Adaptive Transactional Memories: Performance and Energy Consumption Tradeoffs

Diego Rughetti
rughetti@dis.uniroma1.it

Pierangelo Di Sanzo
disanzo@dis.uniroma1.it

Alessandro Pellegrini
pellegrini@dis.uniroma1.it

DIAG — Sapienza, University of Rome

***Abstract*—Energy efficiency is becoming a pressing issue, especially in large data centers where it entails, at the same time, a non-negligible management cost, an enhancement of hardware fault probability, and a significant environmental footprint. In this paper, we study how Software Transactional Memories (STM) can provide benefits on both power saving and the overall applications' execution performance. This is related to the fact that encapsulating shared-data accesses within transactions gives the freedom to the STM middleware to both ensure consistency and reduce the actual data contention, the latter having been shown to affect the overall power needed to complete the application's execution.**

We have selected a set of self-adaptive extensions to existing STM middlewares (namely, TinySTM and R-STM) to prove how self-adapting computation can capture the actual degree of parallelism and/or logical contention on shared data in a better way, enhancing even more the intrinsic benefits provided by STM. Of course, this benefit comes at a cost, which is the actual execution time required by the proposed approaches to precisely tune the execution parameters for reducing power consumption and enhancing execution performance. Nevertheless, the results hereby provided show that adaptivity is a strictly necessary requirement to reduce energy consumption in STM systems: Without it, it is not possible to reach any acceptable level of energy efficiency at all.

I. Introduction

The issue of reducing energy consumption in high-performance multiprocessor systems is quickly becoming pressing. Most of the effort is directed towards the reduction of the environmental footprint of large data centers, the energy consumption of which has increased along with performance increase. This trend is not expected to come to a rest, as business needs demand for an always increasing performance. The United States host approximately 6,000 data centers, which consumed roughly 61 billions kWh of energy in 2006, about 1.5% of all U.S. electricity that year [1]. Overall, energy inefficiency has effects on several sides, e.g. overall management cost, heat waste, operating electronics failures and environmental implications [2]. These joint aspects require novel approaches to wisely exploit computing resources, to tackle at the same time business' high performance requirements, and energy saving. The problem is actually twofold: on the one hand, hardware implementations and their organization can provide a benefit. On the other hand, software optimization can capture aspects related to the actual dynamic workload which can in turn affect the hardware itself.

A large fingerprint on the overall power requirements to execute applications on single-core processors is due to memory accesses. On multicore architectures, this issue is even exacerbated by the need for synchronization of shared memory accesses. This means that both in uniprocessor and multiprocessor architectures, the memory hierarchy is one of the main sources of power dissipation [3], which therefore goes hand in hand with the overall application's performance, as spending more time in synchronization has non-negligible effects on both sides.

In this scenario, Software Transactional Memories (STM) [4], [5] can be regarded as a building block for enhancing performance while reducing the overall power consumption in large data centers. In fact, they have been explicitly designed to improve the overall performance of applications' execution by efficiently exploiting the available hardware parallelism, by relying on speculative execution. At the same time, they have been already shown to provide advantages in terms of energy consumption over locks [3]. Additionally, as STM is the technology already at the heart of several in-memory Cloud-suited data platforms (e.g. [6]), where the encapsulation of application code within transactions allows manipulating in-memory application data according to specific isolation levels across concurrent tasks, energy-efficient approaches can be easily integrated within such platforms. STM are a programming paradigm that borrows from the experience gained in the Database Management Systems (DBMS) field to provide the application programmer with synchronization facilities that leverage from the definition of complex critical sections to enforce correctness of multi-threaded applications. In particular, by relying on the notion of atomic transactions, STM allow the application programmer to access shared data structures without the need for, e.g., traditional locking primitives.

Coherency of the data/access manipulation is therefore demanded to the STM layer, rather than to any handcrafted synchronization scheme demanded from the programmer. This gives the possibility to implement optimized algorithms to manage shared data accesses by means of proper transactions' scheduling and/or proper selection of the number of parallel threads involved in the execution, which can in turn provide a non-negligible reduction in the overall power consumption. This is strictly related to careful exploitation of the parallelism exposed by the application, avoiding thrashing phenomena due to excessive contention on logical resources. Two different sets of orthogonal solutions have been provided in literature. On the one hand, we find optimized schemes for transaction conflict detection and management [7], [8], [9], [10], [11], which aim at dynamically determining which threads need to execute specific transactions, sequentializing the ones expected

to access the same data. Other proposals rely instead on proactive transaction scheduling and thread-level scheduling [12], [13], where performance degradation is avoided by delaying the scheduling of transactions with a high estimated conflict probability. This approach takes into account the non-transactional code cost as well.

On the other hand, we find solutions which support performance optimization by determining the optimal number of threads to be used for running the application on top of the STM layer [14], [15], [16]. This selection of the best-suited level of concurrency is clearly orthogonal to the aforementioned ones, being potentially usable in combination. The key question, here, is how much energy or power is really needed to get some computation done? It is easy to imagine that there is a minimum amount of electrical energy needed to perform a certain task, yet the benefit coming from STM's wiser usage of hardware resources can strictly depend on the system architecture [3] and/or the runtime workload dynamics.

