ProteusTM: Abstraction Meets Performance in Transactional Memory

Diego Didona Nuno Diegues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
didona,ndiegues@gsd.inesc-id.pt

Anne-Marie Kermarrec
INRIA
anne-marie.kermarrec@inria.fr

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Ricardo Neves Paolo Romano
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
ricardo.neves,paolo.romano@tecnico.ulisboa.pt

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 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 their wide adoption.

In this paper, we address the challenge of automatically identifying the best TM implementation for a given workload. Our proposed system, ProteusTM, hides behind the TM interface a large library of implementations. Underneath, it leverages a novel multi-dimensional online optimization scheme, combining two popular learning techniques: Collaborative Filtering and Bayesian Optimization.

We integrated ProteusTM in GCC and demonstrate its ability to switch between TMs and adapt several configuration parameters (e.g., number of threads). We extensively evaluated ProteusTM, obtaining average performance $< 3\%$ from optimal, and gains up to $100\times$ over static alternatives.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Transactional Memory, Recommender Systems, Performance Tuning, Adaptive System

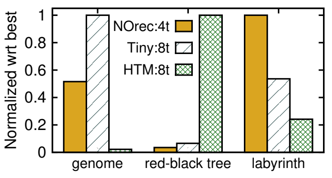
ASPLOS '16, April 02–06, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872385

1. Introduction

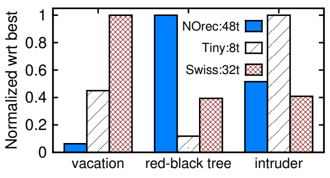
The advent of multi-cores has brought parallel computing to the fore-front of software development, fostering research on paradigms to simplify the development of concurrent applications. The Transactional Memory (TM) [35] abstraction is a prominent approach that promotes a simple idiom for synchronizing 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 [46, 52, 57] on the effectiveness of TM to simplify the development and verification of concurrent programs, enhancing code reliability and productivity. Recently, the relevance of TM was amplified by the standardization of constructs in popular languages (such as C/C++ [49]), and by the integration of hardware support in processors by Intel and IBM [39, 67].

The abstraction vs performance dilemma. Unfortunately, TM performance remains a controversial matter [11]: 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. We report on the energy efficiency (in Fig. 1a) and throughput (in Fig. 1b) of various TMs in different architectures and benchmarks. We normalized the data 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 [27, 29], but configuring them 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 2).



(b) Throughput on a multi-chip 48-core CPU (Machine B in Table 2).

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. Performance can be affected by a number of factors, including program inputs [26], phases of program execution [24], tuning of internal parameters [25], as well as architectural aspects of the underlying hardware [12].

Given the vast TM design space, manually identifying optimal configurations, using trial and error on each workload, 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 [42].

Contributions. We propose a new system, ProteusTM[1], which allows developers to enjoy the simplicity and ease of usage of the TM abstraction, while sparing them from the burden of tuning TM implementations to specific workloads.

Under the simple and elegant interface of TM, ProteusTM hides a large library of TM implementations. At run-time, ProteusTM relies on an combination of learning techniques to pursue optimal efficiency via multi-dimensional adaptation of the TM implementation and its parameters.

At the heart of ProteusTM lie two key components:

• **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 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 [54]. 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 [59] and (possibly inaccurate) static code analysis [65] employed by other machine learning-based solutions.

While building ProteusTM, we solved several challenges:

► *Minimizing the cost of adaptivity.* Supporting reconfigurations across multiple dimensions requires introducing some 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 our self-tuning.

We addressed this challenge by designing lightweight synchronization schemes that exploit compiler-aided, asymmetric code instrumentation. The combination of these techniques allows PolyTM to achieve average and maximum overhead of 1% and 5%, even when considering the most performance sensitive TM implementations.

► *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 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.

We tackled this issue by integrating PolyTM in GCC, via the standard TM ABI [49], and by exposing to programmers standard C++ TM constructs. Not only this preserves the simplicity of TM, but it also maximizes portability due to the widespread availability of GCC across architectures.

► *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) [16, 19, 45]. 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 [20, 21]. One key issue 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 KPIs of TM applications can span very heterogeneous scales. As we shall see, this can severely hinder the accuracy of existing RS techniques.

We cope with this issue by introducing a novel normalization technique, 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 [61] 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 reconfigurations across multiple dimensions, resulting in a vast search space.

RecTM addresses this issue by relying on Bayesian Optimization techniques [8] to steer the selection of the configurations included in the characterization of a TM application. This reduces by up to $4\times$ the duration of the learning process of the RS using Collaborative Filtering (CF) [61].

[1] *Proteus* is a Greek god who can foretell the future and adapt his shape.

We conducted an extensive evaluation of ProteusTM using 15 TM applications, a parameter space of up to 130 configurations, and optimizing 2 metrics: performance and energy efficiency. Our results highlight that ProteusTM obtains quasi-optimal performance (on average $< 3\%$ from optimal) and gains up to 2 orders of magnitude over static alternatives.

While ProteusTM does not solve all challenges of TM (e.g., coping with side-effects of transactions), it drastically improves performance while preserving its simplicity.

The rest of the paper is structured as follows. In §2 we provide background on TM and CF. Then, §3 overviews ProteusTM, which we detail in §4-5. The evaluation follows in §6, with the related work in §7 and conclusions in §8.

