triggering adaptation.

Table 3 summarizes the results averaged across all benchmarks. The contention management for HTM is set to decrease linearly the retries starting from 5 (a common setting [50, 31]).

#threads	TL2	NOrec	Swiss	Tiny	HTM-opt	HTM-naive
1	3	3	2	3	5	13
4	<1	1	<1	3	6	12
8	<1	<1	<1	4	8	19

Table 3: Overhead (%) incurred by ProteusTM for different TM and # threads. Results are an average across ten runs.

Benchmark (Machine)	# Threads					
	1	2	4	8	16	32
TPC-C (Machine A)	21	91	213	3419	N/A	N/A
Memcached (Machine B)	2	8	28	145	1103	1849

Table 4: Reconfiguration (TM and #threads) latency (μsec).

It is also shown the overhead of the optimized code path, employed for HTM, and the one resulting from the default GCC instrumentation (fully instrumented path).

These experiments reveal overheads consistently $<5\%$ for STMs and only slightly larger for HTMs (at most 8%). The lower STM overhead is justifiable considering that STMs natively suffer from instrumentation costs that end up amortizing most of the additional overhead introduced by PolyTM.

Also, it is assessed the average latency of a typical reconfiguration in PolyTM to switch TM algorithms (which also entails changing the number of threads). The results, shown in Table 4, encompass two heterogeneous workloads: Memcached uses 100$\times$ shorter transactions than TPC-C. The results highlight the practicality of the system's reconfiguration algorithm. Even in the worst case of large transactions in TPC-C, the latency is negligible: in fact, this is only incurred during the exploration phase, which, as we shall see, is kept very short by ProteusTM.

6.3. Quality of the Prediction and Learning Processes

Let us evaluate each of RecTM's components by means of a trace-driven simulation. Traces of real executions were collected using a subset of the test cases (namely, STAMP and Data Structures), averaging the results over 5 runs.

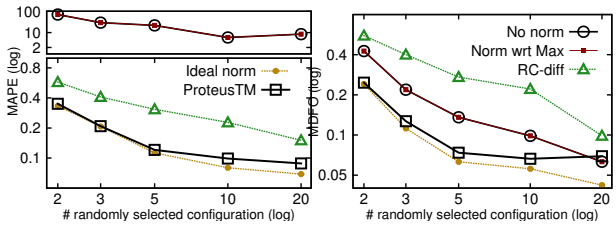
The data-set was split into a training set (30%) and a test set (70%). The training set is used to choose and tune the CF algorithm and to instantiate the predictive model. It is used 10 learners for the bagging ensemble, as this is a typical value [27, 46]. To simulate sampling the performance of the application in a given configuration, the corresponding value from the test set is inserted in the UM of the Recommender.

Rating distillation. The effectiveness of the distillation function was assessed versus several UM preprocessing techniques:

(i) No normalization: CF is applied on the UM containing raw KPI samples. This is equivalent to Quasar [15]);

(ii) Normalization w.r.t. max.: entries in the UM are relative to the highest value, supposed to be known a priori. It resembles Paragon's approach [14], where the machine's peak instructions/sec rate is used as normalizing constant;

(iii) Ideal normalization: the scheme described in Section 5.1;



(a) MAPE, KNN Cosine (b) MDFO, KNN Cosine

Figure 4: Rating distillation for Exec. Time on Machine A.

(iv) Row-column subtraction: noted *RC*, is typically employed in CF to cope with biases in users and item ratings [41]. It consists in removing from each known rating the average value of the corresponding row; then, the average value per column — computed after the first subtraction — is subtracted;
(v) Rating distillation: used in ProteusTM.

A subset of results are shown, focusing on execution time KPI on Machine A employing KNN with cosine similarity. The number of *randomly chosen* known ratings per row is varied and then MAPE and DPO are accordingly computed..

