Automatic Tuning of the Parallelism Degree in Hardware Transactional Memory*

Diego Rughetti[1], Paolo Romano[2], Francesco Quaglia[1], and Bruno Ciciani[1]

[1] Sapienza Universita' di Roma, Italy
[2] Instituto Superior Técnico, Universidade de Lisboa/INESC-ID, Portugal

Abstract. Transactional Memory (TM) is an emerging paradigm that promises to ease the development of parallel applications. Due to its inherently speculative nature, however, TM can suffer of performance degradations in presence of conflict intensive workloads.

A key technique to tackle this issue consists in dynamically regulating the number of concurrent threads, which allows for selecting the concurrency level that best fits the intrinsic parallelism of specific applications. In this area, several self-tuning approaches have been proposed for Software-based implementations of TM (STM). In this paper we investigate the effectiveness of these techniques when applied to Hardware TM (HTM), a theme that is particularly relevant and timely given the recent integration of hardware supports for TM in next generation of mainstream Intel processors. Our study, conducted on Intel's implementation of HTM, identifies several issues associated with the employment of techniques originally conceived for STM. Motivated by these findings, we propose an innovative machine learning based technique explicitly designed to take into account peculiarities of HTM systems, and demonstrate its advantages, in terms of higher accuracy and shorter learning times, using the STAMP benchmark suite.

1 Introduction

Transactional Memory (TM) [12,20] is an attractive programming paradigm for developing parallel/concurrent applications. By relying on the notion of atomic transaction, TM stands as a simper alternative to traditional lock-based synchronization. In more detail, with TM code blocks accessing shared-data can be marked as transactions. The complexity associated with enforcing coherency of concurrent data accesses is then delegated to the TM layer, rather than to any hand crafted synchronization scheme defined by the programmer. The maturing of the intense research that targeted TM over the last decade has recently led to the development of TM supports for the most popular open source compiler (GCC), and to the integration of hardware implementations of TM (HTM) in the last generations of processors produced by major vendors, such as Intel or IBM.

Even though TM shows a big potential for simplifying the software development process, another aspect that is central for the success of TM systems is the actual level

* This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013, by the COST Action IC1001 Euro-TM and by project GreenTM EXPL/EEI-ESS/0361/2013.

of performance they can deliver. In such a context, one core issue to cope with is related to maximize parallelism, while avoiding thrashing phenomena due to excessive data contention and high transaction abort rates.

For the case of Software-based implementations of TM (STM), several approaches have been proposed to cope with thrashing avoidance (see, e.g., [5,18,1,2,23]). One of the key techniques exploited in these approaches consists in (dynamically) regulating the actual level of concurrency, i.e. the number of concurrently active threads. All these approaches rely on performance models (either white-box, e.g. analytic [18,5], or black-box, e.g. machine-learning [1]), which are used to predict the expected performance, depending on the application's workload, while varying the number of threads.

On the other hand, we are not aware of any study in literature that investigates the issue of how to optimize the degree of parallelism in HTM systems. In this paper we aim to fill this gap, whose relevance is particularly strong given the recent integration of HTM in mainstream processors. We start by showing that the problem cannot be effectively addressed by reusing techniques originally conceived to operate in STM contexts, due to two key reasons:

1. Existing techniques targeting STM rely on models that do not consider transaction abort causes that are specific to HTM, and that are completely absent in STM systems. Particularly, in HTM a large number of transaction aborts is due to capacity constraints of processors' caches, as well as to a plethora of different micro-architectural reasons [13] (e.g. interrupts, faults or traps).
2. STM-oriented approaches are typically based on software instrumentation and run-time monitoring of specific parameters (whose values serve as input to instantiate performance models aimed to guide concurrency optimization). Monitoring these same parameters in the context of HTM is however unaffordable: existing HTM implementations do not externalize them, and monitoring them at the software level would induce overheads analogous to those of implementing an STM, defeating the whole purpose of HTM.