2. Background

Next, we provide background on TM and overview 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, we discuss 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 [34]. 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 [23, 67]: 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. This is typically a global lock [67], or an STM [14].

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 subutilizing available processing power; a high one, conversely, may induce excessive contention and lead to thrashing [24].

Contention management. A TM contention manager [30] is in charge of arbitrating conflicts (e.g., by backing off transactions). In HTM, contention management is also responsible for dealing with problematic transactions that might suffer from best-effort caused aborts. Typical approaches allocate a budget of retries to hardware transactions, upon whose exhaustion they resort to the fall-back scheme. Tuning the initial budget, and the retry policy, has significant impact on performance and is strongly workload dependent [25].

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 [45]. We focus on Collaborative Filtering (CF) [61], a prominent prediction technique used in a RS. To infer the rating of a $\langle$user, item$\rangle$ 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 [54]. KNN uses a *similarity function* to express the affinity of two rows or columns: a recommendation for a pair $\langle$u,i$\rangle$ 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) [54]. 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, 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 [54].

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 various TM algorithms and their dynamic reconfiguration. We 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 [49] (supported by GCC [38]).

• **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.

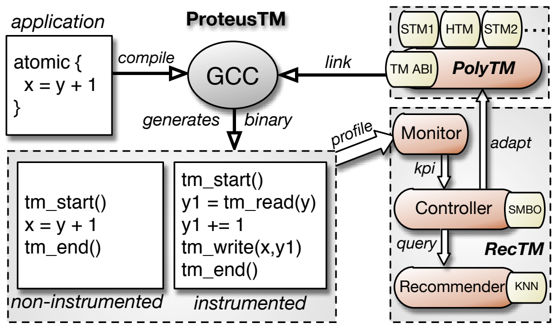


Figure 2: Architecture of the ProteusTM system.

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.

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 [38]. Each atomic block, written by the programmer using standard C/C++ constructs [49], 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 we direct to PolyTM. Also, two code paths are generated: a non-instrumented path, and a second one in which reads and writes are instrumented with calls to PolyTM. The latter allows our code to arbitrate reads and writes, besides the begin and commit of transactions.

Our system is able to integrate with any new TM backend, as long as this backend maintains its metadata (such as locks, ownership records, etc.) in separate memory regions, i.e., it does not interfere with the original memory layout of the application. This is the case for most TMs we are aware of.

Behind the TM ABI interface, we implemented in PolyTM several TM algorithms[2], and run-time support to switch among them: 4 STMs [15, 22, 27, 29], 2 HybridTMs [14, 47], and 2 HTMs [1, 67]. We take advantage of the dual compilation paths and use the instrumented one for the STMs. In contrast, HTMs — which automatically transactionalize reads and writes — execute the non-instrumented one. As shown in §6.2, the dual path optimization is crucial to minimize overhead.

The compiled code is also instrumented to profile performance metrics in a lightweight and transparent manner. In particular, 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.

[2] We used open-source TMs by mapping ProteusTM's implementation of the TM ABI to the API of each TM with a thin software layer.

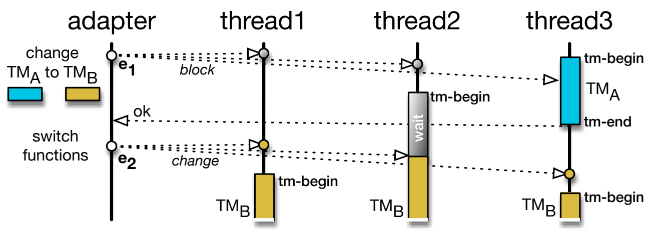


Figure 3: Switching TM algorithm safely in PolyTM.

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

4.1 Switching Between TM Algorithms

Since our library must interact with the compiler via a single ABI, we hide different TM implementations under a common interface defined in PolyTM. Then, 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 [44, 65]. 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$_B$. We illustrate the problem 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 using the following steps: (i) adapt parallelism degree (i.e., number of threads) from the current value, say P, to 0; (ii) change TM back-end; (iii) adapt parallelism degree back to P.

4.2 Adapting the Parallelism Degree

To adapt the maximum number of active threads we use the synchronization scheme described in Algorithm 1.

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. We simplify this in the algorithm by using 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