It is therefore predictable that self-adaptive solutions within STM layers can provide even higher benefits. An adaptive software [17] basically uses available information about changes in its environment to improve its behavior over time. In the context of STM, this has been realized in literature by relying either on analytic methods, or on black-box Machine Learning (ML) methodologies (e.g., [14]). The former have the advantage of generally requiring a lightweight application profiling to gather the data in support of the prediction model, but provide slightly less accurate predictions. Additionally, in some cases they require stringent assumptions to be met by the real systems in order for its dynamics to be reliably captured. On the contrary, ML methods usually require much expensive (offline) profiling in order to build the knowledge base needed to instantiate the performance prediction model, which may make the actuation of the optimized concurrency configuration untimely. Yet, they provide a more precise estimation of the real performance trends, as shown in differentiated contexts.

In this paper we present an empirical study of various implementations of STM layers, namely TinySTM [8] and R-STM [18], configured with differentiated adaptive algorithms. By this study, we show how different approaches are able either to detect shared data access conflicts or to capture different levels of intrinsic parallelism shown by applications taken from the STAMP benchmark suite [19]. By the results, we show that if STM layers do not offer adaptivity facilities, it is definitely not possible to offer a good tradeoff level between efficiency and power saving. A significant reduction in the overall power consumption of a large data center offering STM-based platforms, without sacrificing efficiency, can be only obtained via adaptivity, which therefore becomes an essential building block. We additionally show what is the cost in terms of computing power required by these solution to work, allowing a careful selection of the best suited approach when configuring a data center. The remainder of this paper is structured as follows. In Section II Related Work is discussed. An overview of the reference STM architectures and of the adaptive schemes used is presented in Section III. Experimental data are shown in Section IV.

II. Related Work

The works in [3], [5] address energy reduction in multi-processor systems when STM are used in place of traditional locks, one addressing micro benchmarks [3], and one addressing STAMP benchmarks [5] (in a configuration using only 8 concurrent threads). The works focus on energy consumption issues due to accesses to shared memory, and show how transactional memories can, in practice, provide advantages in terms of enery consumption over locks. We complement the results provided by both works, showing how different levels of adaptiveness can capture aspects related to the system architecture, contention level, and policy of conflict resolution, so that STM layers can provide additional benefits over locks due to their different internal implementations/configurations without any need to alter the application-level code, even in the case where the hardware setup offers a high number of available cores.

The effects of shared data accesses on energy consumption is analyzed as well in [20], where a variation of barrier synchronization is used to reduce the energy required for spinning on a barrier, by relying on a software predictor that decides in which low power mode to place the CPU when the barrier is hit. On the other hand, our proposal studies the impacts on energy consumption in scenarios where alternatives to the spin loop itself (i.e. STM) are used.

The work in [21] proposes a memory-aware resource allocation algorithm that minimizes energy consumption by reducing contention conflicts while maximizing performance. A Min-Min resource allocation algorithm [22] is used upon application scheduling, which computes completion time for each job on each core, then selects the one with the minimum completion time. The application with the overall minimum completion time is then stuck to the core.

In [23] transactional memories are used as a means for supporting consistency in the execution of applications when running on system with reduced voltage supply close to transistors' operating threshold, or even below it, for energy saving purposes. The authors study how transactional memories can be combined with different error detection mechanism suitable for detecting yield losses, hard errors, erratic bits, and soft errors, and propose an analytic model to select what is the best scope of the transactions and the transactions' granularity. Differently from this work, we empirically study how transactional memories can provide benefits on energy consumption when running applications on regularly-powered processing units.

III. Reference Architectures and Configurations

We hereby present a discussion of the STM implementations and the self-adaptative configurations that we have used to carry on our empirical study.

A. TinySTM

The basic setup of TinySTM [8] (which we have used as a baseline configuration for our evaluation) implements the Encounter-Time Locking (ETL) algorithm. It is an algorithm essentially based on locks, which are acquired (on a per-word basis) whenever a write operation is to be performed on a shared variable. In particular, TinySTM relies on a shared array

of locks, where each lock is associated with a portion of the (shared) address space. Upon a write operation, the transaction identifies which lock is covering the memory region which will be affected by the write, and atomically reads its value. A lock value is composed by an integer number, the least significant bit of which tells whether the lock is currently owned by a running transaction. The remaining bits specify the current version number associated with that particular memory region. The writing transaction, therefore, reads the lock bit and determines whether that memory region is already owned or not. In the positive case, the transaction checks whether the lock its owned by the transaction itself. If it is so, the new value is directly written, otherwise for a specified amount of time the transaction gets into a waiting state. In the negative case, an atomic compare-and-swap (`CAS`) operation is used to try to acquire the lock. A failure in the `CAS` operation indicates that the lock has been (concurrently) acquired by another transaction, so the execution of the write operation falls into the first aforementioned case. The current version number stored in the lock is used upon read operations, to check whether the memory region has been updated by other transactions. In particular, before reading, the transaction checks whether the lock is owned, then reads the counter value, then performs the read operation, and then reads the counter value again. If between the two counter reads its value is not changed, then the read value can be regarded as consistent.