In the light of these considerations, this paper makes an additional contribution, by proposing a novel machine learning based technique to dynamically adapt the concurrency degree of HTM-based applications. The proposed self-tuning mechanism is explicitly designed to take into account the peculiarities of HTM systems, and avoids the issues that affect existing STM-oriented solutions. Via an extensive experimental evaluation based on the well known STAMP benchmark suite [14], and on a HTM-equipped Intel Haswell processor (8 virtual cores - 4 physical with hyper-threading), we show that the proposed approach achieves, on average, twice the accuracy of existing methods, while imposing negligible overheads and abating learning times dramatically.

The remainder of this paper is structured as follows. Section 2 discusses the state of the art on adaptive solutions for TM systems. In Section 3, we discuss issues associated with the employment, in the context of HTM, of solutions originally designed to self-tune the degree of parallelism in STM. Section 4 presents the proposed solution for optimizing the parallelism level in HTM applications. Section 5 presents the results of the experimental evaluation based on the STAMP benchmark. Finally, Section 6 concludes the paper.

2 Related Work

Several analytical models of STM [17,11] have been presented in literature. These models adopt a white-box approach to capture execution dynamics of STM and allow for predicting applications' performance in different configurations. Employing these techniques for the self-tuning of HTM systems is however infeasible for several of reasons. First, as they rely on white-box models tailored to STM, they fail to capture important peculiar aspects of HTM, in particular aborts induced by hardware-imposed restrictions. Further, these solutions typically require extensive instrumentation to gather a large set of parameters that serve as input for the white-box performance prediction model. Such instrumentation is not supported by existing HTM, and implementing it via software would induce unaffordable overheads, as we will also show in Section 3.

Other works have been based on black-box approaches, relying on various types of statistical/machine learning techniques to capture STM performance trends. These include techniques based on fitting to predetermined families of functions [8,16], or more generic regressors such as neural networks [15] and decision trees [5]. As we will discuss in more detail in the next section, employing these techniques in HTM systems would induce prohibitive instrumentation overheads. Also, being designed to operate in STM environments, the input parameters used by these models turn out to be inadequate to capture the proper dynamics of HTM.

Other black-box approaches adopt a model-free feedback-based method, implementing hill-climbing techniques that adapt the parallelism degree by reacting to variations of some key performance indicator, such as throughput [4] or abort rate [1]. Due to their model-free/exploratory nature, these approaches suffer of two main issues: slow convergence to the optimal solution [19], and risk of being trapped in local maxima.

Another related topic is transaction scheduling [23,9], in which the mapping of transactions to threads is dynamically adapted in order to minimize data contention. Such a technique has the effect of adapting the degree of parallelism, because rescheduled threads are removed from the execution for a while. Existing scheduling techniques employ different types of information, ranging from high level statistics on the abort ratio [23], to details on transaction's readset and writeset [9]. As already discussed, obtaining information on transactions' data access patterns is not feasible with existing HTM implementations.

Other related works, exploit machine learning to optimize orthogonal configuration parameters of STM, such as selecting the best performing conflict detection and management algorithm [22] or the most suitable mapping of threads to CPU-cores [3].

Our work is also related to recent research in the area of performance evaluation of HTM, both for Intel [7,10] and IBM implementations [21]. To the best of our knowledge, the only existing work in the area of self-tuning for HTM [6] copes with an issue orthogonal to the one tackled in this work, namely the tuning of the retry logic and fall-back path.

3 Concurrency Regulation Approaches: STM vs HTM

In this section we assess the effectiveness of existing approaches for self-tuning the degree of parallelism of STM, when employed in the context of HTM. We focus our study on model-based approaches that rely on machine learning [15,16]. This choice is motivated by the fact that, as discussed in Section 2, we are not aware of any analytical model capable of predicting the performance of HTM. Also, model-based approaches are known to achieve faster convergence than model-free ones [19], and avoid the issue of getting stuck in local maxima.

