Autonomic Concurrency Regulation in Transactional Memory

Short Term Scientific Mission Relation

Applicant: Diego Rughetti

January 15, 2014

Abstract

Transactional Memory (TM) has emerged as a powerful programming paradigm for concurrent applications. TM allows encapsulating the access to data shared across concurrent threads within transactions, thus avoiding the need for synchronization mechanisms to be explicitly coded by the programmer. On the other hand, synchronization transparency must not come at the expense of performance. Hence, TM-based systems must be enriched with mechanisms providing optimized run-time efficiency. Among the issues to be tackled, a crucial one is related to determining the optimal level of concurrency (number of threads) to be employed for running the application. When using too low levels of concurrency, the intrinsic parallelism of applications may not be fully untapped. On the other hand, an excessively high concurrency level may lead to thrashing phenomena caused by excessive data contention and consequent transaction aborts.

In this relation we present the results of a study aimed to evaluate to what extent existing techniques for self-tuning the concurrency level of Software Transactional Memory can be effectively used in the context of Hardware Transactional Memory (Intel TSX). Our study highlights that the techniques developed for STM can exhibit significant drawbacks when employed in a HTM-based system. This finding has motivated the investigation of two novel approaches for optimal concurrency level prediction explicitly tailored for Hardware Transaction Memory. By means of an experimental study based on the STAMP benchmark suite, we show that the proposed techniques are not only lightweight, but also that they can achieve a significantly higher accuracy than pre-existing optimization techniques proposed for STM.

1 The concurrency problem in STM and HTM

One of the main challenges that a programmer has to face during the design of parallel and distributed applications is how to ensure its scalability, that is the capacity of an application to increase proportionally its performance increasing the amount of available computing resources. Focusing our attention on centralized multi-core systems, an ideal parallel application should scale linearly with the number of available cores. Usually this level of scalability is not reachable due to two main factors:

- contention on physical resource: this issue is experienced by processes/threads that compose the parallel application when they try to access shared hardware resource (e.g. memory buses). It is strictly related to the specific hardware used to run the application and keeping under control its impact on performance requires a detailed hardware knowledge;
- contention on logical resource: it is experienced by the processes/threads that compose the parallel application when they try to concurrently access shared logical resources, like data in shared memory. It is strictly related to the application logic and its impact on application performance can often be higher than that caused by the contention on physical resource.

To limit the impact of logical contention on application scalability and performance the programmer must have a detailed knowledge of the application logic and of the application data access pattern. In this way the programmer can divide properly the work between threads minimizing the need for synchronization, and use synchronization mechanisms that minimize wasted time (when synchronization is strictly necessary). But these tasks are not trivial and, in the context of transactional applications, TM helps to simplify developer's work by easing the task of synchronizing access to shared data. This implies that the logical contention experimented by transactional applications is closely related to the concurrency control mechanism implemented by the TM.

As we will show in section 3, unlike STM, in many HTM-based applications the main transaction abort reason is not the logical contention. Performance and scalability still depend from the application transactional profile but the transactions

abort reasons change. In HTM a large number of transaction aborts is due, together with shared data conflicts, to the limited cache size or to a plethora of different micro-architectural reasons.

Despite of these relevant differences, HTM-based applications are also likely to exhibit suboptimal peformance when deployed using a wrong level of parallelism. Like in STM systems, HTM-based applications are prone to thrashing phenomena (caused by excessive transaction rollbacks) in case the degree of parallelism in the execution is excessively high. On the other hand, for too low parallelism levels, the achievable speedup may still be suboptimal. So the dynamic control of the concurrency level is still one of the main technique that can be used to pursue optimal efficiency in HTM-based applications.

The remainder of this report is structured as follow. Section 2 is devoted to providing an overview of the state of the art on adaptive solutions for TM systems. In Section 3, we discuss the issues and limitations 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 novel adaptive solutions that were specifically designed, during this STSM, to optimize the parallelism level of HTM-based applications. Section 5 presents the results of an experimental evaluation based on the STAMP benchmark. Finally, Section 6 concludes the report and presents future work.