We have used TinySTM relying on the *write-back* scheme for the propagation of memory updates. It buffers all the updates in a *write log*, which, upon commit operation, is flushed to memory. This approach reduces the abort time and simplifies guaranteeing consistency of read operations. ETL has two main advantages [8]: On the one hand, by detecting conflicts early it can provide a transaction throughput increase, reducing the amount of wasted work executed by transactions. On the other hand, read-after-writes can be handled efficiently, providing a non-negligible benefit whenever write sets are large enough.

B. TinySTM Adaptive Configurations

1) SAC-STM [14]: this algorithm exploits a machine-learning based controller which regulates the amount of active concurrent threads along the execution of the application. Specifically, a neural network [24] is trained in advanced to learn existing relations between the average wasted transaction execution time (i.e. the average time spent by a thread executing aborted transactions for each committed transaction) and: (i) a set of parameters representing the current workload profile of the application, and (ii) the number of active concurrent threads. The set of workload profile parameters includes the average transaction read-set/write-set size, the average execution time of committed transactions and non-transactional code blocks, and two indices providing an estimation of the probability that an object being read/written by a transaction is also written by other concurrent transactions. The neural network is exploited by the controller to estimate the optimal number of concurrent threads to be kept active. This is done by evaluating the expected throughput as a function of the application's current workload and the number of active threads k, with $1 \leq k \leq max_{thread}$, where max_{thread} is a system configuration parameter denoting the maximum number of concurrent threads which can be activated[1]. The controller also exploits a neural network to predict the system's hardware scalability as a function of the number of active concurrent threads, of the time spent in transactions, and of the time spent in non-transactional code. The optimal number of threads is periodically re-estimated. In more detail, at the end of each workload sampling interval, during which all workload profile parameters are evaluated on the basis of statistics collected through runtime workload measurements, the controller performs new throughput estimations and then activates only a certain number of threads so that throughput is expected to be higher. We note that, in order to train the neural networks, SAC-STM requires an initial profiling phase, where statistics (to be exploited as training data) have to be collected while varying the workload profile and the number of active threads. The effectiveness of the concurrency regulation depends on the prediction accuracy of the neural networks, which, in turn, depends on the availability of training data providing a good coverage of the actual domain of the neural networks' input parameters.

2) SCR-STM [15]: this algorithm is similar to SAC-STM, except that, in place of machine learning techniques, an analytic model is exploited by the controller to regulate the concurrency level. Particularly, a parametric analytic model specifically built for estimating STM applications' performance is exploited to predict the average transaction's wasted time as a function of the same parameters as in the case of SAC-STM. SCR-STM also exploits a parametric analytic model for predicting the system's hardware scalability. Both parametric models must be instantiated via regression analysis, where the parameters are estimated by exploiting statistics collected while varying the workload profile and the number of active threads. However, as it has been shown, SCR-STM typically requires very few statistics to instantiate the models with respect to neural networks in SAC-STM. On the other hand, the accuracy of the machine-learning based prediction of SAC-STM typically outperforms the analytic model-based prediction of SCR-STM when there are a large number of statistics providing good coverage of the actual domain of the neural networks' input parameters.

3) ATS-STM: this algorithm is based on Adaptive Transaction Scheduling (ATS) [13], a transaction-scheduling algorithm relying on runtime measurement of the Contention Intensity (CI). It is a dynamic average along the application execution, and is re-calculated whenever a transaction commits or aborts using the equation $CI_n = \alpha \cdot CI_{n-1} + (1-\alpha) \cdot CC$, where CI_n is the current value of CI, CI_{n-1} is the previous one, α is the weight coefficient and CC is equal to 1 if the last executed transaction has been committed, 0 otherwise. The value of α is in between 0 and 1, determining how much CI is affected by past transactional history. Each thread maintains its own contention intensity. Before starting a new transaction, if the current value of CI exceeds a given threshold, then the thread stalls and the transaction is inserted within a queue shared by all threads. Otherwise, the threads immediately execute the transaction. Transactions stored inside the queue are serialized and executed according to the FIFO order.

[1]This is usually set to the total number of available processing units/CPU cores available in the system.

4) Shrink [25]: is another transaction-scheduling algorithm, which is based on temporal locality, i.e. on the fact that consecutive transactions in a thread access the same data objects. Similarly to ATS-STM, in Shrink the scheduler is activated if the transaction success rate is below a given threshold. Yet, rather than serializing all transactions, a contention probability is evaluated on the read- and write-sets. Specifically, before starting a new transaction, an estimation of the probability of write contention among the entries in the predicted read-/write-sets is computed. This is done by checking if there is an intersection between the predicted write-set of (predicted) concurrent transactions and the union of the read-set and the write-set of the transaction to be executed. In the positive case, the new transaction must be serialized. The predicted read-set of starting transactions is the union of the read-sets of the last n executed transactions, where n is a configuration parameter called *locality window*. The predicted write-set of a transaction includes all the data objects written by the same transaction during previous executions (if any, otherwise the write-set in assumed to be empty).