The performance models adopted in these approaches [15,16] aim at predicting the transaction wasted time (namely the CPU time spent in the execution of transaction instances that are eventually aborted) as a function of the number of concurrent threads. These models take as input a set of parameters, some of which are used to capture the data access pattern, and provide in output the expected wasted time. Specifically, these models can be seen as implementing the following function:

$$w_{time} = f(rs_{size}, ws_{size}, rw_{aff}, ww_{aff}, t_{time}, ntc_{time}, k) \tag{1}$$

where k denotes the number of concurrent threads supposed to run the application, w_{time} is the average transaction wasted time, rs_{size} (resp. ws_{size}) is the average read-set (resp. write-set) size of transactions, rw_{aff} – read-write affinity (resp. ww_{aff} – write-write affinity) is an index providing an estimation of the likelihood that an object read (resp. written) by a transaction is also written by another transaction, t_{time} is the average execution time of the committed transaction runs, and ntc_{time} is the average execution time of non-transactional code blocks. As for the latter parameter, it is typical for TM applications interleave, in the same thread, the execution of transactional and non-transactional code blocks. The non-transactional blocks are typically used for tasks such as the interaction with an external user/application and/or the acquisition of input parameters for the transaction to be run.

In the solutions in [15,16] the shape of the function f is determined either by fitting data in the training set using generic neural networks, or by using a specialized family of analytical functions (which is used to build sub-functions whose composition determines the actual shape of f). In both cases, the predicted value of the transaction wasted time is used to compute the value of the expression $k/(w_{time}+t_{time}+ntc_{time})$, which represents the system throughput, and so to predict the value of k that is expected to maximize the throughput.

As pointed out before, both approaches rely on the run-time monitoring of the input parameters of function f. This is requested both for the initial model instantiation phase, as well as for performance prediction and concurrency regulation (once the application is already deployed). Particularly, the run-time monitoring of rs_{size}, ws_{size}, rw_{aff} and ww_{aff} allows for determining whether workload shifts occur, which may require a change of the parallelism degree k in order to ensure optimal performance.

These approaches adopt a further refinement of the performance model, which takes into account the fact that, besides w_{time}, also t_{time} and ntc_{time} can actually vary significantly with k. This phenomenon is imputable to hardware level contention, such as cross-core cache contention at lower cache-levels in the multi-core architecture. Hence, the observed values of t_{time} and ntc_{time} cannot be immediately used as input to the function f when carrying out predictions with values of k different from the ones used when those values were observed. Rather, *correction functions* are used to predict the values of t_{time} and ntc_{time} in the target configuration of the parallelism level for which

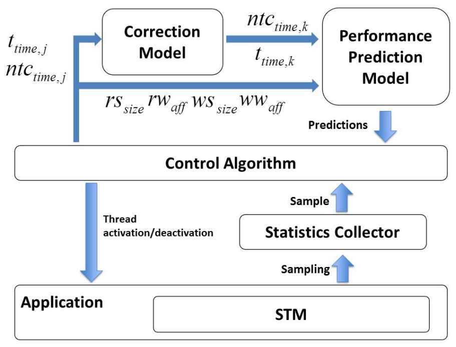


Fig. 1. STM-oriented concurrency regulation architecture

estimation is being carried out. These correction functions are typically much simpler than f (in fact, they are often linear), and have been shown to be identifiable in various ways, e.g., via a simple polynomial regression approach [16]. Overall, the final equation used for dynamically computing the best suited parallelism level, via maximization vs the value of k, is

$$\frac{k}{w_{time,k} + t_{time,k} + ntc_{time,k}} \tag{2}$$

where the subscript 'k' exactly expresses the above depicted parameter dependency (also involving t_{time} and ntc_{time}). The architecture that has been proposed for exploiting the above model in order to dynamically regulate concurrency in STM systems is schematized in Figure 1.