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)
```

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.

We implement these atomic operations with the primitives fetch-and-*op* (e.g., *op* = *add* would be XADD in x86). These primitives always succeed, and are cheaper than the traditional compare-and-swap loop [18, 48]. Furthermore, in the common case of our algorithm — a thread starting a transaction is not concurrently disabled — each thread performs the atomic operation on a variable residing (with high probability) in its cache and without contention. In this case, the latencies (in processor cycles), in our Machine A are 17 cycles for a fetch-and-add and 32 for a compare-and-swap. As such, the cost for managing the number of active threads is quite limited, for instance when compared to the begin and commit of a hardware transaction (>120 cycles [56]).

We also use a conditional variable, associated with each thread t, for t to wait on, in the case it is disabled. We omit the details of its management, 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) [43, 64].

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.

We note that, depending on the application, it may not be safe for PolyTM to permanently disable an arbitrary thread: for instance, a web server may have a single thread accepting requests. To account for such cases, in which it is impossible to know the application's semantics, we provide a library call for the programmer to forbid PolyTM from disabling a specific thread (e.g., to tune the parallelism degree). Such a thread may still be disabled temporarily to allow switching the TM algorithm, which is a brief procedure as noted above.

4.3 Adapting the Contention Management

PolyTM's optimization encompasses other configuration parameters related to contention management [31]. Specifically, PolyTM integrates a scheme for HTM [25] that considers two parameters: (i) the budget of retries using HTM for a transaction, (ii) whether, upon a capacity abort, the budget should be decreased by one, halved, or fully consumed.

In fact, different contention management policies can coexist without affecting correctness [31]. Hence, both parameters can be changed at any point without synchronization.

5. RecTM: a Recommender System for TM

RecTM optimizes PolyTM by relying 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 a base 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, we detail the building blocks of RecTM.

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), an efficient and simple technique for rating prediction [61].

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 rating of 0 to 10). The absolute KPI values produced by different TM applications, instead, can span orders of magnitude (e.g., from millions [9] to few txs/sec [32]). Further, KPIs of specific configurations provide no indication on the max/min KPI that the application can obtain, impairing their normalization.

Our Recommender tackles this issue with an innovative technique, which we call *rating distillation*. This function maps KPI values of diverse TM applications to a scale that can be fruitfully exploited by CF to identify correlations among 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 our 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):

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 (§ 5.1).
3: Selection of CF algorithm and setting of its hyper-parameters (§ 5.1).
4: Upon the arrival of a new workload (§ 5.3):
5:     Sample the workload on a small set of initial configurations (§ 5.2).
6:     Recommend the optimal configuration (§ 5.1).
```

Algorithm 3 Rating Distillation function in ProteusTM.

```
1: for $C_i \in C_1 \dots C_K$ do
2:     Normalize Matrix $KPI$ w.r.t. $C_i$
3:     Collect the vector $M_w$ with the max values per row
4:     Compute $mean_i(M_w)$ and $var_i(M_w)$
5: end for
6: Return C* $= argmin_{i \in 1 \dots M}\ var_i(M_w)/mean_i(M_w)$
```

$\begin{pmatrix} 1 & 2 & 3 \\ 30 & 20 & 10 \\ 100 & 200 & ? \end{pmatrix}$, 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, we can 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. We want 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, we show 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 used similarity functions in KNN CF are the Euclidean, Cosine and Pearson [54]. The first cannot be applied to heterogeneous ratings, because it is based on the scale-sensitive Euclidean distance: in the example above, it would incorrectly regard C_2 as more similar to C_3 than C_1. The other two are scale-insensitive, so they identify C_1 as similar to C_3. However, they would yield an incorrect prediction in absolute value, as it will lie on C_1's scale, which is different from C_3's[3].

A similar shortcoming applies to MF CF. The P and Q matrices — recall §2.2 — are typically obtained via Stochastic Gradient Descent [54]: starting from random matrices, this technique iteratively tries to minimize the fitting error of $P^T Q$ over A. Thus, it is prone to over-fitting around the highest absolute value ratings, yielding poor accuracy.

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

Note that feature normalization is often performed in Machine Learning (ML): the most notable example is in Artificial Neural Networks, which normalize input features in the range [0,1] [5]. In ML, however, normalization is performed on the input features, whose values are fully known for samples in the training set and for queries. In contrast, in ProteusTM, the normalization has to be performed on the UM, which contains values of the output feature KPI, and whose entries are not all known. Next, we describe how ProteusTM normalizes ratings to meet the two aforementioned requirements and, thus, enables CF to optimize TM applications.

Normalization in the Recommender. If the minimum and maximum KPIs of an application were known *a priori*, they could be mapped to a homogeneous scale with a simple, per workload, normalization. Since KPIs of applications can take arbitrary values, then this *ideal* solution cannot be used.

The rating distillation used by the Recommender approximates the ideal approach with 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: $D(M_w) = var(M_w)/mean(M_w)$ (where M_w is defined below).

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.

We define this function in Algorithm 3. The rating of each row is obtained by normalizing its KPI with respect to a column C*$\in \{C_1 ... C_K\}$ (assuming there are K configurations), so to minimize the index of dispersion among the resulting maximum ratings in the normalized domain.

Note that not only does this function reduce the numerical heterogeneity of ratings; it also projects all the elements of the matrix to a semantically common domain: now, a rating k for configuration i can be seen as "configuration i delivers performance that are k times the reference one". While an absolute throughput of 5K txs/sec may correspond to either a good or a bad performance depending on the application, our rating function gives ratings a "more universal" meaning. Also, minimizing the dispersion of the maximum values allows to align the upper extreme of the rating distributions of each application (i.e., matrix row) to a common value: the tighter the distribution around a common value M_w, the closer it approximates an ideal "omniscient" normalization.

Tuning the Recommender. We used Mahout [51], a ML framework containing several CF algorithms. This design choice allows the Recommender to seamlessly leverage a vast library of techniques, rather than binding to a single one.

The Recommender uses the training UM to choose one of the available CF algorithms, to adopt at run-time, and properly tunes its parameters (e.g., similarity function). Deter-

[3] This reasoning applies for user-based KNN. Item-based KNN is not suitable for our domain, as it expresses any unknown rating for a ⟨u,i⟩ pair as a *weighted average* of the ratings already provided by u itself. Hence, it cannot predict any value outside the range already witnessed by u itself.

mining the best algorithm and its hyper-parameters, given a training set, is a challenge that falls beyond the domain of CF [3]. In our Recommender, we use an approach based on random-search [4] and n-fold cross-validation [6, 37, 62].