2 Adaptivity in Transactional Memories

An effective way to face performance tuning and dynamic resource allocation problems is to develop some kind of model that allows to predict the system performance given a specific input and a specific system configuration, so that the model can be used to choose the better configuration given the current system's input. The methodologies for developing such a model can be coarsely classified in three main classes:

- **White box approach**: this class includes techniques that require the explicit modelling of the internal system dynamics.
- **Black box approach**: this approach is the exact opposite of the previous one. Black box techniques observe only inputs, context and outputs of the system and use statistical methods (like machine learning) to identify internal system's patterns and rules.
- **Gray box approches**: in case black and white box approaches are combined, the resulting, hybrid solutions are often referred to as (**Grey box approaches**).

In [1] an analytical modeling approach that capture dynamics related to the execution of both transactional read/write memory access and non-transactional operation has been proposed. The model is used to evaluate the performance of STM applications as a function of the number of concurrent threads and other workload configuration parameters (E.g. execution cost of transactional and not-transactional operations, cost of begin, commit and abort operations). This kind of approach is targeted at building mathematical tools allowing the analysis of the effects of the contention management scheme on performance. To develop that models a detailed knowledge of the specific conflict detection and management scheme used by the target STM is required.

The work in [2] presents an analytical model taking as input a workload characterization of the application expressed in terms of transaction profiles (length of transactions, transactions arrival frequency, number of checkpoints and computing cost of transactions), contention probability and hardware resources consumption. The model predicts the application execution time (estimating the wasted time due to conflicts) as function of the number of concurrent threads sustaining the application, however the prediction is a representation of the average system behaviour over the whole lifetime of the application. In this approach the input parameters need to be calculated by running the application and profiling the workload and by inspecting the application code. Predictions are related to the execution scenario of the profiled application as determined by the workload configuration used to run the application. Hence, changing the workload configuration of the application, a new profiling may be required. In addition, predictions are related to the entire execution of the application. Because during the lifetime of an application some features, as the workload profile and the transaction conflict probability, can change, then it is not possible to perform dynamic predictions on basis of the current workload of the application.

The proposal in [3] is targeted at evaluating scalability aspects of STM systems. It relies on the usage of different types of functions (such as polynomial, rational and logarithmic functions) to approximate the performance of the application when considering different amounts of concurrent threads. The approximation process is based on measuring the speed-up of the application over a set of runs, each one executed with a different number of concurrent threads, and then on calculating the proper function parameters by interpolating the measurements, so as to generate the final function used to predict the speed-up of the application vs the number of threads. This approach doesn't require any knowledge of the system and of the workload but it has a limitation due to the fact that the workload profile of the application is not taken into account. Hence the prediction may prove unreliable when the profile gets changed wrt the one used during measurement and interpolation phases. If it changes, e.g. in terms of transaction profiles, over the lifetime of the application, the performance achieved with a given

number of concurrent threads can changes. As a consequence, the calculated performance function may become unreliable, unless calculating it again by taking new measurements.

In [4] a control algorithm dynamically changes the number of threads which can concurrently execute transactions on basis of the observed transaction conflict rate. It is decreased when rate exceeds an threshold while it is incremented when the rate is lower than another threshold.

In the approach proposed in [5], incoming transactions are enqueued and sequentialized when an indicator, referred to as *contention intensity* CI, exceeds a pre-established threshold. The contention intensity is calculated, by each concurrent thread, as a dynamic average depending on the number of aborted vs. committed transactions. In this approach the scheduler doesn't take scheduling decision for all the executed transactions but only for those that start under high contention. This infrequent access allows the scheduler to be implemented as a centralized module, thereby enabling an advanced and coherent system-wide scheduling scheme. Such approach doesn't affect negatively the performance when the contention is low and it limits the performance degradation when the contention grows.