C. R-STM

Robust Adaptivity STM (R-STM) [18] is a transactional memory middleware which allows adaptivity on two different sides. On the one hand, it implements a coarse-grained adaptivity system, that allows to change the STM implementation being used during the execution of the application. On the other hand, once a particular STM implementation is selected, it allows to fine tune the execution parameters for an active transaction. The selection of a particular STM implementation is realized in a fast, efficient way via four function pointers. When a thread decides to start switching to a different implementation, all the threads wait until in-flight transaction either commit or abort. Then, all per-thread function pointers allowing the access to the current STM implementation's `TMRead`, `TMWrite`, and `TMCommit` functions are updated to reflect the selection of the new implementation. Finally, a global function pointer to `TMBegin` is updated as well, allowing a newly starting transaction to benefit from the re-selection. Transitions happen when a specific API is called at any point from the application, when a Mutex transaction determines (at commit time) that it was blocked for too long, or when transactions abort too often. We have evaluated R-STM when using the extension presented in [26], which bases its adaptivity on a ML algorithm. This algorithm is based on offline training to learn relationships between the overall application throughput and features like *shared memory accesses*, *non-transactional work*, and *write frequency*. A simple runtime, ProfileTM, is used to monitor (via a hardware tick counter) transaction running time, while preventing concurrent execution of transactions, via a fair ticket lock. This allows a faster transactional run. Using the information learnt during the training phase, R-STM selects an initial algorithm taking as input the required semantics of the application. During the execution, three events can trigger the reevaluation of the STM algorithm being used, i.e. when a new thread is spawn, when a thread aborts more than 16 times, or when a thread block for more than 2048 cycles when attempting to start a new transaction. The reevaluation of the STM algorithm is supported by switching again on ProfileTM, and executing N consecutive transaction to dynamically refine the adaptivity policy.

IV. Experimental Data

In this section we present the results of an experimental study carried out to compare performance results and power consumption of a selection of applications from the STAMP benchmark suite [19] on top of the above described STM implementations and adaptive configurations. The experimental setups have been hosted by an HP ProLiant server equipped with two AMD Opteron™ 6128 Series Processor, each one having eight hardware cores (for a total of 16 cores), and 32 GB RAM, running a Linux Debian 6 distribution with kernel version 2.7.32-5-amd64. This hardware architecture has been used to run only our transactional applications, mimicking a scenario where dedicated hardware is used to carry on scientific experiments without concurrent tasks being scheduled by the operating system (and thus affecting the overall results). The selected STAMP applications are kmeans, intruder and yada.

intruder is an application which implements a signature-based network intrusion detection systems (NIDS) that scans network packets for matches against a known set of intrusion signatures. In particular, it emulates Design 5 of the NIDS described in [27]. Three analysis phases are carried on in parallel: *capture*, *reassembly*, and *detection*. The capture and reassembly phases are each enclosed by transactions, which are relatively short and show a contention level which is either moderate or high, depending on how often the reassembly phase rebalances its tree. Overall, the total amount of time spent in the execution of transactions is relatively moderate. Therefore, we expect that the reduction of energy consumption is relatively limited, while allowing us to evaluate how much the overhead induced by the different self-adaptive solutions impacts on the overall execution performance.

kmeans is a transactional implementation of a partition-based clustering algorithm [28]. A cluster is represented by the mean value of all the objects it contains, and during the execution of this benchmark the mean points are updated by assigning each object to its nearest cluster center, based on Euclid distance. This benchmark relies on threads working on separate subsets of the data and uses transactions in order to assign portions of the workload and to store final results concerning the new centroid updates. Given the reduced amount of shared data structures being updated by transactions, in this benchmark it is more likely to incur in logical contention when a larger number of threads is used for the computation. Therefore, this application benchmark is a good candidate to study how changing workload dynamics can affect both performance and energy consumption when scaling up a transactional application.

yada implements Ruppert's algorithm for Delaunay mesh refinement [29], which is a key step used for rendering graphics or to solve partial differential equations using the finite-element method. This benchmark discretizes a given domain of interest using triangles or thetraedra, by iteratively refining a coarse initial mesh. In particular, elements not satisfying quality constraints are identified, and replaced with new ones, which in turn might not satisfy the constraints as well, so that a new replacement phase must be undertaken. This benchmark shows a high level of parallelism, due to the fact that elements which are distant in the mesh do not interfere with each other, and operations enclosed by transactions involve only updates