Fig. 4 shows that using no normalization, or normalization w.r.t. the maximum performs very poorly, both in terms of MAPE (Fig. 4a) and MDFO (Fig. 4b). Both yield similar results as a normalization w.r.t. any constant. RC is subject to lower MAPE than the two aforementioned normalizations, yet its accuracy is significantly lower than rating distillation's, both in terms of MAPE and MDFO. Also, the approach of ProteusTM closely follows the ideal normalization. To ensure a fair comparison, the same training set was used, and not forcing the presence of the column used for normalization among the profiled configurations for ProteusTM.

Controller. Let us now evaluate the proposed EI-based approach (called EI) with respect to a randomized sampling approach, used in Quasar and Paragon [14, 15]), and two other SMBO approaches using acquisition functions different from EI: Variance explores configurations with high uncertainty for the underlying model (i.e., high $variance/mean$ ratio); Greedy explores the configuration with highest predictive mean.

The simulation proceeds in rounds: each one profiles the target workload on the reference configuration chosen by the rating distillation function; then the sampling phase begins. Afterwards, the Recommender produces a recommendation for the optimal configuration, noted $\hat{c^*}$. If such a configuration is explored, then the optimization is concluded; otherwise, a final exploration of $\hat{c^*}$ is performed. The final recommendation c^* is the one which, among those explored, yields the best performance. The MDFO is computed on the basis of c^* and the MAPE is an average MAPEs computed per workload.

In Fig. 5a, it is reported the MDFO for EDP (on Machine A). The EI exploration policy is able to identify a high quality solution requiring, on average, less explorations than any competitor. Fig. 5b shows that the 80-th percentile of the DFO obtained by EI — after 5 explorations — is less than 10%. Note that, the EDP KPI was the most challenging to optimize: hence, the latter result represents a lower bound on the system's accuracy.

In Fig. 5d, it is shown the MDFO when optimizing execution time (on Machine B): once again, the EI-based Controller's exploration performs best. Fig. 5c shows the MAPE per explorations. Interestingly, the Variance policy has the best *mean* prediction accuracy. However, as it does not aim at sampling potential optimal solutions, but only at reducing uncertainty, it does not learn the behavior of the target function for potentially good configurations. Thus, the quality of the recommended configurations is significantly worse than EI's (see Fig. 5d).

Finally, let us compare EI policy with random sampling in Figs. 5a and 5d: taking 5% distance as reference, EI achieves a number of explorations vs MDFO trade-off that is up to 4× better than its competitor. This highlights the effectiveness of the SMBO-based approach over simpler sampling techniques used in recent systems [15, 14]).

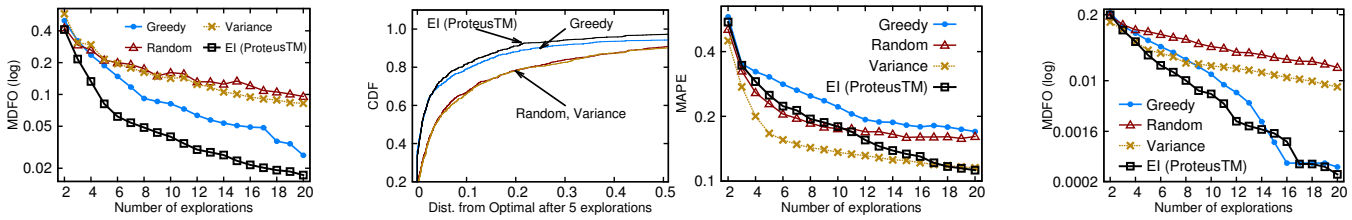
Comparison with ML approaches. Let us now compare ProteusTM with an approach based on the same technique proposed by Wang et. al [49] to automate the choice of the TM algorithm for a given workload. This approach relies on workload characterization data to train a ML-based classifier that is used to predict the best TM configuration for a given workload. The workload characterization uses 17 features: e.g., duration of transactions, data access patterns, and level of data contention. Wang et al. also uses static analysis to obtain other features, e.g., the number of atomic blocks. This step was not performed but we extend the set of features proposed by the authors with information on contention management (Aggressive, Suicide, Polite, Karma, Timestamp). These features were not considered by the authors, but they were found to be highly correlated with performance.