In the proposal presented in [6], a transaction is sequentialized when a potential conflict with other running transactions is predicted. The prediction leverages on the estimation of the expected transaction read-set and write-set (on the basis of the past behaviour of other or the same transaction). Actually, the sequentializing mechanism is activated only when the amount of aborted vs. committed transactions exceeds a given threshold. The authors state that the scheduler can be integrated with any STM that uses visible writes (e.g. [7, 8, 9]).

All the above proposals do not directly estimate the wasted time due to aborted transactions (vs the level of concurrency). They only indirectly attempt to control the wasted time according to heuristics schemes.

As for machine learning, to the best of our knowledge, it has been used in the context of transactional memories by three works. In [10] machine learning techniques are used to select the best performing conflict detection and management algorithm. Conversely, in [11], machine learning is used to select the most suitable thread mapping, i.e., the placement of application threads on different CPU-cores in order to get the best performance. The goals of both these works are different and orthogonal with respect to our one, which focus on the regulation of the overall concurrency level in the system. In [12], a pure Neural Network based approach is used to regulate the level of parallelism STM based applications with the aim of optimize performance, but a weak point of this approach is that to obtain good performance prediction it is necessary to collect a consistent number of samples, the most possible distributed in the input parameter space. This is due to the low extrapolation/generalization capacity of Neural Network approach when a very limited view of the system behaviour is available (a very limited training set).

In [13], an analytical modelling approach is used to regulate the level of parallelism of STM based applications with the aim of optimize performance. In this approach a model is developed through the interpolation of real performance samples using some predefined type of mathematical functions.

In [14] an exploration-based approach that periodically performs on-line monitoring of the number of transaction commits and aborts and then decides to increase or decrease the level of parallelism has been developed (hill climbing technique maximizing transaction commit rate). For Distributed Transactional memories (STM) two approaches have been developed: transactional auto scaler (TAS) and self correcting transactional auto scaler (SCTAS). TAS[14] relies on a mixed AM/ML approach in which the AM is used to capture the data contention dynamics and the ML is used to predict the inter-node communication latencies in a DTM platform. The advantage of using ML lies in its black-box nature, which makes it a very well-fitting choice for coping with performance forecasting of components in cloud infrastructures, where typically there is little knowledge of the hardware system architecture, particularly as concerns the network. SC-TAS[14] extends TAS exploiting the idea of learning, by means of on-line ML techniques, a correction function to the output of TAS, hence allowing to minimize the prediction errors of TAS' AM-based forecaster.

3 Using already developed approaches with HTM based applications

In this section I will present in more detail two solutions [12, 13] for tuning the parallelism level of STM-based applications, which I have co-authored during the course of my PhD programme, and prior to performing this STSM. Next, I will discuss how these approaches should be adapted to be applied to the case of HTM, and what effectiveness is expectable that such mechanisms achieve, when adapted to operate with HTM-based applications.

Until now we have developed two approaches: one based on neural network [15] and one based on a family of analytical functions (whose parameters are learnt on-line using regression) to build forecasting models that allow to predict the application performance while varying the concurrency level. These predictions are then used runtime to choose the better concurrency level for application execution.

More in detail these performance prediction techniques rely on run-time monitoring of parameters characterizing the execution of the application, which are fed as input parameters of black-box performance models that allow to predict the

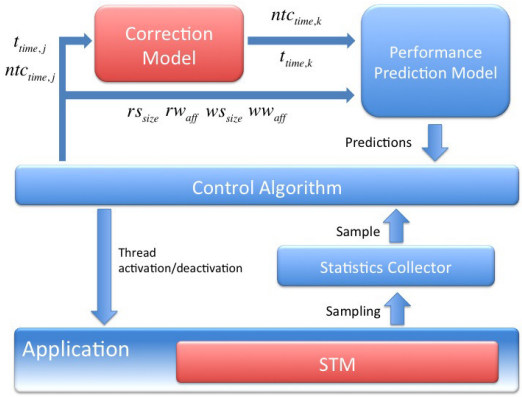

Figure 1: Adaptive STM architectures

average wasted time of transactions, i.e. the average execution time spent by all the aborted runs of a transaction. More formally, the performance model can be seen as the following function:

