Identifying the Optimal Level of Parallelism in Transactional Memory Systems*

Diego Didona[1], Pascal Felber[2], Derin Harmanci[2], Paolo Romano[1], and Jörg Schenker[2]

[1] Instituto Superior Técnico/INESC-ID, Portugal.
[2] University of Neuchâtel, Switzerland.

Abstract. In this paper we investigate the issue of automatically identifying the "natural" degree of parallelism of an application using software transactional memory (STM), i.e., the workload-specific multiprogramming level that maximizes application's performance. We discuss the importance of adapting the concurrency level to the workload in two widely different scenarios, a *shared-memory* and a *distributed* STM infrastructure. We propose and evaluate two alternative self-tuning methodologies, explicitly tailored for the considered scenarios. In shared-memory STM, we show that lightweight, black-box approaches relying solely on on-line exploration can be extremely effective. For distributed STM settings, we introduce a novel hybrid approach that combines model-driven performance forecasting techniques and on-line exploration in order to take the best of the two techniques, namely enhancing robustness despite model's inaccuracies, and maximizing convergence speed towards optimum solutions.

1 Introduction

The pervasive adoption of multi-core architectures (from HPC clusters to embedded systems) has raised the urge to identify paradigms capable of simplifying the development of parallel applications. Transactional memory (TM) [2] has garnered a lot of interest of late precisely because, thanks to its simplicity and scalability, it appears to be a promising alternative to classic lock-based synchronization. Over the last years, a wide body of literature has been published on TM, and several variants have been developed, including hardware-based (HTM), software-based (STM), and distributed (DTM) [3]. One of the key results highlighted by existing research is that, independently of the nature of the synchronization scheme adopted by a TM platform, its actual performance is strongly workload dependent and affected by a number of complex, often intertwined factors (e.g., duration of transactions, level of data contention, ratio of update vs read-only transactions).

Among these numerous factors, an often neglected one is the concurrency level used by the application. However, as we will also quantitatively show in

* Part of this work has been presented in a non-archival workshop [1].

this paper, the identification of the "right" level of concurrency represents a critical factor that can have a strong impact on performance of TM applications. Unfortunately, this decision is far from being trivial, as the off-line tuning of this parameter is a costly and error-prone process. Further, any static configuration can lead to suboptimal performance in presence of dynamic workloads. Hence, we argue that work done by the TM programmers to develop parallel applications risks to be wasted, unless effective mechanisms are available to tune the concurrency level of TM applications, and let them take full advantage of the underlying parallel architecture.

In this paper, we address the problem of self-tuning the concurrency level according to the application workload (which we call "elastic scaling") in various application settings. Related problems have been addressed previously but limited attention has been devoted to dynamically identifying the optimal degree of parallelism for a (D)TM platform, namely the degree of local (i.e., number of active threads) and possibly global (i.e., number of nodes in a DTM) concurrency that maximizes the throughput of complex (D)TM applications.

We present experimental results obtained considering two extreme scenarios: on shared-memory systems with a low-level STM library written in C, and in distributed systems with a high-level DSTM infrastructure written in Java. We first show that realistic benchmarks exhibit widely different performance depending on the degree of parallelism, and that adapting the number of threads at runtime can improve performance of some applications over any execution with a fixed number of threads. By applying small modifications to the benchmarks and the underlying STM runtime in a shared-memory system, one can straightforwardly optimize the concurrency level using exploration-based on-line optimization techniques, e.g., using hill climbing or gradient descent algorithms.

In distributed settings, however, the cost of testing configurations with a different number of threads (i.e., nodes) is prohibitive, as it requires transferring state, generates additional traffic, and takes orders of magnitude more time than in centralized settings. Therefore, in such settings, one should instead rely on modeling techniques to predict the expected gains from adding or removing nodes for adapting the concurrency level. However, performance modeling techniques are unavoidably subject to approximation errors, which can lead to the identification of suboptimal configurations. We show how this problem can be tackled by introducing a novel self-tuning methodology that combines exploration-based and model-driven approaches: models help predicting the evolution of performance at a large scale, while local, inexpensive exploration can be used to gather feedback on the model's accuracy and allow its progressive enhancement. To this end, we show how machine learning techniques can be exploited to learn corrective factors aimed at "patching" the output of performance models and correcting biases/approximation errors that may otherwise impair their accuracy.

The rest of this paper is organized as follows. We first give a brief overview of related work in Section 2. We present, in Section 3, our on-line exploration-based approach for shared-memory STM systems. In Section 4, we focus on

distributed STM systems, and present a hybrid self-tuning approach combining on-line exploration and model-driven optimization techniques. We finally conclude in Section 5.

2 Related Work

Adapting the concurrency level to system workload (elastic scaling) in order to improve performance and/or resource usage is an issue that has already been addressed in previous research. Different approaches to the problems exist, especially depending on the area where the adaptation is performed.