The simulation for ProteusTM evolves as previously explained. For the ML competitor, instead, a workload is first profiled over a reference configuration (TinySTM, 4 threads) and then the ML is invoked to predict the best configuration. After it, the MDFO is computed for this predicted configuration.

300 STAMP and Data Structures workloads were used in Machine A, and splitted randomly into training and test sets: 30-70 and 70-30 train-test splits. For ProteusTM, the training set is the UM corresponding to the selected workloads; for ML approaches, the training set is composed, for each workload, by the aforementioned features and the identifier of the best configuration as target class. The target KPI is throughput.

3 ML algorithms, implemented in Weka [23] were considered: Decision Trees (CART), Support Vector Machines (SMO), and Artificial Neural Networks (MLP) [4]. Their parameters were chosen via random search optimization [3], which evaluated 100 combinations with cross-validation on the training set.

Fig. 6 reports the CDF of the DFO of each technique over 10



(a) MDFO for EDP (b) CDF of DFO for EDP (c) MAPE for Exec. Time (d) MDFO for Exec. Time

Figure 5: Controller's exploration policies for EDP on Machine A (left figures) and Exec. Time on Machine B (right figures).

runs. The data shows the superiority of ProteusTM relatively to pure ML approaches. In particular, with 30% training set, ProteusTM already delivers a DFO of 1.6% against the 10% of the ML competitors, and a 90-th percentile of 3.5% against 25% of CART (the best alternative). Also, by increasing the training set to 70%, ProteusTM delivers a DFO of 1.3% and a 90-th percentile of 3%, against 6.8% DFO and 21% 90-th percentile of the best alternative (SMO).

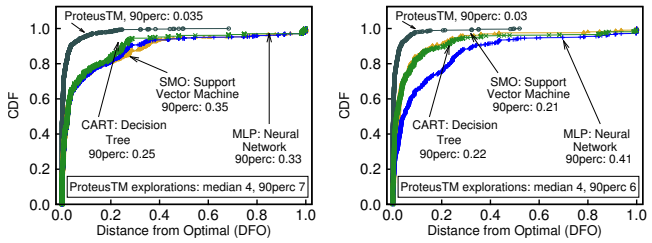
Note that the DFO of ProteusTM is similar (both in mean and 90-th percentile) in both cases, whereas ML greatly benefits from more training data. This difference can be explained by the number of explorations required by ProteusTM to perform its profiling phase (with threshold $\varepsilon = 0.01$): at 30% training, the 90-th percentile number of explorations is 7, but this lowers to 6 with 70% training set. This means that ProteusTM delivers high accuracy also in presence of scarce training data, by autonomously exploring more.

The evaluation suggests that detecting similarities on the KPI is more effective than statistically inferring relationships from training data. This possibly depends on two, tightly intertwined, causes: (i) thanks to the employed novel normalization, using CF is more robust than ML, as it is based on *direct KPI observations*, rather than on *learning* the mapping of input to output features; (ii) the adaptive profiling phase proved to be more effective than a *one-shot* classification-based solution.

6.4. Online Optimization of Dynamic Workloads

In Fig. 7, the ProteusTM system is evaluated as a whole. Both RecTM and PolyTM amount to 6.5K and 6K lines of code, excluding third party code (e.g., Mahout and TMs).

4 use cases for ProteusTM's runtime optimization are shown on 2 TM benchmarks (Red-Black Tree and STMB7 [22]), a TM porting of TPC-C [47] and of Memcached [43]. For each application 3 workloads are triggered and chosen to exemplify



(a) 30% Training data used. (b) 70% Training data used.

Figure 6: Comparison vs Machine Learning based techniques.