5.2 Controller: Bayesian Workload Exploration

The Controller uses Sequential Model-based Bayesian Optimization (SMBO) [37] to drive the 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. Our Controller uses as acquisition function the criterion of *Expected Improvement (EI)* [40], 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 [40]. High EI values are associated either with points that are regarded by the model as likely to be the minimum (high predicted mean), or with points whose corresponding value of the target function the model is uncertain about (high predicted variance). By selecting as next point for evaluation the one that maximizes the EI, SMBO naturally balances exploitation and exploration: on one side it exploits model's confidence to sample the function at points that are supposedly good candidates to be the minimum; on the other, it explores zones of the search space for which the model is uncertain, to increase its predictive power by iteratively narrowing uncertainty zones.

Computing $\mathbf{p}_M(\mathbf{c}|\mathbf{x})$. 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 [37] and other optimization techniques [50] 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 [40].

More in detail, the Controller builds a *bagging* ensemble [7] of k CF learners, each trained on a random subset of the training set. Then, it computes μ_x as the average of the values output by the single predictors, and σ_x^2 as their variance. In ProteusTM, we use 10 bagged models; the cost of employing them instead of a single one is negligible, mainly because they are only queried during profiling phases.

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

Our 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. More precisely, the exploration is terminated after k steps when: (i) the EI decreased in the last 2 iterations; (ii) the EI for the k-th exploration was marginal, i.e., lower than ϵ with respect to the current best sampled KPI; (iii) the relative performance improvement achieved in the $k-1$-th iteration did not exceed ϵ. In §6.3, we evaluate the effectiveness of this policy.

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 KPI from the mean value observed in recent time windows [2]. This allows the Monitor to detect both abrupt and smooth changes and to trigger a new profiling phase in our Controller. Note that environmental changes (e.g., inter-process contention or VM migration) are indistinguishable from workload changes from the perspective of our behavior change detection.

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 2: Machines used in our experimental test-bed.

6. Evaluation

This section provides an extensive validation of our contributions. We introduce, in §6.1 the test-bed, applications, and accuracy metrics used. In §6.2 we assess the overhead incurred by PolyTM to provide self-tuning capabilities. In

Benchmark	Lines of Code	Atomic Blocks	Description
STAMP [9]	28803	35	Suite of **8** heterogeneous benchmarks with a variety of workloads (genomics, graphs, databases).
Data Structures	3702	12	Concurrent Red-Black Tree, Skip-List, Linked-List and Hash-Map with workloads varying contention and update ratio.
STMBench7 [32]	8623	45	Based on OO7 [10] with many heterogeneous transactions over a large and complex graph of objects.
TPC-C [63]	6690	5	OLTP workload with in-memory storage adapted to use one atomic block encompassing each transaction.
Memcached [58]	12693	120	Caching service with many short transactions that are used to read and update the cache coherently.

Table 1: TM applications used in our evaluation. These 15 benchmarks span a wide variety of workloads and characteristics.

Machine ID	TM Backend	# threads	HTM Abort Budget	HTM Capacity Abort Policy
Machine A	STMs and TSX [67]	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 3: Parameters tuned by ProteusTM. STMs are TinySTM [29], SwissTM [27], NORec [15] and TL2 [22].

§6.3, we evaluate the effectiveness of RecTM's components separately. Finally, in §6.4 we evaluate the ability of ProteusTM to perform optimization of dynamic workloads.

6.1 Experimental Test-Bed

We deployed ProteusTM in two machines with different characteristics (described in Table 2) and used a wide variety of TM applications (summarized in Table 1). We considered over 300 workloads, which are representative of heterogeneous applications, from highly to poorly scalable, from HTM to STM friendly [26]. Moreover, we tested three KPIs: execution time, throughput and EDP (Energy Delay Product, a popular energy efficiency metric [36]). We measure energy consumption via RAPL [17] (available on Machine A).

Our system optimizes the KPI by tuning the four dimensions listed in Table 3[4]. Overall, we consider a total of 130 TM configurations for Machine A and 32 for Machine B.

Evaluation metrics. We evaluate the performance of ProteusTM along 2 accuracy metrics: Mean Average Percentage Error (MAPE) and Mean Distance From Optimum (MDFO).

Noting $r_{u,i}$ the real 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** is: $\sum_{\langle u,\cdot\rangle\in S} |r_{u,i^*_u} - r_{u,\widehat{i^*_u}}|/r_{u,i^*_u}$.

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

6.2 Overhead Analysis and Reconfiguration Latency

We now assess the overhead of PolyTM, i.e., the inherent steady-state cost of supporting adaptation. We compare

[4] ProteusTM also includes HybridTMs: we omit them as HybridTMs never outperformed STMs/HTMs (similarly to recent work [26]).

#threads	TL2	NOrec	Swiss	Tiny	HTM-opt	HTM-naive
1	3	3	2	3	3	14
4	< 1	1	< 1	3	3	14
8	< 1	< 1	< 1	4	5	24

Table 4: 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 5: Reconfiguration (TM and #threads) latency (μsec).

the performance of a bare TM implementation T with that achieved by PolyTM using T without triggering adaptation.

Table 4 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 [41, 67]). We also show the overhead of the optimized code path, employed for HTM, and the one resulting from the default GCC instrumentation (fully instrumented path).