Part of the research targets concurrent execution on a single machine. Reimer et al. [4] propose to adapt the concurrency level of parallelizable portions of a scientific application. This approach does not allow changing the concurrency level during the execution of the code portion, whereas with our exploration-based approach workload changes within such portions can be tracked and concurrency levels can be adapted. Heiss and Wagner [5] study the tuning of the concurrency level (number of concurrently running transactions) within a transactional processing system (e.g., database server) running on a single machine. They propose two algorithms, one of which is a hill climbing approach similar to ours. Schroeder et al. [6] use a feedback control loop that is initialized with a close-to-optimal value thanks to the use of queueing theoretic models. This initialization allows the approach to converge fast under abrupt workload changes. Abouzour et al. [7] propose a hybrid approach merging these two studies. Our work differs since we adapt the concurrency level of threads within an arbitrary application, while other approaches tune the concurrency level of transactions only in the restricted context of a transactional processing system.

Few studies exist for the adaptation of the concurrency level for TM-based applications (using non-replicated TM). Yoo and Lee [8] propose rescheduling threads in order to reduce contention due to data conflicts. Such a technique has the effect of adapting the concurrency level of the application, because rescheduled threads are removed from the execution for a while. Ansari et al. [9] aim to improve application efficiency by reducing resource usage without sacrificing performance. To achieve this objective, they maintain the transaction abort rate under a predefined threshold. Our approach differs from this work as (i) we do not use any fixed thresholds, and (ii) we focus on optimizing transaction throughput, allowing us to improve application performance (especially under changing workloads) compared to an execution with any fixed concurrency level.

Elastic scaling in distributed settings corresponds to automatically adapting, in face of varying workloads, the number of nodes the platform is deployed onto. In the area of replicated relational databases, several mechanisms have been proposed [10, 11] to tackle this problem. Our work proposes a solution specialized for (D)TM platforms. A distinguishing aspect of our work is that it identifies the level of concurrency that maximizes the throughput of complex (D)TM applications by considering scaling at two levels: (i) number of nodes of the platform, and (ii) number of active threads running on a node.

In the area of performance modelling of STM, existing literature can be subdivided in solutions based either on analytical techniques [12, 13], or on statistical methods [14, 15]. Solutions based on analytical models have a good extrapolation power, namely they typically exhibit good accuracy even in workload/scale scenarios not previously explored. However, their accuracy can degrade significantly with scenarios that challenge the set of assumptions they rely on to ensure mathematical treatability. Statistical methods, due to their black box nature, suffer of limited extrapolation, but can achieve typically very accurate predictions in regions of the state space close to those already observed during the learning phase. The solution proposed in Section 4 aims at combining the advantages of both approaches. On the one hand, it relies on analytical performance models to achieve high extrapolation. On the other hand, it exploits feedback collected from deployed DTM system to learn, using machine learning techniques, a corrective function aimed at fixing possible inaccuracies of the analytical model.

3 Shared-memory STM

Ideally, it is desired to run applications at their natural degree of parallelism, i.e., a point where each thread does "sufficient" useful work without inducing "too much" contention. The exact definition of both quantities varies depending on the context and it is generally not obvious to find this natural degree of parallelism for a given application. For workloads where contention due to data synchronization does not change throughout the application execution, the best level of parallelism can be found offline by repeatedly restarting the application with different sets of parameters. However, the contention a workload generates may vary during the lifetime of the application, i.e., the natural degree of parallelism represented by the workload varies as the application executes. Hence, a general solution to this problem would need to track the workload generated by the application on-line. In this section, we first motivate why adaptivity is important. We then describe our expiration-based mechanisms for adjusting the degree of parallelism. Finally, we show how these mechanisms can help optimize throughput according to the dynamic properties of the workload

3.1 The Need for Adaptivity

Before dwelling on the actual exploration algorithm, let us briefly consider the benefits of such adaptive techniques on the application `intruder`, part of the widely used STAMP benchmark suite [16]. This application emulates a signature-based network intrusion detection system and exhibits a workload that evolves over time.

Figure 1 indicates the performance of the benchmark when executed with varying number of threads (dashed line), or when dynamically changing the number of threads (plain straight line corresponding to a constant value). One can observe that performance is significantly better when dynamically adapting concurrency than with any fixed number of threads. Note that the experiment

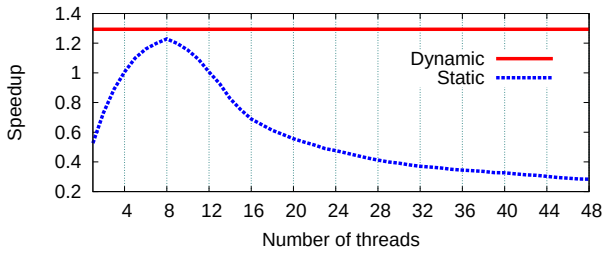


Fig. 1: Speedup of the `intruder` benchmark as compared to sequential (non-STM) version, using static and dynamically evolving numbers of threads.

was run on a 48-core machine, i.e., the number of physical cores was not the limiting factor.