Machine	Benchmark		Mean Distance from Optimum (MDFO %)			
			Optimal in Workload i			ProteusTM
	Name	Workload (Opt Conf)	Opt 1	Opt 2	Opt 3	(explorations)
A	RBT	1 (NOrec: 7t)	0	137	93	$<$ **1** (4 expl)
		2 ★ (HTM:8t Half-20)	33	0	71	**2** (4 expl)
		3 (HTM: 4t GiveUp-4)	154	37	0	$<$ **1** (7 expl)
A	STMB7	1 (HTM: 4t Linear-2)	0	20	210	**2** (6 expl)
		2 (Swiss: 4t)	135	0	28	$<$ **1** (4 expl)
		3 ★ (TL2: 8t)	390	29	0	$<$ **1** (3 expl)
A	TPC-C	1 ★ (Tiny: 4t)	0	273	47	$<$ **1** (3 expl)
		2(HTM:3t GiveUp-16)	68	0	152	**3** (4 expl)
		3 (Tiny: 8t)	22	370	0	$<$ **1** (3 expl)
B	Memchd	1 ★ (Swiss: 32t)	0	50	26	**4** (3 expl)
		2 (Tiny: 32t)	19	0	258	$<$ **1** (4 expl)
		3 (Tiny: 4t)	18	66	0	$<$ **1** (3 expl)

Table 5: For each benchmark (of Fig. 7), it is shown the MDFO (in %) of ProteusTM, each Optimal and BFA (★) configurations. Each workload is labeled with its optimal configuration.

contrasting characteristics and resulting performances. In each case, ProteusTM is totally oblivious of the target application: no workloads of the application are present in its training set. This highlights the Recommender's ability to detect similarity patterns between the target workloads and the set of *disjoint* applications used as training set.

Setting the Monitor period to 1 sec and the SMBO ε to 0.01. In each run, let us measure the performance of (i) ProteusTM, (ii) the 3 configurations that perform best in each workload, (iii) the Best Fixed configuration on Average (BFA) across the workloads, and (iv) a Sequential non-instrumented execution.

Three conclusions can be drawn from these plots: (i) ProteusTM is able to quickly identify, at runtime, configurations that are optimal — or very close. Remarkably, ProteusTM delivers performance that is, on average, only 1% lower than the optimal; (ii) employing *any* of the baseline alternatives yields up to two orders of magnitude lower performance; (iii) thanks to the SMBO approach, the performance degradation incurred when exploring is minimal (at most 7 explorations in these use cases). Such cost is usually amortized in long-running services (e.g., databases), in which workload shifts are infrequent [8].

A summary is provided in Table 5 where it is listed the optimal configurations in each workload. It also shows the BFA (with ★) which is always also an optimal configuration in some workload. This data highlights the robustness of ProteusTM to optimize heterogeneous applications with diverse optimal configurations, in terms of TM algorithm (STMB7), parallelism degree (TPC-C) and HTM tuning (RBT and Mem-

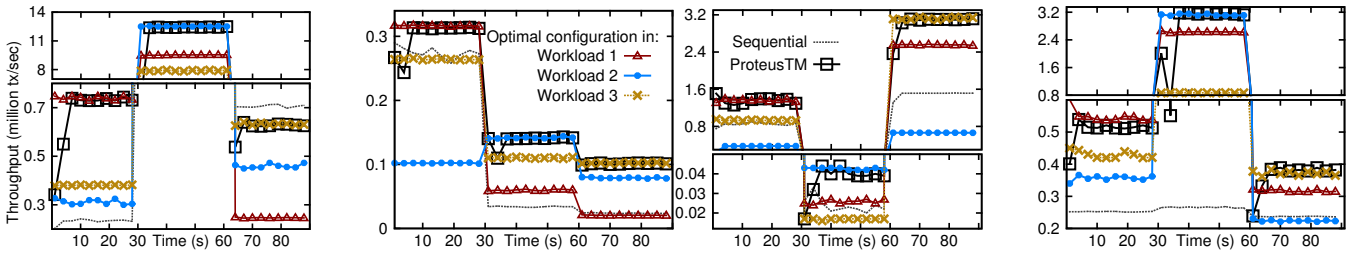


Figure 7: Performance of four applications when their workload changes three times. It is shown the performance obtained with ProteusTM, and three additional fixed configurations, each one corresponding to an optimum in each workload.