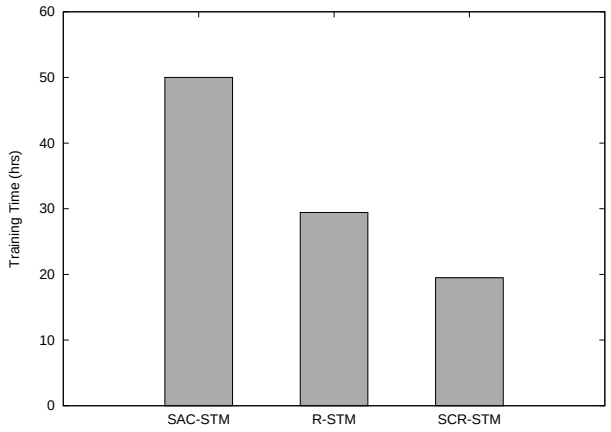

Fig. 1. Training Time in hours, intruder benchmark

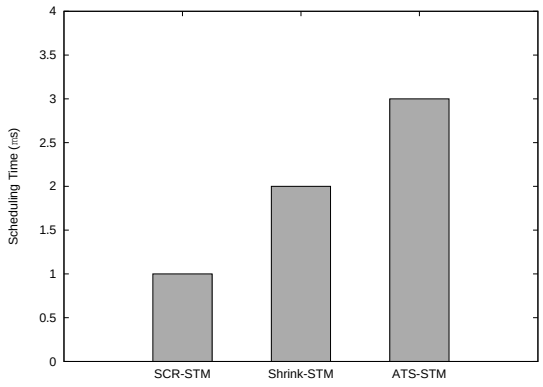

Fig. 2. Scheduling Time (per transaction) in μs, intruder benchmark

of the shared mesh representation and cavity expansion. The overall execution time of this benchmark is relatively long, showing a high duration of transaction operations and a significantly higher number of memory operations. Overall, this is a good candidate application to measure the impacts of power consumption when transactional memory middlewares must manage a very high level of logical contention.

Figure 1 reports the training time (averaged over 5 different runs) related to the intruder benchmark for all the configurations which actually rely on pre-computed values to support the re-selection of the number of active threads (namely, SAC-STM, R-STM, and SCR-STM). By the results, it is clear that all the approaches require a non-negligible time (all in the order of some hours) to explore all the possible configurations in order to determine the values of the coefficients internally used to perform the re-selection. Moreover, we note that SAC-STM has a training time which is more that twice the one required by SCR-STM. Nevertheless, results in Figures 3, 4, and 5 (which will be discussed later) show that SAC-STM is far more precise in the re-selection of the active threads with respect to the other two configurations. We emphasize that, given a particular application, training time is a cost which is paid only once. Therefore, if one application is non-desultorily used, the benefit gained by relying on SAC-STM (in terms of both performance and energy efficiency) can quickly repay the cost needed by actual training. As for the configurations which do not rely on any pre-computed value (namely SCR-STM, Shrink-STM, and ATS-STM), in Figure 2 we present results (related again to the intruder benchmark) associated with the time required by these approaches to schedule a transaction. This is, therefore, a per-transaction cost, which affects the overall throughput depending on the actual transactional workload, which can be very different depending on the application's configuration. While SCR-STM shows the smallest overhead, we note that this approach requires as well a training time, as discussed before. Therefore, if an application's configuration exhibits a very high transactional profile, SCR-STM has to compensate both its training time and its scheduling time to provide effective benefits on performance and energy consumption.

In Figures 3, 4, and 5, we report measurements related to per-transaction energy consumption (in joule/Transaction), Throughput/Energy consumption ratio, and overall applica-