3.2 An Exploration-based Approach

Our exploration-based approach performs on-line monitoring of key performance metrics. It allows us to find the natural degree of parallelism of an application by running it with an iterative algorithm controlling its concurrency level. The algorithm terminates when all the work to be performed by the application is accomplished. Each iteration of the algorithm has three phases, as illustrated in Figure 2:

- **Measurement phase:** In this phase, the application runs with a fixed number of threads. Key performance metrics (numbers of commits and aborts) are measured during a certain time period. The commit rate gives an indication of raw transaction throughput, while the abort rate is a good measure of contention.
- **Decision phase:** In this phase the algorithm decides between two actions: increasing or decreasing the number of threads. If the last measurement phase shows improvements in terms of commit rate, the action performed in the previous iteration is repeated (addition or removal of threads); otherwise, it is reversed. The decision taken in this phase corresponds to a *hill climbing* technique maximizing transaction throughput, i.e., commit rate. The technique explores configurations in the vicinity of the current one by dynamically adding or removing threads, until a (local) maximum is reached. Even when reaching such a point, the configuration is tested for adapting to possible variations in the workload that would shift the optimal configuration(s).[3]

[3] One should note at this point that none of the benchmarks we experimented with (STAMP applications and various micro-benchmarks) exhibits multiple maxima when observing throughput as a function of the number of threads, up to the hardware limit of our 48-core test machine.

– **Transition phase:** An external controller thread adds or removes threads to/from the application according to outcome of the decision phase.

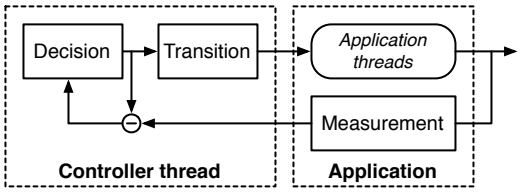


Fig. 2: The principle of the exploration-based algorithm is akin a feedback control loop. The three phases are shown in rectangles with solid lines.

For faster adaptation to the workload, we tune the duration of the measurement phase such that we have sufficiently many samples (i.e., commits) to take sound decisions but without wasting too much time. In this way, the algorithm reacts fast by quickly collecting measurements with applications composed by short transactions while it will take more time to adapt, but will still take correct decisions, for applications with long transactions.

Inserting the application inside the iterative algorithm required us to introduce (i) code observing performance, for the measurement phase, and (ii) a *controller thread* that performs decision and transition phases to modify the parameters of the application based on the measured performance. This extra thread controls the main execution loop of the application and can add or remove as many transactional threads as required during run time.

3.3 Performance in Centralized Systems

We give a brief overview of the performance of our exploration-based approach on a 48 core machine with four AMD Opteron 6172 processors with one STAMP application. A large collection of other experimental results can be found in a companion research report [17].

Figure 3 (left) shows the behavior of the exploration-based algorithm with the `intruder` application. As one can observe, the number of threads (values averaged over 2-second periods for clarity) increases in the first half of the execution to reach 13, then drops sharply to account for changes in the workload. The last part of the execution uses only few threads, which reduces the commit throughput but limits contention and avoids most aborts.

To better understand what triggers such changes in the workload, we show in Figure 3 (right) the variations in transaction lengths, as reported by the size of the read and write sets, during the execution of `intruder`. Values are averaged over groups of 10,000 transactions and sizes are shown on a logarithmic scale. The application repeatedly executes a sequence of 3 transactions. Two of

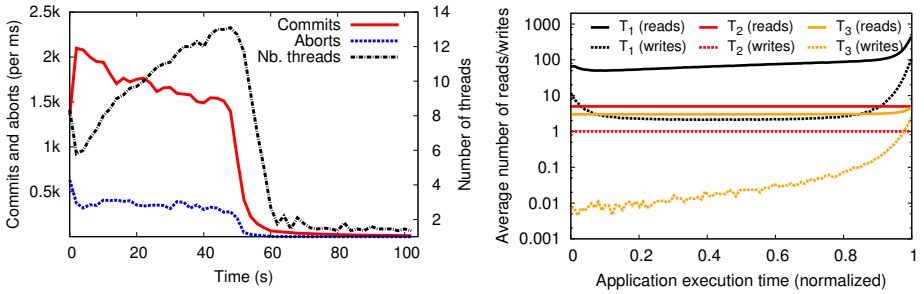


Fig. 3: Evolution of the number of threads (left) and transaction read- and write-set size (right) with the `intruder` benchmark using exploration-based scaling.