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

where w_{time} is the average wasted time of transactions, rs_{size} is the average read-set size of transactions, ws_{size} is the average write-set size of transactions, rw_{aff} is an index providing an estimation that an object read by a transaction could also be written by another concurrent transaction, ww_{aff} is an index providing an estimation that an object written by a transaction could also be written by another concurrent transaction, t_{time} is the average execution time of the committed transaction runs (i.e. the average execution time of the transaction runs that do not get aborted), ntc_{time} is the average execution time of non-transactional code blocks and k is the number of concurrently running threads.

A key issue with this approach is that the values of the parameters t_{time} and ntc_{time} can change significantly as the concurrency level changes. Hence, if one were to measure the values of these parameters when running an application at a concurrency level k, and used these values as input of the function f to predict w_{time} at a different concurrency level k', the prediction's accuracy would be most likely significantly degraded.

To cope with this issue both solutions use a, so called, *correction function* to predict the values of t_{time} and ntc_{time} in the target configuration of the parallelism level k', given the currently measured values of the set of input parameters. This correction function is tipically simpler (often linear) than f, and can be implemented in various ways, ranging from a simple polinomial regression approach [13] to neural networks [12]. In this way, as depicted in figure 1, we obtain 2-steps approaches to predict application performance: in the first step the values of $t_{time,k}$ and $ntc_{time,k}$ at different level of parallelism k are evaluated (via the correction function) starting from the $t_{time,j}$ and $ntc_{time,j}$ sampled at the currently used level of parallelism j, in the second step we evaluate

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

and than we use $w_{time,k}$ to predict, via the performance model, the application throughput for each level of parallelism K:

$$\frac{k}{w_{time,k} + t_{time,k} + ntc_{time,k}} \tag{1}$$

In the first phase of the STSM I have been working on adapting/extending this approach (originally conceinved for STM) to HTM-based systems. The following considerations apply:
1. *monitoring overhead*: tracing the input features used in these approaches would be too costly in HTM. In order to obtain informations like readset/writeset size would require instrumenting every single transactional operation, paying a cost analogous to STM instrumentation. Also, some of these features are quite onerous to compute (e.g., rw_{aff}) in STM environments, and these cost would result relatively amplified in a HTM system (which incurs no instrumentation costs).
2. *inadequacy of the input features*: As already mentioned, a crucial difference between HTM and STM is that, in the former, data conflicts are only a possible source of transaction aborts. The input parameters for the performance models used in [12, 13] are focused in logical contention, and do not to 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 in all the STAMP benchmarks, which highlights the relevance of capturing these phenomena.

These considerations led us, first of all, to reconsider the set of input features to be used in the performance model, which was redefined as follows:

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

where t_{time} and ntc_{time} are the same of STM, $abort_{conflict}$ is the abort rate due to conflict, $abort_{capacity}$ is the abort rate due to L1 cache size and $abort_{other}$ is the abort rate due to reason different by the previous two.

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 1: Abort reasons

Concurrency 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%

Table 2: Sampling overhead varying concurrency level

We evaluated this approach considering an implementation of the performance model using neural networks, and two alternative implementations of the correction function, i.e. using linear regression and, again, neural networks. Table 3 shows the accuracy obtained for all the benchmarks of the STAMP suite, and Table 2 the tracing overhead as the number of thread varies. We can see that instrumentation overhead is very limited, confirming the adequacy of our choice of input features for the model, from the perspective of efficiency. Concerning accuracy, the results are less exciting, with errors (expressed in terms of throughput penalty respect to the optimal obtainable throughput) of up to 18% for the approach using linear regression, and 15% for the one using neural networks.