When porting the above approaches (that are naturally conceived for STM) on top of HTM-based systems, the following two issues arise:

1. *Monitoring overhead* - tracing the features to be used as input to the performance model in Eq. 1 would be too costly in HTM. Specifically, obtaining information on readset/writeset size would require instrumenting every single transactional operation, paying a cost analogous to the one paid when handling transactional accesses via software techniques (just like in STM). Also, the relative cost for computing parameters like rw_{aff} and ww_{aff} (which are based on the scalar product of relative read/write access rates to individual transactional objects kept by the TM) would dominate, when compared to the actual transaction processing time in HTM based systems. These overheads would hinder performance severely, especially when considering that the key advantage of HTM systems is to avoid any cost related to additional software instrumentation.
2. *Inadequacy of the input features* - as already mentioned, a key difference between STM and HTM is that, in the former, data conflicts are the unique source of transaction aborts. In fact, the input parameters for Eq. 1, used as the base performance model by the works in [15,16], are targeted to characterise the data access profile as the unique cause for transaction aborts, and do not capture the dynamics of aborts due to architectural constraints. As shown in Table 1, this kind of aborts actually represents the dominant source of aborts for all the STAMP benchmark applications.

Table 1. Abort reasons

Benchmark	conflict	capacity	other
vacation	1%	41%	58%
kmeans	0%	2%	98%
genome	1%	35%	64%
intruder	1%	40%	59%
labyrinth	0%	79%	21%
ssca2	0%	2%	98%
yada	34%	37%	29%

Table 2. Sampling overhead

Conc. level	kmeans	intruder	genome
1	2%	3%	3%
2	2%	4%	$3,5\%$
3	3%	$1,3\%$	$3,5\%$
4	2%	$1,8\%$	$1,3\%$
5	4%	$0,1\%$	$3,5\%$
6	$3,5\%$	$0,1\%$	3%
7	$1,6\%$	$0,1\%$	$3,5\%$
8	$4,5\%$	$4,5\%$	$1,7\%$

Hence, the need for devising models capable of explicitly capturing these phenomena, and to overcome the inadequacy of existing STM-oriented models.

These considerations led us to reconsider the set of input parameters to be used in the performance model, and to investigate on the ability of the following variant of the model to capture the dynamics proper of HTM:

$$w_{time} = f(t_{time}, ntc_{time}, abort_{conflict}, abort_{capacity}, abort_{other}, k) \tag{3}$$

where t_{time} and ntc_{time} have the already explained meaning, whereas the explanation of the other parameters is the following: $abort_{conflict}$ is the abort rate due to data-conflict, $abort_{capacity}$ is the abort rate due to overflows of cache capacity, and $abort_{other}$ is the abort rate due to other architectural reasons.

We evaluated this approach considering an instantiation of Eq. 3 based on neural networks, and two alternative instantiations of the correction function for t_{time} and ntc_{time}, one using linear regression and the other using again neural networks (NN). We refer to the whole approach as 2-layered, due to the presence of the correction function. Table 3 shows the discrepancy in the throughput (compared to the optimal throughput, statically determined by exploring all the concurrency levels between 1 and 8 for all the different phases of each benchmark run) which is achieved by regulating concurrency via the reliance on the model in Eq. 3. Instead, in Table 2 we report the run-time monitoring overhead for sampling the input parameters of the performance model as the number of thread varies (again between 1 and 8). We can see that the sampling overhead is very limited, confirming the adequacy of our choice in relation to the input features for the performance model in Eq. 3, from the perspective of efficiency. Concerning effectiveness while regulating concurrency, which is a reflection of the performance model accuracy, the results are less exciting, with errors (expressed in terms of throughput penalty with respect to the optimal achievable throughput) of up to 18% for the approach using linear regression, and 15% for the one using neural networks.