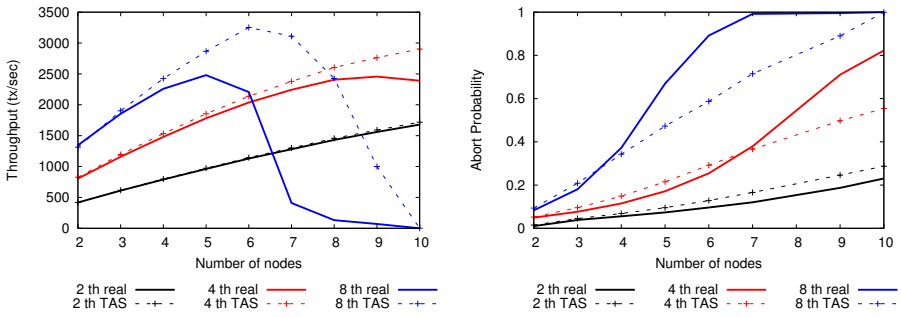
them, denoted as T_2 and T_3 in the graph, do not vary much over time. The third one, T_1, exhibits an interesting trend that explains why our approach is so effective: transactions read more and more data, with a sharp spike in the end, while their number of writes first decreases before stabilizing and increasing steeply in the end. Therefore, the last transactions to execute are very long and, hence, are expected to encounter much contention. Limiting concurrency increases the likelihood of commit and, in turn, improves overall performance. This example clearly shows the benefits of using a adaptive strategy for best tuning the concurrency degree to the varying workload properties encountered in real applications.

Note that we modified and experimented with other applications of the STAMP benchmark suite. We found out that, while `intruder` benefits most from dynamic adaptation of the concurrency level because of the wide variations in its workload, our exploration-based algorithm is also effective with other applications and can quickly find the optimal number of threads [17].

4 Distributed STM

When considering DTM systems [18], the degree of concurrency of the platform, which we call also *global multiprogramming* level, is determined not only by the number of threads deployed on each node, but also by the number of nodes composing the platform. We note that in distributed settings, purely exploration-based techniques, like the one described in Section 3, are much less effective for two main reasons. First, due to the quadratic growth of the solution's space, the number of exploration steps required to identify the optimal solution is expected to grow significantly. Second, unlike in shared memory TM systems, in DTM scenarios exploratory steps requiring altering the number of nodes in the platform require triggering state-transfer phases that can induce significant additional load [19, 20] and lead to severe performance degradation [10].

We argue, therefore, that in DTM settings approaches relying on performance models to forecast the optimal degree of concurrency of the platform are



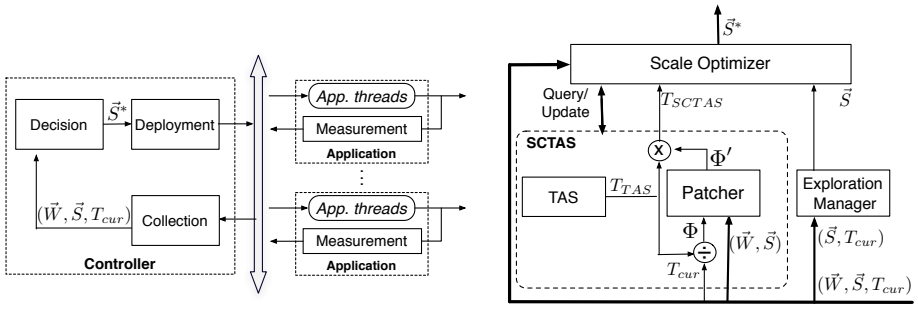
(a) Throughput (b) Abort probability

Fig. 4: Accuracy of TAS' predictions.

preferable to strategies based on pure exploration. Model-driven elastic scaling techniques are also attractive as they represent a fundamental building block for QoS-oriented provisioning schemes, in which the amount of computational resources (and their type, e.g., medium vs large instances in a IaaS platform) needs to be elastically adjusted to ensure given SLAs in face of variable loads [21–23]. On the other hand, model-based performance forecasting techniques, due to their approximate nature, rely on simplifying assumptions which can degrade significantly their accuracy in presence of challenging workloads. Further, most of these modelling techniques rely on analytical methods (e.g., queuing theory), which are *rigid*, in the sense that they cannot be "bent" to learn from the feedback gathered by the actual system and accordingly correct to enhance their accuracy. In the following we report the result of an experimental study that highlights the above mentioned issues of model-driven approaches by assessing the accuracy of a state-of-the-art performance forecasting model for DTM, namely Transactional Auto Scaler (TAS) [22].