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%

Table 3: Throughput penalty for already developed approaches

The key reason for this is that, contrary to the model developed for STM, in the proposed model for HTM all the input features for the performance model depend on the level of parallelism. So specific correction functions must be used for each parameter, increasing significantly the complexity of this approach, and ultimately degrading its accuracy. As a proof, in the third column of table 3 we provide data about the performance that can be reached if a set of optimal correction functions for input parameters were available. As we can see comparing the third column with the first two, the performance strictly depends on the accuracy of the correction functions.

4 A classification based approach

In order to cope with the issues identified above, we worked on an alternative way of approaching the problem of learning the performance model used to guide adaptation. To this end, we cast the performance prediction problem as a classification problem, instead of a regression problem. Namely, given an application workload characterization, instead of predicting the system performance for every possible concurrency level (and then pick the optimal one), we now try to determine exclusively which is the optimal parallelism level, among the (finite set of) possible ones.

In this way we obtain a "1-step" approach that does not require the use of correction functions, allowing to remove the uncertainty introduced by them. We decided to use and compare two different algorithms to solve our classification problem: Decision Trees and Neural Networks but, as we will see in section 5, both algorithms allow to obtain very similar accuracy.

The fulcrum of the new approach is the construction of the training set for the algorithms. Each sample 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} that represent the optimal level of parallelism, that is the concurrency level that ensure the better throughput given the workload profile represented by $\mathbf{i}$.

The training set can be populated executing a few runs of the application with different inputs/configuration parameters. For each input the application is executed with every available level of parallelism, i.e. from 1 to the maximum number of

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	$3,71\%$	$3,39\%$	$9,33\%$	$7,06\%$

Table 4: Throughput penalty

hardware thread supported by the target system. This way, for each workload/configuration tested during the training of the system, it is possible to determine the best performing concurrency level.

As we will show in section 5, the new approach achieves consistently better accuracy that the approaches in [12] and [13]. A relevant advantage of the new approach, beyond its higher accuracy, consists of its simplicity. On the other hand, a drawback with respect to our previous approach — to which we refer to in the following as, 2-layered approach— is that, differently from [12] and [13], it doesn't allow to estimate the absolute performance achievable when using a different degree of parallelism, which could be useful, for instance, to support what-if analysis. So the 2-layered approach is preferable in environments where it is necessary to make predictions in a wide set of scenarios (e.g. resource allocations in cloud environments).

5 Experimental Results

To evaluate the effectiveness of the approaches in [13], [12] and the new classification based approach we developed an adaptive prototype adding a statistic collector and a concurrency regulation module to the STAMP[16] benchmark suite. We executed our test on top of system equipped with an Intel Haswell Xeon E3-1275 3,5 GHz processor (8 virtual core: 4 physical with hyper-trading) with 32 GB RAM. Using intel TSX extension can append that a transaction, for different reasons (e.g. the size of the transaction dataset exceed the available L1 cache), can not be executed in hardware. In this case a software execution of the transaction is required. For our first experiments, as suggested by Intel, we use global lock acquisition that ensure the atomicity for software transactions on the fall-back path.

During the benchmark execution the statistic collector monitors the application and collects data about the workload profile. These data are then used runtime by the concurrency regulation module to schedule the application thread on top of the HTM. The table 4 shows the mean penalty, respect to the optimal throughput, due to the choice of the wrong concurrency level choices. The first and the second columns, with labels classification-DT and classification-NN show results for the classification approach implemented respectively with decision three algorithm and neural network. The third column shows results for the Neural Network based 2-steps approach in which the correction function is obtained with linear regression. The fourth column shows results for the Neural Network based 2-steps approach in which the correction function is obtained using the ratio between the mean values of the sampled parameters at each level of parallelism.

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, which report (just like in Table 3) the accuracy achievable by using the 2-layered approach, it emerges clearly that the proposed classification approach can achieve significantly higher accuracy than the previously employed solutions: 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 approaches is of about $9,33\%$ when using a linear correction function and of approximately $7,06\%$ when using Neural Networks.