The key reason for this is that, contrary to the base performance model developed for STM (expressed by Eq. 1), in the proposed model for HTM in Eq. 3, all the input parameters may exhibit a dependency on the level of parallelism. So specific correction functions should be used for each of them (which might exhibit non-linear shape), increasing significantly the complexity of the approach, and ultimately degrading its accuracy. In order to back this claim, in the third column of Table 3 we provide data related to the performance that could be reached by the 2-layered approach if a set of

Table 3. Throughput penalty with the 2-layered approach

Benchmark	2-layered-linear	2-layered-NN	2-layered-optimal
intruder	8%	$6,3\%$	$3,2\%$
genome	10%	$4,4\%$	$2,7\%$
kmeans	18%	15%	$5,6\%$
vacation	18%	14%	$3,4\%$
ssca2	$0,80\%$	$0,74\%$	$0,55\%$
yada	0%	0%	0%
labyrinth	10%	9%	$3,2\%$

optimal correction functions for input parameters were available. As we can see comparing the third column with the first two, the performance delivered by the 2-layered approach strictly depends on the accuracy of the correction functions.

4 A Classification Based Approach

In order to cope with the issues pointed out in the previous section, we worked on an alternative way of approaching the problem of instantiating the performance model used to guide the adaptation of the concurrency level. To this end, we cast the performance prediction problem as a classification, and not a regression, problem. Specifically, given a workload profile, instead of predicting the system performance for every possible concurrency level (and then picking the optimal one), we aim to determine directly the optimal parallelism level, among the (finite set of) possible ones.

In this way we operate according to a "1-step" approach that does not require the use of correction functions, which were shown to be the Achilles' heel of existing approaches in Section 3. We decided to use and compare two different machine learning approaches to cope with this classification problem: Decision Trees and Neural Networks. However, as we will see in Section 5, both the algorithms provide very similar accuracy levels.

The fulcrum of the new approach is the construction of the training set for the classification algorithms. Particularly, each sample we relied on is a couple $< \mathbf{i}, \mathbf{o} >$ where $\mathbf{i} = [t_{time}, ntc_{time}, abort_{conflict}, abort_{capacity}, abort_{other}]$ and $\mathbf{o} = [k_{opt}]$, with k_{opt} representing the optimal level of parallelism, namely the concurrency level that ensures the best throughput given the workload profile expressed by $\mathbf{i}$.

The training set can be populated by executing a few runs of the application with different inputs and configuration parameters. For each input, the application is executed for any level of parallelism, namely varying the number of threads from 1 to the maximum number of hardware-threads supported by the target system. This way, for each workload/configuration tested during the training phase, it is always possible to determine the best performing concurrency level.

As we will show in Section 5, the new approach achieves consistently better accuracy than the 2-layered approach based on the performance model expressed by Eq. 3, namely the variation of the STM performance model originally exploited in [15,16]. Further, a relevant advantage of the new approach, beyond its higher accuracy, consists of its simplicity. On the other hand, a drawback with respect to the 2-layered approach, is that it does not allow to estimate the absolute performance achievable when using a degree of parallelism not considered in the training phase, which could be instead useful, for instance, to support what-if analysis. This aspect is inter-twinned with, e.g., provisioning processes in the Cloud, since what-if analysis with non-observed parallelism levels may lead to planning for provisioning adequately powerful multi-core machines (or scaling up/down already in use resources) in order to meet specific performance levels (while optimizing the costs). On the other hand, the new 1-step approach based on classification is targeted at optimizing the application run-time in scenarios where the available resources (and hence the set of possible parallelism levels for the hosted application) are known and could be tested during the training phase used to instantiate the performance model. Note that this is a means for optimizing already done investments.

5 Experimental Results

In this section we report experimental data for a comparison between the 2-layered approach derived by adapting the proposals in [16], [15] and the new classification based approach. We executed our tests on top of system equipped with an Intel Haswell Xeon E3-1275 3,5 GHz processor (8 virtual cores: 4 physical with hyper-treading[1]) with 32 GB RAM. Intel TSX extension (i.e., Intel's implementation of HTM) requires that a software-based fall-back method is specified, in case a transaction cannot be executed in hardware. In the evaluation we consider a fall-back path based on a single global lock. We keep on relying on the STAMP benchmark suite [14] also in this comparative study.