The plot in Figure 4a shows the accuracy of TAS in predicting the throughput of a DTM application when deployed over different scales. The DTM platform used in this study is Infinispan [24], a popular in-memory distributed transactional key-value store [25]. The application running over it is a porting of the TPC-C benchmark [26], and we used, as experimental testbed, a cluster of 10 servers, each equipped with 8 cores and interconnected via a private Gigabit Ethernet. The plot highlights the ability of TAS to correctly forecast the throughput of the application when deployed over different scales, as long as the global multiprogramming level (and correspondingly the abort rate) does not grow too high. This loss of accuracy of TAS is imputable to the very high contention probability that the application exhibits in those scenarios, as reported in Figure 4b. In fact, TAS, like other analytical models of transactional systems [27, 21], relies on a set of simplifying assumptions on the modelling of transactions' conflict patterns, which can lead to significant errors in very high contention scenarios.

In the remainder of this Section we introduce a novel, hybrid approach that combines model-driven and exploration-based techniques in order to achieve the



(a) General architecture (b) Internals of the decision module

Fig. 5: Overview of the Controller for the DTM platform.

benefits of both approaches, namely high robustness and high speed of convergence to the optimal solution. Finally, we evaluate its effectiveness via a trace-driven simulation study.

4.1 Combining model-driven and exploration-based techniques

The key idea at the basis of the proposed solution consists in progressively enhancing the accuracy of an analytical-based performance model (such as TAS), by exploiting the feedback on the actual performance achieved by the platform when using alternative multiprogramming levels. In order to minimize exploration costs, the additional multiprogramming configurations explored by the self-tuning mechanism are obtained by fixing the current number of nodes in the platform (hence avoiding state transfers) and altering exclusively the number of active threads per node. Figure 5a illustrates the architecture of the system, which is composed of a set of DTM nodes and a controller process. The controller process is a logical component, which can be physically deployed either on one of the computing nodes or on a dedicated one. The controller is responsible for adjusting the scale of the DTM platform, i.e., the number of nodes and threads per node, which we note as $\overrightarrow{S}$. Its logic is implemented via a closed control loop that is analogous to the one show in Figure 2, with two noteworthy exceptions: (i) in this case the controller is fed with data from multiple nodes, which need to be aggregated and averaged by the collection module, and (ii) in addition to the current throughput of the DTM, noted T_{cur}, the controller gathers also a set of statistics characterizing the workload, referred to as $\overrightarrow{W}$, which are used as input parameters of its performance prediction methodology. The workload is characterized by means of the following set of key features (which represent also the main input parameters of the analytical performance model integrated in the decision module): duration, and relative frequency, of read-only and update transactions, abort rate, average number of write operations per transaction.

The key component of the controller is the decision module, whose internal structure is illustrated in Figure 5b. The decision module is composed of three main sub-components: the *Self Correcting Transactional Auto Scaler (SCTAS)*,

the *Exploration Manager* and the *Scale Optimizer*. Below, we describe these components in detail.

SCTAS. Like classic performance forecasting models, SCTAS allows predicting the throughput achievable by the DTM platform when faced with a workload $\overrightarrow{W}$ and configured to use a global multiprogramming level $\overrightarrow{S}$. To this end, we assume that SCTAS exposes the primitive `query`, which takes as input parameters $\overrightarrow{W}$ and $\overrightarrow{S}$, and returns the forecast throughput, denoted as T_{SCTAS}. Additionally, SCTAS allows incorporating feedback on the actual throughput (T_{cur}) achieved by the DTM platform in a given operational condition via the primitive `update`, which takes as input parameters $\overrightarrow{W}$, $\overrightarrow{S}$ and T_{cur}.

As shown in Figure 5b, SCTAS is actually composed of two main modules: the TAS model, and a, so called, Patcher. The role of the Patcher is to learn a *correcting function*, denoted as Φ and defined over $\overrightarrow{W} \times \overrightarrow{S}$, which, when "applied" to the output of TAS, denoted as T_{TAS}, allows minimizing its prediction error. In principle, several approaches may be used to derive the value of $\Phi(\overrightarrow{W}, \overrightarrow{S})$, given T_{cur} and the corresponding T_{TAS} for a given $(\overrightarrow{W}, \overrightarrow{S})$ pair. In this paper, however, we take a pragmatical approach and use a relatively simple solution which, as we will show in the following, was experimentally proved to work quite effectively. Namely, we define:

$$\Phi(\overrightarrow{W}, \overrightarrow{S}) = \frac{T_{TAS}(\overrightarrow{W}, \overrightarrow{S})}{T_{cur}(\overrightarrow{W}, \overrightarrow{S})} \tag{1}$$

In order to learn how the corrective factor Φ varies with $\overrightarrow{W}$ and $\overrightarrow{S}$, SCTAS uses a decision-tree based machine learning regressor, namely Cubist [28]. Analogously to classic decision tree based classifiers, such as C4.5 and ID3 [29], Cubist builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain. However, unlike C4.5 and ID3, which contain elements in a finite discrete domain (i.e., the predicted class) as leaves of the decision tree, Cubist places a multivariate linear model at each leaf.