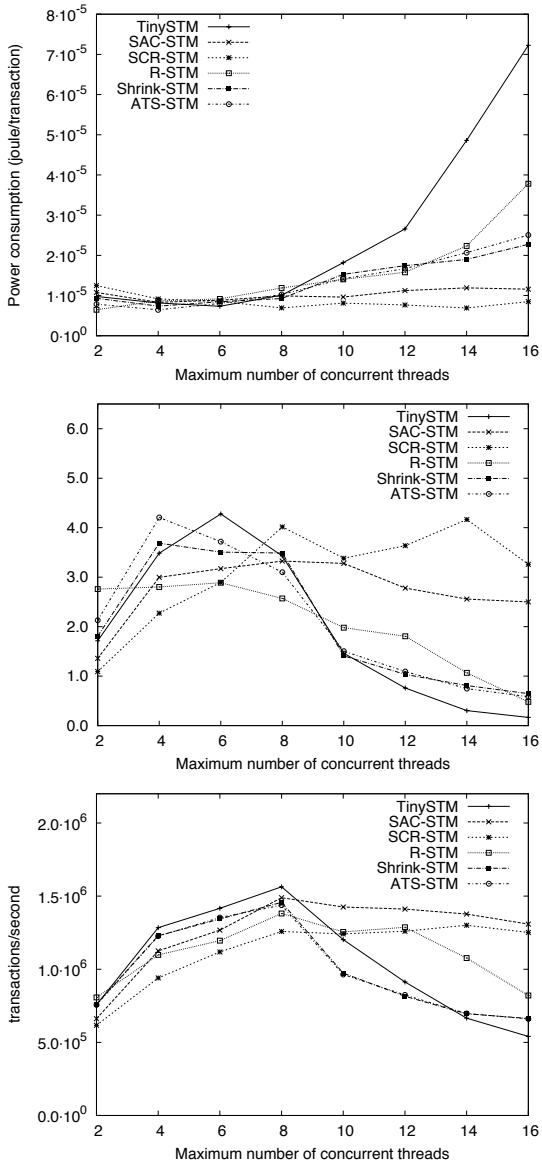

Fig. 3. Experimental results for the intruder benchmark

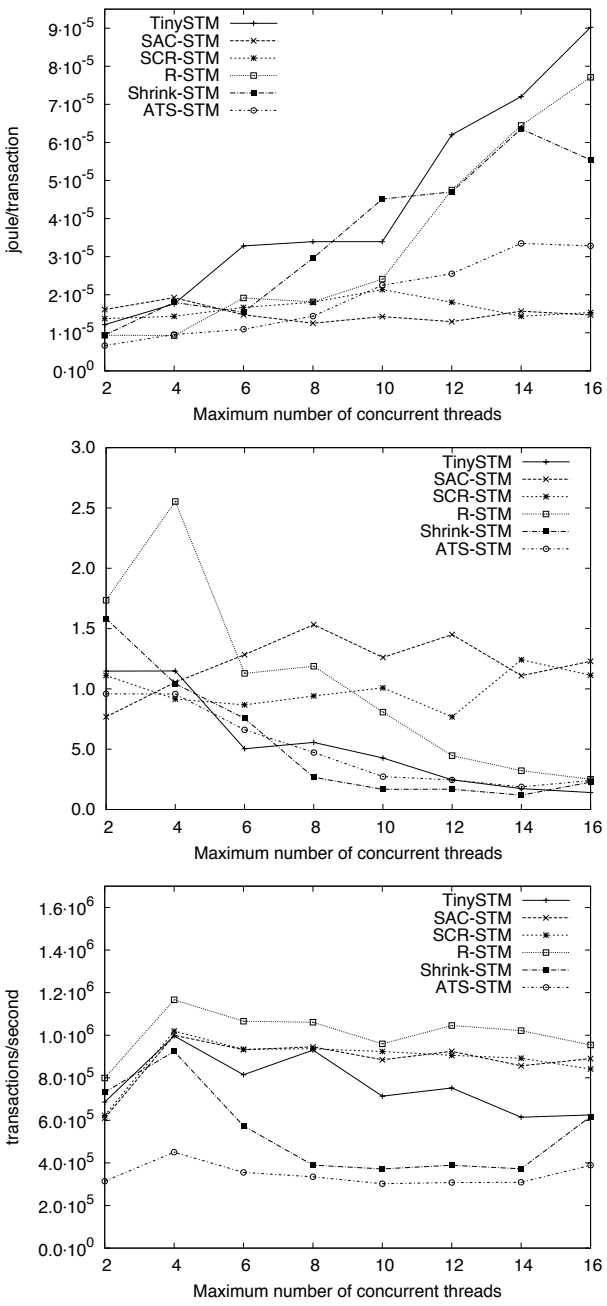

Fig. 4. Experimental results for the kmeans benchmark

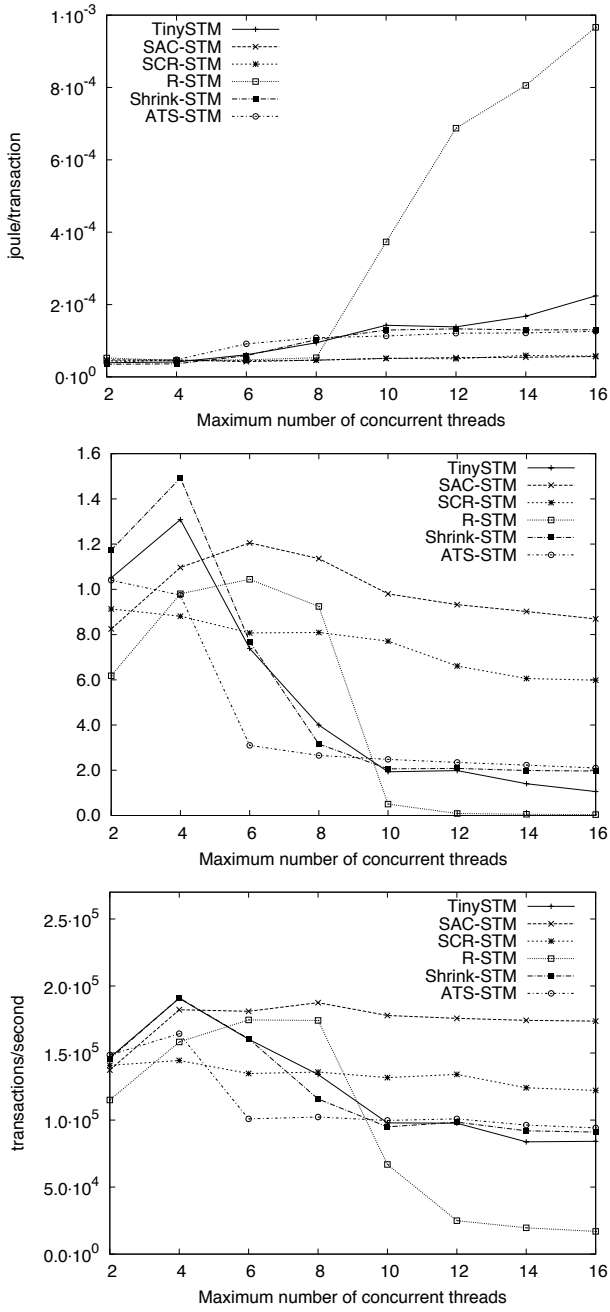

Fig. 5. Experimental results for the yada benchmark

tion throughput, associated with the above-discussed STAMP benchmarks. We have measured energy consumption by relying on the pTop tool [30], which has been modified to allow on-file statistics logging. pTop estimates the energy consumption of a specific application[2] indirectly through its resource utilization. For each resource (in our study we consider CPU and memory only) the application's energy consumption is evaluated as function of its operating states (e.g., read or write for the memory) and its transitions across different operating frequency states. We have explicitly configured our benchmarks relying on input data which avoid both disk and network usage during application's steady state. The average per-transaction energy consumption—which is an index of how much power is required by the application benchmark to complete the execution of a single transaction— is computed, on the basis of the the current level of parallelism (i.e., the number of active threads). The Throughput/Energy consumption ratio is a measure to express the speedup per unit of energy, when considering that the unit of energy for committing a transaction is the one employed by the sequential run. Hence these curves express a kind of iso-energy speedup. Clearly, for the sequential run this curve has constant value equal to 1. Therefore, the higher the value, the higher performance

[2]In case of multiple applications being concurrently run on a single machine, pTop is able to gather per-process data by relying on Linux kernel Performance Counters [31] management architecture.

we are obtaining, without affecting energy consumption in the overall execution. This iso-energy speedup measurement can be regarded as well as an index useful for QoS-related assessment. In fact, it enables the system administrator to quantify how much energy can be saved (e.g., by downscaling the frequency associated with specific CPU cores running a transactional application) without violating the SLA entered into by customers and the provider.

The first graph in Figure 3 presents per-transaction energy consumption for the intruder benchmark. Despite the fact that in this benchmark the time spent in transactions is relatively small, we can see that the basic configuration of TinySTM shows a consumption which is from 2 to 7 times larger than the other approaches. SCR-STM and SAC-STM present the best results, although they are based on two different approaches (namely an analytic and a ML-based approach). This is related to the fact that both approaches are able to capture the best degree of parallelism exposed by the application, and therefore stick to a number of thread which is the optimal one (around 8 threads). At the same time, SAC-STM is able to capture dynamics related as well to non-transactional code. While this does not have a great impact on the overall per-transaction energy consumption (we recall that the time spent in transactions in intruder is very reduced), it does on the overall throughput (see third graph in figure 3), where SAC-STM offers the best performance. This is related as well to the fact that SAC-STM has no requirements to recalculate the