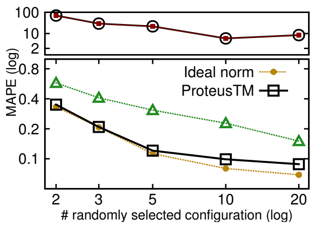
These experiments reveal overheads consistently $< 5\%$ across the TM backends. The lower STM overhead is justifiable considering that STMs natively suffer from instrumentation costs that end up amortizing most of the additional overhead introduced by PolyTM.

We also assess 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 5, encompass two heterogeneous workloads: Memcached uses $100\times$ shorter transactions than TPC-C. The results highlight the practicality of our reconfiguration. Even in the worst case of large transactions in TPC-C, the latency is small. In fact, this is only incurred during the exploration phase, which, as we shall see, is very short with ProteusTM.

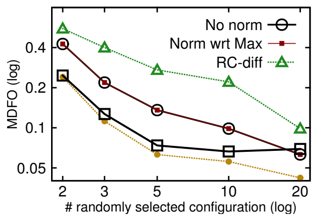
6.3 Quality of the Prediction and Learning Processes

We now evaluate each of RecTM's components by means of a trace-driven simulation. We collected traces of *real* executions of 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



(a) MAPE, KNN Cosine



(b) MDFO, KNN Cosine

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

algorithm (§5.1) and to instantiate the predictive model. We used 10 learners for the bagging ensemble, as this is a typical value [37, 62]. 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. We assess the effectiveness of the distillation function versus several UM preprocessing techniques: *(i) No normalization:* CF is applied on the UM containing raw KPI samples. This is equivalent to Quasar [21] (see §7); *(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 [20] (see §7), where the machine's peak instructions/sec rate is used as normalizing constant; *(iii) Ideal normalization:* the scheme described in §5.1; *(iv) Row-column subtraction:* noted RC, is typically employed in CF to cope with biases in ratings [54]. 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 (Algorithm 3).

For space constraint we show only a subset of results, focusing on execution time on Machine A with KNN and cosine similarity. We vary the number of *randomly chosen* known ratings per row and compute MAPE and MDFO.

Figure 4 shows that using no normalization, or normalization w.r.t. the maximum performs very poorly, both in terms of MAPE (Figure 4a) and MDFO (Figure 4b). That is because they are both performing a normalization with respect to some constant that has no meaning in the scope of the applications used. RC achieves lower MAPE than the two aforementioned normalizations, yet its accuracy is significantly worse than that of rating distillation, both in terms of MAPE and MDFO. Also, the approach of ProteusTM closely follows the ideal normalization. To ensure a fair comparison, we used the same training set, without forcing the presence of the column used for normalization among the profiled configurations for ProteusTM.

We have obtained other similar results, omitted due to space constraints, with other distance functions in KNN and MF (which is used by other proposals that rely on a RS for performance prediction, see §7) . Our results confirm the key role of rating distillation to enable the use of CF in the domain of performance prediction for TM applications.

Controller. We evaluate the effectiveness of our SMBO approach to the sampling of new workloads. We compare our solution (called EI) with a randomized sampling approach, used in Quasar and Paragon [20, 21] (see §7), and two other SMBO approaches using acquisition functions different from ours: Variance explores configurations with high uncertainty for the underlying model (i.e., high $variance/mean$ ratio); Greedy explores the configuration with highest predictive mean.

Our 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 with c^* and the MAPE is an average of MAPEs computed per workload.

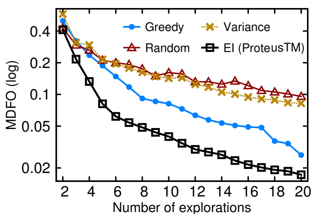
Due to space constraints we show only a subset of results. In Fig. 5a, we report the MDFO for EDP (on Machine A). The EI exploration policy is able to identify a high quality solution requiring, on average, less explorations than any competitor. Fig. 5b shows that the 80-th percentile of the DFO obtained by EI — after 5 explorations — is less than 10%. We highlight that the EDP KPI was the most challenging to optimize: hence, the latter result represents a lower bound on our accuracy.

In Fig. 5d, we show the MDFO when optimizing execution time (on Machine B): once again, our 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).

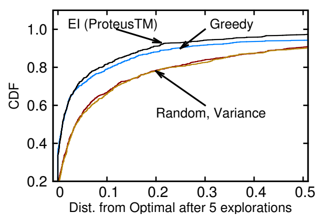
Finally, we compare our EI policy with random sampling in Figs. 5a and 5d: taking 5% distance as reference, EI achieves a number of explorations vs MDFO trade-off that is up to $4\times$ better than its competitor. This highlights the effectiveness of our SMBO-based approach over simpler sampling techniques used in recent systems [20, 21] (see §7).

Stopping criterion. We now evaluate our stopping heuristic (§5.2), called *Cautious* (C in the plots). We compare it with a *Naive* stopping rule N that blindly trusts the model, by stopping explorations when the expected improvement over the best known configuration falls $< \epsilon$. The results are shown in Fig. 6, portraying the sensitivity of both heuristics to ϵ.