As we will see shortly, the knowledge base of the machine learner embedded in the Patcher is progressively built by exploring alternative multiprogramming levels ($\overrightarrow{S}$) in presence of a workload $\overrightarrow{W}$. For each couple $(\overrightarrow{W}, \overrightarrow{S})$ corresponding to an explored scenario, the machine learner is fed with the triple $<\overrightarrow{S}, \overrightarrow{W}, \Phi(\overrightarrow{W}, \overrightarrow{S})>$. Based on its knowledge base, the machine learner builds a function Φ', which is used by the Patcher to estimate the corrective factor for a future (unexplored) configuration $(\overrightarrow{W'}, \overrightarrow{S'})$. The throughput forecast by SCTAS, T_{SCTAS}, can then be obtained (inverting Eq. 1) as:

$$T_{SCTAS}(\overrightarrow{W}, \overrightarrow{S}) = \Phi'(\overrightarrow{W}, \overrightarrow{S}) \cdot T_{TAS}(\overrightarrow{W}, \overrightarrow{S}) \tag{2}$$

Exploration Manager. This module is in charge of determining which configurations of the multiprogramming level should be tested in order to gather feedback on the accuracy of the SCTAS performance model, and, ultimately,

Algorithm 1: Pseudocode of the controller for the DTM platform.

```
seenWorkloads ← ∅
function Optimize()
    while TRUE do
        (W⃗, T_cur) ← Collect()
        if W⃗ ∉ seenWorkloads then
            CorrectViaLocalExploration(W⃗, S⃗, T_cur)
            seenWorkloads ← seenWorkloads ∪ W⃗
        S⃗* ← argmax_{s⃗} (SCTAS.Query(W⃗, s⃗))
        if S⃗* ≠ S⃗ then
            Deploy (S⃗*)

function CorrectViaLocalExploration(W⃗, S⃗, T_cur)
    numExploration ← 0
    while numExploration ≤ M do
        SCTAS.Update(W⃗, S⃗, T_cur)
        numExplorations ← numExplorations + 1
        if numExplorations ≥ μ ∧ SCTAS.GetCurErr() ≤ ε then
            break
        S⃗ ← ExporationManager.Explore(T_cur, S⃗)
        Deploy (S⃗)
        (W⃗, T_cur) ← Collect()
```

allow its correction. We abstract over the implementation of the exploration algorithm via the Explore () primitive that takes as input parameters the pair $(\overrightarrow{S},\overrightarrow{T})$. This abstraction allows to encapsulate arbitrary exploration logics, but, in this paper, we propose and evaluate a specific heuristic, described as follows.

At each invocation of the Explore () primitive, the heuristic alters the multiprogramming level (i.e., the number of active threads per node) leaving however unchanged the total number of nodes in the platform, in order to avoid triggering expensive state transfers. The heuristic operates in two phases. In the first phase, it executes according to a hill climbing technique analogous to the one described in the Sec. 3, and aims to identify the optimal multiprogramming level t^*, using the current number of nodes. In case the Explore () primitive is invoked after having identified such value, the heuristic enters a second phase during which it suggests to test configurations according to a *zig-zag* policy that explores (untested) values around t^*, by picking alternatively between values larger and smaller than t^* at increasing distance from t^*. The rationale underlying the design of this heuristic is to prioritize the exploration of configurations that maximize the throughput of the system, and, if necessary, to allow the testing of additional, suboptimal configurations which may be beneficial to the enhancement of the knowledge base of the Patcher module of SCTAS.

Scale Optimizer. This module is responsible for choosing the scale of the DTM, by exploiting in synergy the performance forecasting capabilities of SCTAS and the local exploration-based policy of the Exploration Manager. Its purpose is to drive SCTAS through its learning phase, by inducing local explorative steps,

aimed at assessing and improving SCTAS' accuracy. The pseudocode describing its logic is in reported Algorithm 1. The method OPTIMIZE() illustrates the interactions between the Scale Optimizer and the other modules composing the architecture of the Controller. It implements a simple control loop, which starts by gathering, via the COLLECT() method, information concerning the current throughput T_{cur} and workload $\overrightarrow{W}$. Once collected this information, the correction process of SCTAS is triggered by invoking the method CORRECTVIALOCALEXPLORATION(). This method, whose detailed description will be provided shortly, explores a number of alternative configurations of the multiprogramming level, in order to gather feedback from the DTM system and extending the knowledge base of the Patcher module of SCTAS. As this method terminates, SCTAS is queried to determine the optimal global multiprogramming level. Finally, if the optimal scale predicted by SCTAS differs from the current one, the DTM is accordingly reconfigured via the DEPLOY() primitive. The loop cycles back, in order to continue monitoring for possible changes of the workload. Note that the correction process for a workload $\overrightarrow{W}$ is triggered only if SCTAS has not been already corrected against $\overrightarrow{W}$, which ensures the system stability in presence of stable inputs (i.e., workloads).