scheduling of transactions, as discussed before. Nevertheless, in the second graph of Figure 3 we see that the best Throughput/Energy consumption is asymptotically shown by SCR-STM, due to the fact that it takes into account only measurements associated with transactional execution. Additionally, SCR-STM's analytic model is implemented in a way that the number of running threads finally selected is always floored down. This implies that the overall throughput is slightly lower than SAC-STM's, but energy consumption is reduced as well. This reduction is to an extent higher than the throughput's, so that the ratio of SCR-STM is better than SAC-STM's. Additionally, near the optimum we note that a small enhancement in performance produces a non-negligible increase in power consumption. This aspect is well captured by the analytic model included in SCR-STM's implementation. At the same time, we see that if the number of available threads is set below the optimum (i.e., 8 threads), the best ratio is shown by either ATS-STM or plain TinySTM. This is related to the fact that if the number of threads is below the optimum, then the contention on data is not yet the most significant impact factor on the overall performance and energy consumption. Therefore, approaches which are based on simple analytic methods (e.g., ATS-STM), or no approaches at all (e.g., standard TinySTM), which do not waste CPU cycles to perform additional housekeeping operations.

As for the kmeans benchmark, the high level of contention related to the reduced amount of shared data present in the application and the overall variable workload offered by the application, show more fluctuations on the mean energy consumption levels, as presented in the first graph of Figure 4. R-STM and ATS-STM are able to capture parallelism-related dynamics up to a certain number of threads (namely 12), but then fail to select the best-suited configuration. Plain TinySTM presents an energy consumption level which basically grows with the number of available threads, due to the fact that the contention level (mostly) linearly grows. An interesting behaviour is shown by Shrink-STM, which oscillates between very good and very bad energy consumption levels. This is due to the fact that Shrink-STM is a scheduling approach which is based on temporal locality, which is a relevant factor in kmeans. In fact, the number of shared variables is very reduced, and it is very likely that consecutive accesses will involve the same data. Nevertheless, this is the same for all the active threads, and therefore when the contention grows, it is not sure that a time-based approach can compensate between the linkeliness of working on the same data and interfering with other threads, hence the oscillating results. Again, SAC-STM and SCR-STM show the best results in terms of energy consumption, and Throughput/Energy consumption ratio, as expressed by the plot in the second graph of Figure 4. This has a direct impact on the overall throughput, as shown by the third graph of Figure 4. Nevertheless, the best result (in terms of performance) is shown by R-STM. This is because R-STM implements a coarse-grain adaptivity system, which is activated before a fine-grained system. Due to the variable workload shown by kmenans, R-STM is able to execute a more immediate change in execution dynamics which, while requiring slightly more energy, is more aggressive with respect to performance enhancement.