Let us start by analyzing the results considering the usage of a global lock on the fall-back path. Table 4 shows the mean penalty, with respect to the optimal throughput, due to wrong concurrency level choices. The first and the second columns report results for the classification approach implemented resp. with decision trees and neural networks. The third and fourth columns show results for the 2-layered approach using neural networks for the performance prediction model, and linear regression (column 3) or neural networks (column 4) for the correction function. Note that for all the considered approaches we are here considering the set of features specified by Eq. 3.

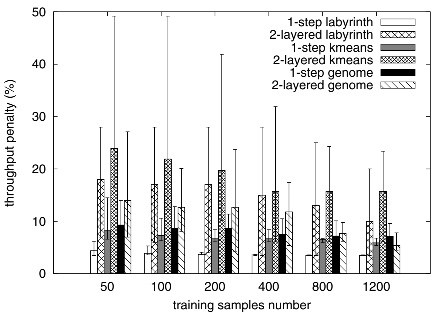
As we can see by comparing the first two columns, excluding the row related to the Intruder benchmark, using neural network or decision tree to implement classification approaches yields approximately the same performance. Looking at the third and fourth column, it emerges clearly that the proposed classification approach can provide significant benefits in terms of accuracy: the average throughput penalty (across all benchmarks) is in fact equal to $3,71\%$ and $3,39\%$, for the classification-based approach using, respectively, decision tree (DT) and neural network (NN), whereas the average throughput penalty for the 2-layered approach is of about $9,33\%$ when using a linear correction function and of approximately $7,06\%$ when using neural networks. This means, on average, a relative increase of accuracy by a factor 2.

[1] At the time of writing, this is the largest degree of parallelism achievable using HTM-equipped Intel processors.

Table 4. Throughput penalty comparison

Benchmark	classification-DT	classification-NN	2-layered-linear	2-layered-NN
intruder	$7,8\%$	$2,7\%$	8%	$6,3\%$
genome	$5,2\%$	$7,1\%$	10%	$4,4\%$
kmeans	$5,4\%$	$5,9\%$	18%	15%
vacation	$3,1\%$	$3,8\%$	18%	14%
ssca2	$0,70\%$	$0,72\%$	$0,80\%$	$0,74\%$
yada	0%	0%	0%	0%
labyrinth	$3,8\%$	$3,5\%$	10%	9%
average	$3,71\%$	$3,39\%$	$9,33\%$	$7,06\%$

The graphs in Figure 2 show how the performance penalty due to wrong prediction varies with respect to the number of samples used to train two different performance predictors, the one based on the proposed classification approach and the one based on the 2-layered approach. Each point is the mean value of the results of experiments executed with a fixed number of predictors that have been trained varying the composition of the training set and the configuration of the predictors (e.g. the number of hidden nodes in the neural networks). If we look at the left graph, which shows the results for the labyrinth, genome and kmeans benchmarks, we can see that the classification approach consistently outperforms the 2-layered one. Moreover the proposed approach requires less samples to ensure optimal performance and presents less variation in the results as shown by the bars on top of the histograms. These trends are confirmed by the right graph, which shows the performance penalty for other three benchmarks, namely intruder, vacation and ssca2. We avoid to present results related to the yada benchmark because, as shown in Table 4, for this benchmark all the approaches always ensure optimal performance (this is due to the fact that, at any point in time, the optimal configuration for yada never varies).