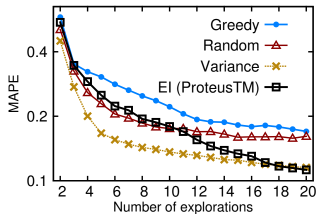
For any fixed ϵ, we observe that the Naive predicate consistently chooses a worse configuration than the one of our



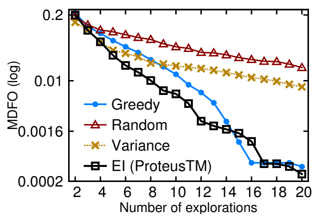
(a) MDFO for EDP



(b) CDF of DFO for EDP



(c) MAPE for Exec. Time



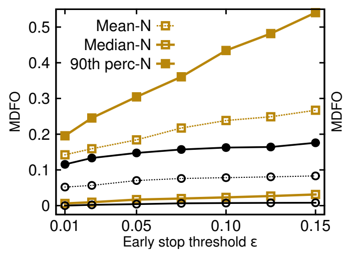
(d) MDFO for Exec. Time

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

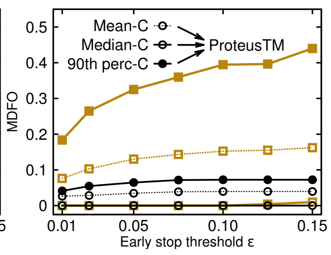
Cautious heuristic: blindly trusting the predictive model results in an excessively eager policy, which does not provide the model with enough training data.

As expected, the plots also show that the lower ϵ, the lower the obtained MDFO. Notably, for $\epsilon = 0.01$, the Controller achieves, in 90% of the cases, MDFO of only 5% when considering execution time on and 12% when optimizing EDP. This comes at the price of a higher number of explorations. Although not shown for space constraints, we report that the Controller is able to keep this price very low, by requiring, on average, a similar number of explorations of a policy that performs a fixed amount of explorations and is tuned to deliver the same mean performance. This confirms the effectiveness of the Controller in determining the duration of the profiling, by striking a balanced trade-off between the extent of online exploration and final performance.

Comparison with ML approaches. We now compare ProteusTM with an approach based on the same technique proposed by Wang et. al [65] 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: duration of transactions, data access patterns, data contention, etc. Wang et al. also uses static analysis to for other features, e.g., the number of atomic blocks. We did not perform this step but complemented it with contention management features. These are not considered by the authors, but we found them to be highly correlated with performance.



(a) MDFO for EDP, Machine A



(b) MDFO for Exec time, Machine B

Figure 6: Comparing early-stop exploration predicates.

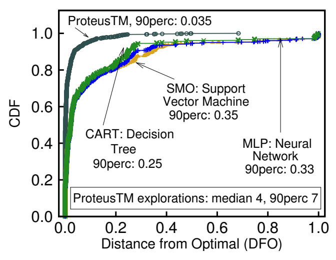
The simulation for ProteusTM evolves as explained above. For the ML competitor, instead, a workload is first profiled over a reference configuration (TinySTM, 4 threads) and then the ML is invoked to predict the best configuration. We then compute the MDFO for this chosen configuration.

We used 300 STAMP and Data Structures workloads, on Machine A, and split them randomly into training and test sets: 30-70 and 70-30 train-test splits. For ProteusTM, the training set is the UM of 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.

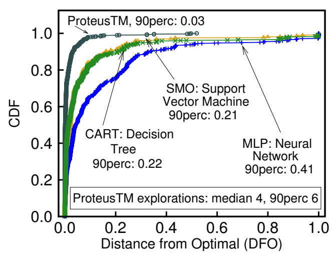
We consider 3 ML algorithms, implemented in Weka [33]: Decision Trees (CART), Support Vector Machines (SMO), and Artificial Neural Networks (MLP) [5]. Their parameters were chosen via random search optimization [4], which evaluated 100 combinations with cross-validation on the training set.

Fig. 7 reports the CDF of the DFO of each technique over 10 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).

We note that the DFO of ProteusTM is similar (both in mean and 90^{th} percentile) in both cases, whereas ML

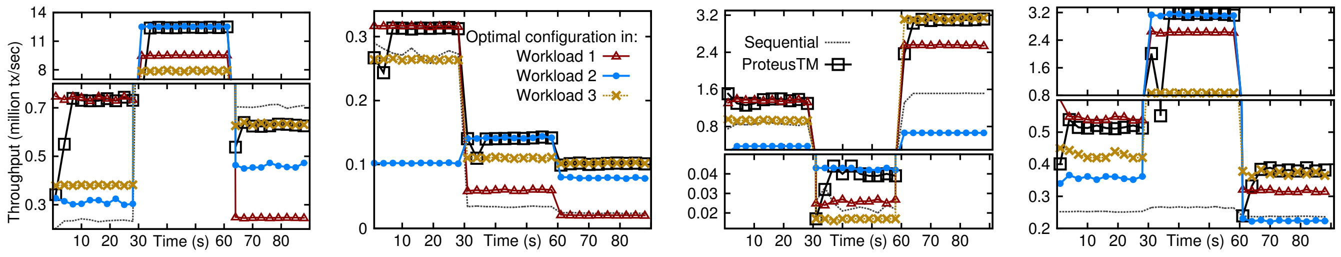


(a) 30% Training data used.



(b) 70% Training data used.

Figure 7: Comparison of ProteusTM vs various Machine Learning based techniques.