Let us now analyze the logic of the CORRECTVIALOCALEXPLORATION() method, which consists of a loop that performs two main operations. First, SCTAS is provided with a feedback about the throughput of the application in the current configuration. Next, the Exploration Manager is queried to determine the multiprogramming level to be tested in the next iteration, and the DTM is accordingly reconfigured via the DEPLOY() primitive. The loop terminates when either one of the following two conditions is met: (i) a minimum number of explorations, μ, has already been performed, and the accuracy of the SCTAS predictions on the set of configurations tested so far has achieved a configurable threshold, denoted as ϵ in the pseudocode; (ii) a maximum number of explorations, M, has already been performed.

The latter condition ensures the eventual termination of the local exploration phase after a bounded number of attempts. The former allows to control the duration of the local exploration phase via two parameters: ϵ allows to bound the error of SCTAS on the configurations explored so far, for which it has therefore already collected measurements from the DTM platform; μ, on the other hand, allows to control directly the minimum duration of each local exploration phase, and, indirectly, to determine the amplitude of the knowledge phase of the Patcher module and its ultimate effectiveness.

4.2 Evaluation

In this section we assess the validity of the proposed hybrid approach. We compare it with a purely model-driven one and evaluate it in terms of final accuracy of the corrected model, capability of identifying the optimal scale for a DTM application and convergence speed towards it. To this end, we built a simulator which implements the logic of the Controller. Data consumed by the simulator

Model	Normalized Throughput wrt Max	#Local Expl	#Global Expl	Global Avg Rel Err
TAS	0.70	0	0	1.03
SCTAS($\mu = 3$)	1	9	1	0.7
SCTAS($\mu = 5$)	1	15	1	0.6
SCTAS($\mu = 7$)	1	28	2	0.05

Table 1: Comparison between TAS and SCTAS with different values of μ.

are relevant to real traces, collected deploying the TPC-C application over a DTM of different scales, varying the number of nodes from 2 to 10, and the number of threads per node from 1 to 12. The testbed is the same as the one described at the beginning of Section 4.

We ran four simulations. Each run starts with the application deployed over a DTM composed by two nodes and one thread per node, and simulates the elastic scaling policy described in Section 4.1 by feeding it with measurements gathered (off-line) from the DTM system, till a stable state is reached. In the first simulation, the controller's decision module only relies on TAS' performance forecasting model to determine the optimal scale for the DTM; in the others, it implements the DECISION() method of the pseudocode in Algorithm 1. For these three runs we fix $M = 10$ and $\epsilon = 10\%$ and vary only the value of μ, which, we recall, determines the minimum duration of SCTAS learning phase through local exploration.

The results in Table 1 clearly demonstrate the advantages (in terms of increased accuracy) of the proposed hybrid approach when compared a purely model-driven one. Using only TAS' performance forecasting model, the controller selects a scale for which the DTM delivers a throughput that is 30% lower than the maximum achievable. On the other hand, the SCTAS-based controller is able to converge to the optimal scale, regardless of the value of μ. The sensibility of the hybrid approach to this parameter is clear when analyzing the last columns in Table 1. Results demonstrate that the value of μ represents a trade-off between convergence speed towards the optimal solution and the accuracy of the SCTAS' performance forecasting model. In Figure 6 we show how the accuracy of SCTAS increases with the numbers of explored configurations, by plotting the average prediction error of SCTAS' model. The prediction error is computed over the set of all possible scales for the DTM (in the considered range), including the unexplored ones, for which the actual throughput value has been collected offline. The discontinuity points of the curves are in correspondence of the deployment of the DTM on a different number of nodes, which yields to a major update of the corrective function learnt by the Patcher component. The plot shows that the highest accuracy is reached for μ=7; for this configuration we also show, in Figure 7, the predictions produced by SCTAS after the controller has reached its final state (contrasting them with the ones produced by the pure model-driven approach of TAS). The plot demonstrates the self-correcting capabilities of SCTAS, which, when fed with a sufficient number of feedbacks concerning the prediction's errors of TAS, can significantly improve its accuracy both in explored and unexplored scenarios, lowering the average relative absolute error across all scenarios from 103% to 5%. These results highlight that the

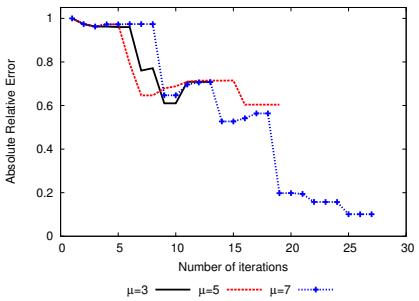


Fig. 6: Average of the relative absolute error across all scenarios

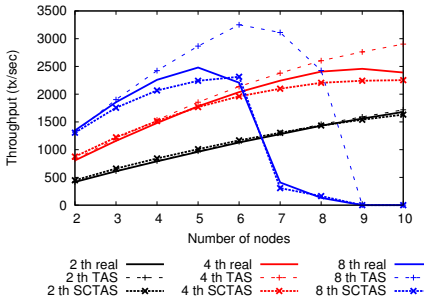


Fig. 7: Forecasts of SCTAS after a full optimization cycle ($\mu = 7$).

self-correcting capabilities of SCTAS can be beneficial not only to optimize the multiprogramming level of DTM applications, but also in applications (such as QoS/cost driven elastic scaling policies and what-if analysis of the scalability of DTM applications) requiring to speculate on the performance of the platform in different scale settings.