The graphs in figure 2 shows the application speedup with respect to an uninstrumented sequential version, while varying the degree of parallelism, for two benchmarks of the STAMP suite, respectively Intruder and Vacation. When running the non-adaptive version of the benchmark, we fix the degree of parallelism statically; 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 self-tuning mechanism adjust the parallelism level according to the indications of the performance model.

For the Intruder benchmark, 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. Over 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 thread greater than 4, the adaptive version ensures the same speed-up that the appli-

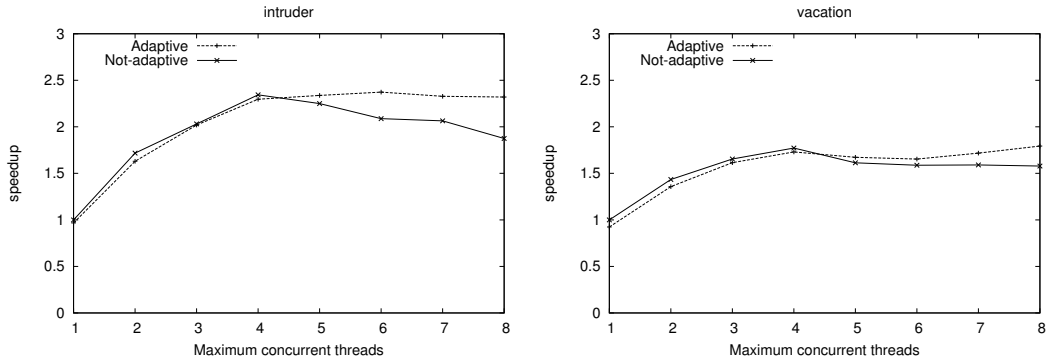

Figure 2: speedup

cation can reach when it is executed with the optimal concurrency level. Similar results can be obtained with the Vacation benchmark as showed by the second graph.

6 Conclusion and future work

In this report we presented the results of a study aimed at evaluating the applicability of already developed concurrency regulation techniques, originally conceived for STM systems, when applied to HTM. We showed that these techniques can suffer of relevant drawbacks, which are significantly exacerbated in HTM context.

Next, we presented a new approach for concurrency regulation based on classification techniques that are simpler and allow to obtain better performance with respect to the previously developed ones. We presented preliminary experimental results executing the STAMP benchmark suite on top of an autonomic concurrency regulation prototype that uses global lock when a transaction can not be executed in hardware. Moreover we have implemented another prototype that use NOrec STM[17] on the fall-back path. The next step will be to evaluate the new techniques using the second prototype.

References

[1] Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. volume 69, pages 187 – 205, 2012.

[2] Zhengyu He and Bo Hong. Modeling the run-time behavior of transactional memory. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2010 IEEE International Symposium on*, pages 307 –315, aug. 2010.

[3] Aleksandar Dragojević and Rachid Guerraoui. Predicting the Scalability of an STM: A Pragmatic Approach, 2010.

[4] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th international Euro-Par Conference on Parallel Processing*, pages 719–728, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

[6] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 7–16, New York, NY, USA, 2009. ACM.

[7] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. volume 44, pages 155–165, New York, NY, USA, June 2009. ACM.

[8] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[9] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[10] Qingping Wang, Sameer Kulkarni, John V. Cavazos, and Michael Spear. Towards applying machine learning to adaptive transactional memory. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[11] Marcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[12] 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*, pages 278–285, Washington, DC, USA, 2012. IEEE Computer Society.

[13] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Regulating concurrency in software transactional memory: An effective model-based approach. In *Proceedings of the 7th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, 2013.

[14] Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Jrg Schenker. Identifying the optimal level of parallelism in transactional memory applications. In *NETYS*, Lecture Notes in Computer Science, pages 233–247. Springer, 2013.

[15] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[16] Chi C. Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008.

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