(a) Red-Black Tree (RBT) (b) STMBench7 (c) TPC-C (d) Memcached

Figure 8: Performance of four applications when their workload changes three times. We show the performance obtained with ProteusTM, and three additional fixed configurations, each one corresponding to an optimum in each workload.

greatly benefits from more training data. This difference is explained by the number of explorations used by ProteusTM in its profiling phase (with threshold $\epsilon = 0.01$): 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.

Our evaluation suggests that detecting similarities on the KPI is more effective than statistically inferring relationships from training data. We argue that this depends on two, tightly intertwined, causes: (i) thanks to our 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. 8, we evaluate the ProteusTM system as a whole[5]: 2 TM benchmarks (Red-Black Tree and STMB7 [32]) and 2 TM ports of TPC-C [63] Memcached [58]. For each, we trigger 3 workloads chosen to exemplify contrasting performances. In each case, ProteusTM is totally oblivious of the target application: no workloads of the application are in its training set. This highlights the Recommender's ability to detect similarity patterns between the target workloads and the set of *disjoint* workloads used as training set. We set the Monitor period to 1 sec and the SMBO ϵ to 0.01. In each run, we 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.

We draw three conclusions: (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 2 orders of magnitude worse performance; (iii) thanks to our SMBO approach, the performance degradation 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 [13].

[5] The source materials can be found in `https://github.com/nmldiegues/proteustm`.

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

Table 6: For each benchmark in Fig. 8, we show the MDFO of ProteusTM, each Optimal and BFA (★) configurations. Each workload is labeled with its optimal configuration.

A summary is provided in Table 6 where we list the optimal configurations in each workload. We also show the BFA (with ★) which is always also an optimal configuration in some workload. This data highlights the robustness of ProteusTM to optimize applications with diverse optimal configurations, in terms of TM algorithm (STMB7), parallelism degree (TPC-C) and HTM tuning (RBT and Memchd).

Finally, in Fig. 9, we confirm our claims of §5.3 by using a static TPC-C workload and varying external factors to the application to trigger behavior changes. To simulate these external changes we used the *stress* Unix tool with different configurations over periods of 30 seconds: it either created high CPU, memory or IO usage in each workload. The results are once again positive, in that ProteusTM performs close to the optimal configuration across all workloads.

7. Related work

Our work lies at the intersection of 3 major research fields: optimization of TM systems, performance prediction via RS, and experiment-driven optimization of computer systems.

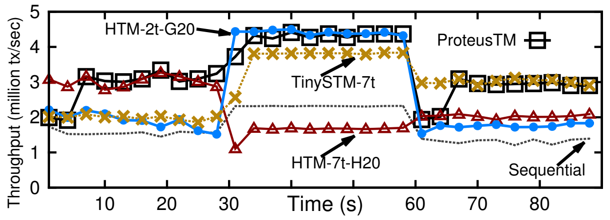


Figure 9: Similar to Fig. 8c, but with a static workload, varying instead the availability of machine resources.

Optimization of TM applications. The most researched problem in TM self-tuning is choosing the number of active threads. Proposed solutions rely on analytical modeling [60], off-line machine learning [59], or exploration-based strategies [24]. Wang et al. [65] use Artificial Neural Networks and programmer heuristics to determine the best TM for an application (excluding HTM). Workload characterization uses static analysis and runtime detailed profiling. Tuner [25] exploits hill climbing and reinforcement learning to adaptively determine the retry-on-abort policy for HTM.

These solutions optimize a single aspect of TM. Moreover, purely exploration-based solutions [24, 25] are impractical in high dimensional spaces, as the number of configurations to explore rapidly grows with the parameters to optimize. ProteusTM, instead, is effective in optimizing in a large space (we considered 130 configurations over 4 parameters). Thanks to the capability of CF techniques to deal with sparse information and to our model to steer online explorations, ProteusTM operates in even higher dimensional problems by simply including more configurations as extra columns of the UM.

In addition, all surveyed solutions need either preliminary code analysis or intrusive instrumentation for workload characterization. These add complexity to the code of TM algorithms and overhead to the application. ProteusTM, conversely, relies solely on profiling high-level KPIs, which incurs minimal overhead and maximizes portability. We stress that ProteusTM's work-flow is fully automated, avoiding the need for programmer heuristics [65]. Finally, ProteusTM avoids off-line training on the target application, unlike other ML approaches [59].

Performance prediction via Recommenders. To the best of our knowledge, Paragon [20], Quasar [21] and U-CHAMPION [53] are the only systems relying on RS for performance prediction, job scheduling and resource provisioning. In common, they characterize an incoming job via *random* sampling of a *fixed* number of configurations and then apply MF-based CF.

ProteusTM differs from these works in three key aspects: (i) it relies on a novel rating distillation function that identifies similarity patterns among the performances of heterogeneous applications (one noteworthy finding of our work is that this pre-processing step, not used in previous works, is of paramount importance to achieve high accuracy in the TM domain); (ii) ProteusTM leverages model-based techniques to determine which and how many configurations to experiment with during the sampling phase of a new workload (we showed that it outperforms random sampling), yielding lower sampling time and higher accuracy; (iii) it integrates both MF- and KNN-based CF, being able to determine the best one to employ, depending on the training data.