5 Conclusion

In this paper, we proposed and evaluated algorithms aimed to self-tune the multi-programming level in two radically different types of TM systems: a *shared memory STM* and *distributed STM*. We showed that for shared memory STM a simple exploration-based hill-climbing algorithm can be extremely effective, even when faced with challenging workloads. However, in the distributed TM case, pure exploration-based approaches are no longer a viable option, as testing configurations with a different number of nodes requires triggering costly state transfer phases. We tackled this problem by introducing a novel, hybrid approach that combines performance models and local exploration, in order to achieve the best of the two methodologies: quick convergence towards the global optimum and robustness to possible inaccuracies of the performance models.

References

1. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Elastic scaling for transactional memory: From centralized to distributed architectures. In: HotPar. (2012)
2. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. of ISCA. (1993)
3. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory. 2nd edition edn. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher (2010)
4. Reimer, N., Haenssgen, S., Tichy, W.F.: Dynamically adapting the degree of parallelism with reflexive programs. In: Proc. of IRREGULAR. (1996)

5. Heiss, H.U., Wagner, R.: Adaptive load control in transaction processing systems. In: Proc. of VLDB. (1991)
6. Schroeder, B., Harchol-Balter, M., Iyengar, A., Nahum, E., Wierman, A.: How to determine a good multi-programming level for external scheduling. In: Proc. of ICDE. (2006)
7. Abouzour, M., Salem, K., Bumbulis, P.: Automatic tuning of the multiprogramming level in Sybase SQL Anywhere. In: Proc. of ICDE Workshops. (2010)
8. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proc. of SPAA. (2008)
9. Mohammad, A., Mikel, L., Christos, K., Kim, J., Chris, K., Ian, W.: Robust adaptation to available parallelism in transactional memory applications. In: HIPEAC Journal. (2008)
10. Ghanbari, S., Soundararajan, G., Chen, J., Amza, C.: Adaptive learning of metric correlations for temperature-aware database provisioning. In: Proc. of ICAC. (2007)
11. Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: Proc. of ICAC. (2007)
12. di Sanzo, P., Ciciani, B., Quaglia, F., Romano, P.: A performance model of multiversion concurrency control. In: Proc. of MASCOTS. (2008)
13. Heindl, A., Pokam, G., Adl-Tabatabai, A.R.: An analytic model of optimistic software transactional memory. In: Proc. of ISPASS. (2009)
14. Dragojevic, A., Guerraoui, R.: Predicting the scalability of an stm: A pragmatic approach. (2010)
15. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: Machine learning-based self-adjusting concurrency in software transactional memory systems. In: Proc. of MASCOTS. (2012)
16. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proc. of IISWC. (2008)
17. J. Schenker: Optimistic Synchronization and the Natural Degree of Parallelism of Concurrent Applications — MSc Thesis (June 2012)
18. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2stm: Dependable distributed software transactional memory. In: Proc. of PRDC. (2009)
19. Narasimha Raghavan, R.V.: Balancing the communication load of state transfer in replicated systems. In: Proc. of SRDS. (2011)
20. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G.: Non-intrusive, parallel recovery of replicated data. In: Proc. of SRDS. (2002)
21. Elnikety, S., Dropsho, S., Cecchet, E., Zwaenepoel, W.: Predicting replicated database scalability from standalone database profiling. In: Proc. of EuroSys. (2009)
22. Didona, D., Romano, P., Peluso, S., Quaglia, F.: Transactional auto scaler: elastic scaling of in-memory transactional data grids. In: Proc. of ICAC. (2012)
23. Rahul Singh, Upendra Sharma, E.C.P.S.: Autonomic mix-aware provisioning for non-stationary data center workloads. In: Proc. of ICAC. (2010)
24. Francesco, M., Manik, S.: Infinispan Data Grid Platform. Packt Publishing (2012)
25. Red Hat/JBoss: JBoss Infinispan. http://www.jboss.org/infinispan (2011)
26. TPC Council: TPC-C Benchmark. http://www.tpc.org/tpcc (2011)
27. Yu, P.S., Dias, D.M., Lavenberg, S.S.: On the analytical modeling of database concurrency control. ACM Journal (1993)
28. Quinlan, J.R.: Rulequest Cubist. http://www.rulequest.com/cubist-info.html
29. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc. (1993)