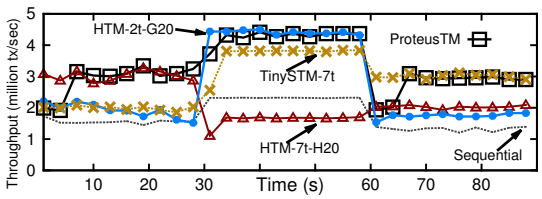


Figure 8: Similar to Fig. 7c, but with a static application workload, varying instead the availability of machine resources.

chd).

Finally, in Fig. 8, what was claimed in Section 5.3 is confirmed by using a static TPC-C workload and varying external factors to the application to trigger behavior changes. To simulate these external changes the *stress* Unix tool was used with different configurations over periods of 30 seconds: it either created high CPU, memory or IO usage in each workload. The results are similar to what was chosen previously, in that ProteusTM obtains perform close to the optimal across the test.

7. Conclusions

This thesis presents ProteusTM, the first TM system with multi-dimensional self-tuning capabilities. ProteusTM is integrated with GCC and exposes a standard TM interface, which ensures full transparency, ease of use and portability. At its heart, ProteusTM relies on a novel self-tuning technique that leverages on Collaborative Filtering and Baeysian Optimization.

Via an extensive evaluation based on real-word application and well-known benchmarks, it was demonstrated ProteusTM's capability to optimize heterogeneous applications in high-dimensional configuration spaces: ProteusTM achieves performance that are, on average, $< 3\%$ from optimum and gains up to $100\times$ relatively to static configurations.

References

[1] Allon Adir, Dave Goodman, Daniel Hershcovich, Oz Hershkovitz, Bryan Hickerson, Karen Holtz, Wisam Kadry, Anatoly Koyfman, John Ludden, Charles Meissner, Amir Nahir, Randall R. Pratt, Mike Schiffli, Brett St. Onge, Brian Thompto, Elena Tsanko, and Avi Ziv. Verification of transactional memory in power8. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 58:1–58:6, New York, NY, USA, 2014. ACM.

[2] Michèle Basseville and Igor V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., 2006.

[5] Eric Brochu, Vlad M Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. eprint arXiv:1012.2599, arXiv.org, December 2010.

[6] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 51(11):40–46, 2008.

[8] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[9] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[10] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[11] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[12] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[13] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The youtube video recommendation system. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 293–296, New York, NY, USA, 2010. ACM.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 77–88, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 127–144, 2014.

[16] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168, 2009.

[18] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Joerg Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing Journal*, pages 1–21, December 2013.

[19] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM.

[20] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[21] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[22] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[23] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[24] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[25] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[26] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, Oct 1994.

[27] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. https://gcc.gnu.org/wiki/TransactionalMemory?action=AttachFile&do=get&target=Intel-TM-ABI-1_1_20060506.pdf, 2009.

[29] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the Annual nternational Symposium on Microarchitecture (MICRO)*, pages 25–36. IEEE Computer Society, 2012.

[30] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *J. of Global Optimization*, 13(4):455–492, December 1998.

[31] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel transactional synchronization extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 476–487, Feb 2014.

[32] Andi Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, March 2014.

[33] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[34] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007.

[35] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[36] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM.

[37] Alexander Matveev and Nir Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM.

[38] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 195–212, New York, NY, USA, 2008. ACM.

[39] Takayuki Osogami and Sei Kato. Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 145–156, New York, NY, USA, 2007. ACM.

[40] Victor Pankratius and Ali-Reza Adl-Tabatabai. Software engineering with transactional memory versus locks in practice. *Theor. Comp. Sys.*, 55(3):555–590, October 2014.

[41] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[42] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Not.*, 45(5):47–56, January 2010.

[43] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 399–412, New York, NY, USA, 2014. ACM.

[44] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 278–285, Washington, DC, USA, 2012. IEEE Computer Society.

[45] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:4:2–4:2, January 2009.

[46] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 847–855, New York, NY, USA, 2013. ACM.

[47] TPC Council. TPC-C Benchmark. http://www.tpc.org/tpcc, 2011.

[48] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[49] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim.*, 8(4):54:1–54:23, January 2012.

[50] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19. ACM, 2013.