Experiment-driven optimization. The classic approach in this field is dynamic sampling: a performance model of the system is initialized, by evaluating its performance corresponding to some randomly chosen configurations; then, the next experiment to run is chosen according to the observations progressively collected [28, 50, 66, 68]. The proposal that is closest to ours is iTuned [28], which optimizes configuration parameters of a database for a target application. It exploits Gaussian processes [55] to build a performance model of the application and uses the EI acquisition function to choose the next experiment to run. However, iTuned does not use knowledge of previous the workloads: by leveraging CF, ProteusTM alleviates the need for the initial static sampling phase, and is able to perform accurate recommendations after a short online adaptive sampling phase.

8. Conclusions

We proposed ProteusTM, the first TM system with multi-dimensional self-tuning capabilities. ProteusTM has been integrated in GCC to expose a standard TM interface, which ensures full transparency, ease of use and portability. Together with this simple abstraction, we provide high performance by relying on a novel technique that leverages Collaborative Filtering and Bayesian Optimization.

Via an extensive evaluation based on a real-word application and well-known benchmarks, we demonstrated the ability ProteusTM 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.

As a final remark, we believe that the methodologies underlying the RecTMdesign could be applicable to optimize the self-tuning of broader and more generic classes of systems than just TM. In particular, its novel ability to normalize the data for the RS, together with the Bayesian-driven exploration of workloads to sample.

As a final remark, we believe that the methodologies underlying RecTM— namely, the techniques of rating distillation and the Bayesian approach to drive the workloads' sampling — can be used to optimize the self-tuning of a broader, more generic class of systems, than just TM.

Acknowledgements

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and EXPL/EEI-ESS/0361/2013.

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 Annual Design Automation Conference*, DAC, pages 1–6, 2014.

[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, R. Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Proceedings of the Annual Conference on Neural Information Processing Systems*, NIPS, Granada, Spain, 2011.

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

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

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2007.

[7] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[8] 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.

[9] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, IISWC, 2008.

[10] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The oo7 benchmark. *SIGMOD Rec.*, 22(2):12–21, June 1993.

[11] 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.

[12] Mrcio Castro, LusFabrcioWanderley Ges, LuizGustavo Fernandes, and Jean-Franois Mhaut. Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications. In *Proceedings of the European Conference on Parallel Processing*, Euro-Par, pages 465–476. 2012.

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

[14] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52, 2011.

[15] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 67–78, 2010.

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

[17] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED, pages 189–194, 2010.

[18] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, pages 33–48, 2013.

[19] 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 ACM Conference on Recommender Systems*, RecSys, pages 293–296, 2010.

[20] 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.

[21] 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.

[22] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the International Conference on Distributed Computing*, DISC, pages 194–208, 2006.

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

[24] 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.

[25] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *Proceedings of the USENIX International Conference on Autonomic Computing*, pages 209–219, Philadelphia, PA, 2014.

[26] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation*, PACT, pages 3–14, 2014.

[27] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching Transactional Memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 155–165, 2009.

[28] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[29] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 237–246, 2008.

[30] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management. In *Proceedings of the International Conference on Distributed Computing*, DISC, pages 303–323, 2005.

[31] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC, pages 258–264, 2005.

[32] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems*, EuroSys, pages 315–324, 2007.

[33] 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.

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

[35] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, pages 289–300, 1993.

[36] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 8–11, Oct 1994.

[37] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-based Optimization for General Algorithm Configuration. In *Proceedings of the International Conference on Learning and Intelligent Optimization*, LION, pages 507–523, 2011.

[38] 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.

[39] 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, 2012.

[40] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, 13(4):455–492, December 1998.

[41] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 476–487, 2014.

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

[43] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proceedings of the VLDB Endownment*, 5(4):298–309, December 2011.

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

[45] Greg Linden, Brent Smith, and Jeremy York. Amazon.Com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[46] 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 ACM SIGOPS European Conference on Computer Systems*, EuroSys, pages 41–54, 2010.

[47] Alexander Matveev and Nir Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 11–22, 2013.

[48] Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 103–112, 2013.

[49] 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 ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA, pages 195–212, 2008.

[50] Takayuki Osogami and Sei Kato. Optimizing System Configurations Quickly by Guessing at the Performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 145–156, 2007.

[51] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[52] 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.

[53] Eric Pettijohn, Yanfei Guo, Palden Lama, and Xiaobo Zhou. User-Centric Heterogeneity-Aware MapReduce Job Provisioning in the Public Cloud. In *Proceedings of the Inter-*

national Conference on Autonomic Computing, ICAC, pages 137–143, 2014.

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

[55] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2005.

[56] Carl Ritson and Frederick Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *Proceedings of Communicating Process Architectures*, CPA, pages 271–292, 2013.

[57] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is Transactional Programming Actually Easier? *SIGPLAN Not.*, 45(5):47–56, January 2010.

[58] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 399–412, New York, NY, USA, 2014. ACM.

[59] 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.

[60] Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Regulating Concurrency in Software Transactional Memory: An Effective Model-based Approach. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO, pages 31–40, 2013.

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

[62] 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 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, pages 847–855, 2013.

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

[64] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore Inmemory Databases. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, pages 18–32, 2013.

[65] 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.

[66] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A Smart Hill-climbing Algorithm for Application Server Configuration. In *Proceedings of the International Conference on World Wide Web*, WWW, pages 287–296, 2004.

[67] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19. ACM, 2013.

[68] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-based Management of Virtualized Data Centers. In *Proceedings of the Conference on USENIX Annual Technical Conference*, ATC, pages 18–18, Berkeley, CA, USA, 2009. USENIX Association.