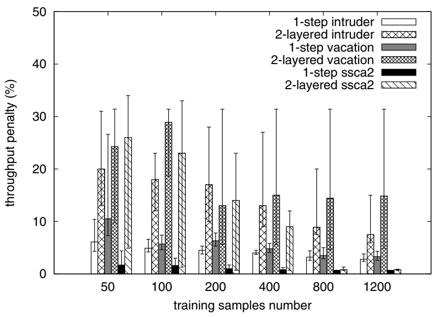


Fig. 2. Performance penalty varying predictor's training set size

The graphs in Figure 3 show the application speedup with respect to a non-instrumented sequential version, while varying the degree of parallelism, for two benchmarks of the STAMP suite, respectively Intruder and Genome. When running with no

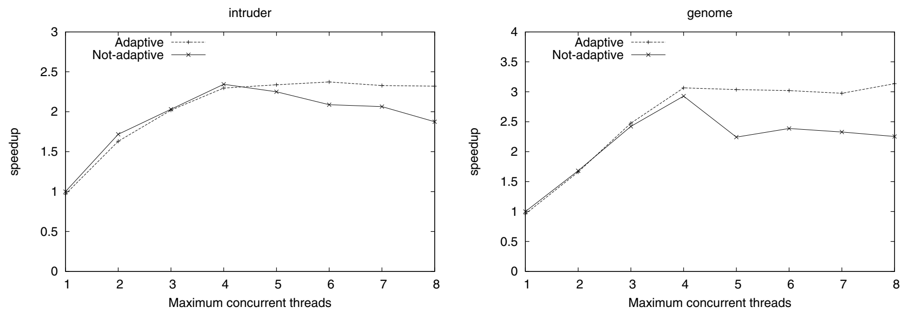


Fig. 3. Speedup

adaptive regulation of concurrency, we fix the degree of parallelism statically at start-up. On the other hand, when considering the adaptive version, we set the initial *and maximum* parallelism level according to the value reported on the x-axis of the figure, but then let the concurrency regulation mechanism adjust the parallelism level according to the indications of the performance model.

For the Intruder benchmark, when increasing the level of parallelism, the performance of the non-adaptive version of the application increases until it reaches a concurrency level equal to 4. Beyond this optimal level of parallelism, the performance decreases due to an excessive number of transaction aborts. The adaptive version of the application, instead, is able to determine at runtime which is the optimal concurrency level. As the dotted line in the graph shows, if we execute the application with a number of maximum available threads larger than 4, the adaptive version ensures the same speed-up that the application can reach when it is executed with the optimal concurrency level. Similar results can be obtained with the Genome benchmark as shown by the right plot.

Finally, in Figure 4 we report data showing the relative performance improvements achievable by approaches that dynamically regulate concurrency vs the static case where all the 8 available virtual cores are always employed for running the application. In this study we considered both our 1-step proposal, based on machine learning,

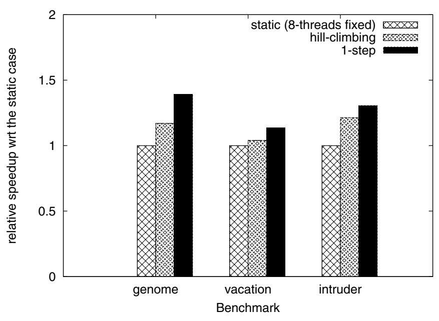


Fig. 4. Speedup (vs the static configuration employing 8 threads) of hill-climbing [4] and 1-step

and the hill climbing based technique investigated in [4]. The data refer to three different benchmark applications from STAMP, namely genome vacation and intruder. The plots highlight that the 1-step approach constantly outperforms the hill-climbing technique. This is as a result of the ability of the proposed approach to avoid sub-optimal exploration phases (unlike hill-climbing) and of identifying the optimal configuration in a prompt and accurate way.

6 Conclusions

In this paper we presented the results of a study aimed at evaluating the feasibility of re-using concurrency regulation techniques originally conceived for STM systems, or adaptations of them, in the context of HTM systems.

We have shown, also via experimentation, that these techniques do not fully fit HTM scenarios for two main reasons. On the one hand, the inadequacy of the parameters selected as input to the performance models used to drive the concurrency regulation process. On the other hand, the overhead for the monitoring of the model's input parameters, which becomes unaffordable in HTM.

We then devised and investigated a machine learning approach, based on classification and specifically tailored for HTM, which we have shown to yield higher accuracy, reduced overhead and shorter learning time. The assessment of this approach has been based on experimental results achieved by running the STAMP benchmark suite on top of a machine equipped with and Intel 8 virtual cores CPU (4 physical plus hyperthreading) with HTM support.

As future work along the concurrency regulation path we plan to investigate how to combine performance prediction models, and how to devise innovative models, for contexts where STM and HTM co-exist as an hybrid support for shared-data management in parallel/concurrent applications.

References

1. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 719–728. Springer, Heidelberg (2008)
2. Blake, G., Dreslinski, R.G., Mudge, T.: Proactive transaction scheduling for contention management. In: Proc. of MICRO, pp. 156–167. ACM (2009)
3. Castro, M., Goes, L.F.W., Ribeiro, C.P., Cole, M., Cintra, M., Mehaut, J.F.: A machine learning-based approach for thread mapping on transactional memory applications. In: Proc. of HiPC, pp. 1–10. IEEE Computer Society (2011)
4. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Identifying the optimal level of parallelism in transactional memory applications. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 233–247. Springer, Heidelberg (2013)
5. Didona, D., Romano, P., Peluso, S., Quaglia, F.: Transactional auto scaler: elastic scaling of in-memory transactional data grids. In: Proc. of ICAC, pp. 125–134. ACM (2012)
6. Diegues, N., Romano, P.: Self-tuning intel transactional synchronization extensions. In: Proc. of ICAC (2014)

7. Diegues, N., Romano, P., Rodrigues, L.: Virtues and limitations of commodity hardware transactional memory. In: Proc. of PACT (2014)
8. Dragojević, A., Guerraoui, R.: Predicting the scalability of an STM: A pragmatic approach. In: TRANSACT (2010)
9. Dragojević, A., Guerraoui, R., Singh, A.V., Singh, V.: Preventing versus curing: Avoiding conflicts in transactional memories. In: Proc. of PODC, pp. 7–16. ACM (2009)
10. Goel, B., Titos, R., Negi, A., McKee, S.A., Stenstrom, P.: Performance and energy analysis of the restricted transactional memory implementation on haswell. In: Proc. of IPDPS. IEEE Computer Society (2014)
11. He, Z., Hong, B.: Modeling the run-time behavior of transactional memory. In: Proc. of MASCOTS, pp. 307–315 (2010)
12. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
13. Intel Corporation: Intel Transactional Synchronization Extensions (Intel TSX) - Programming Considerations
14. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Proc. of IISWC, Seattle, WA, USA, pp. 35–46 (2008)
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, pp. 278–285. IEEE Computer Society (2012)
16. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: Regulating concurrency in software transactional memory: An effective model-based approach. In: Proc.of SASO. IEEE Computer Society (2013)
17. di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., Romano, P.: On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. Performance Evaluation 69(5), 187–205 (2012)
18. di Sanzo, P., Palmieri, R., Ciciani, B., Quaglia, F., Romano, P.: Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns. In: Proc. of WOSP/SIPEW (2010)
19. 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)
20. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. of PODC, pp. 204–213. ACM (1995)
21. Wang, A., Gaudet, M., Wu, P., Ohmacht, M., Amaral, J.N., Barton, C., Silvera, R., Michael, M.M.: Software support and evaluation of hardware transaction memory on blue gene/q. IEEE Transactions on Computers 99 (2013)
22. Wang, Q., Kulkarni, S., Cavazos, J.V., Spear, M.: Towards applying machine learning to adaptive transactional memory. In: Proc. of TRANSACT (2011)
23. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proc. of SPAA, pp. 169–178. ACM (2008)