yada shows us the effects of an application whose execution is both long and with a high level of contention. By the results in the first graph of Figure 5, we can see that the overall per-transaction energy consumption is slightly higher if compared to the previous application benchmarks. Yet, the differences between the approaches is very reduced. In fact, if independently of the number of active threads the contention level is very high, different approaches can have a very reduced impact on the overall execution. This is even more clear if we see that SAC-STM, which is an approach able to detect the effects of non transactional code as well, determines that the best configuration for executing the application is using only one thread. This does not prove as the best choice in terms of energy consumption, but as it can be seen by the third graph of Figure 5, it shows the best throughput. An interesting behaviour is shown by R-STM, which, from a certain number of concurrent threads, shows a performance and an energy consumption level which are both very degraded. This is related to the fact that R-STM selects the overall transactional algorithm, rather than the parameters of a single algorithm. Therefore, it always uses all the available threads. In this scenario, with a very high contention, hardware contention can affect the overall execution with non-negligible secondary effects. In fact, hardware contention impacts the efficiency of both transactional and non-transactional code, with sensible effects on power consumption, related, e.g., to waiting time on memory bus.

By all the results, we can gather an important general result. In fact, we have (again!) shown that the impacts on both performance and energy consumption is somewhat related, so that a configuration which shows a reduced energy consumption is more likely going to behave in a more performing way. Nevertheless, we have also clearly shown that in differentiated configuration (i.e. both when the execution dynamics vary the contention level of the workload, and when the contention level is stable, being high or low), transaction scheduling

mechanisms are not sufficient for capturing the intrinsic degree of parallelism. Therefore, if service providers are really concerned about energy consumption of hosted applications, and their overall performance, adaptative solutions become mandatory. Of course, they require more computing power before the actual application is deployed, but in the long run the benefits shown by these approaches can fully repay the initial cost.

V. Conclusion and Future Work

In this paper we have presented an empirical study of the effects on execution performance and energy consumption of different self-adapting solutions regulating transactions' scheduling and proper selection of concurrent threads, on top of different STM middlewares. By the results, we have shown that adaptivity is an essential building block for creating STM systems which are energy efficient and that can offer a performance level which is better (or competitive) with other existing approaches.

References

[1] P. Kurp, "Green computing," *Communications of the ACM*, vol. 51, no. 10, pp. 11–13, Oct. 2008.

[2] P. Ranganathan, "Recipe for efficiency: Principles of power-aware computing," *Communications of the ACM*, vol. 53, no. 4, pp. 60–67, Apr. 2010.

[3] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy reduction in multiprocessor systems using transactional memory," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ser. ISLPED. IEEE Computer Society, 2005, pp. 331–334.

[4] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, Aug. 1995.

[5] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency," in *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, ser. HotPower. USENIX Association, 2012, pp. 1–6.

[6] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-TM: harnessing the cloud with distributed transactional memories," *SIGOPS Operating Systems Reviews*, vol. 44, no. 2, pp. 1–6, Apr. 2010.

[7] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proc. of the 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[8] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP. ACM, 2008, pp. 237–246.

[9] M. P. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. of the 20th annual international symposium on computer architecture*, ser. ISCA. ACM, 1993, pp. 289–300.

[10] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," *SIGPLAN Notices*, vol. 44, no. 4, pp. 141–150, Feb. 2009.

[11] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in *Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing*, ser. TRANSACT. ACM, 2009.

[12] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate," in *Proc. of the 14th international Euro-Par conference on Parallel Processing*, ser. Euro-Par. Springer-Verlag, 2008, pp. 719–728.

[13] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proc. of the twentieth annual symposium on Parallelism in algorithms and architectures*, ser. SPAA. ACM, 2008, pp. 169–178.

[14] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proc. of the 20th IEEE International Symposium On Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS. IEEE Comp. Soc., Aug. 2012, pp. 278–285.

[15] P. Di Sanzo, F. Del Re, D. Rughetti, B. Ciciani, and F. Quaglia, "Regulating concurrency in software transactional memory: An effective model-based approach," in *Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO. IEEE Computer Society, Sep. 2013.

[16] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *NETYS*, ser. Lecture Notes in Computer Science. Springer, 2013, pp. 233–247.

[17] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[18] M. F. Spear, "Lightweight, robust adaptivity for software transactional memory," in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA. ACM, 2010, pp. 273–283.

[19] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. of the IEEE International Symposium on Workload Characterization*, ser. ISWC, Sep. 2008.

[20] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA. IEEE Computer Society, 2004, pp. 14–.

[21] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "Memory-aware green scheduling on multi-core processors," in *Proceedings of the 39th International Conference on Parallel Processing Workshops*, ser. ICPPW. IEEE Computer Society, 2010, pp. 485–488.

[22] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

[23] A. Cristal, P. Felber, C. Fetzer, D. Harmanci, A. Sobe, O. Unsal, J.-T. Wamhoff, and G. Yalcin, "Leveraging transactional memory for energy-efficient computing below safe operation margins," in *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, ser. TRANSACT. ACM, Mar. 2013.

[24] T. M. Mitchell, *Machine Learning*, 1st ed. McGraw-Hill, 1997.

[25] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: Avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC. ACM, 2009, pp. 7–16.

[26] Q. Wang, S. Kulkarni, J. V. Cavazos, and M. Spear, "Towards applying machine learning to adaptive transactional memory," in *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[27] B. Haagdorens, T. Vermeiren, and M. Goossens, "Improving the performance of signature-based network intrusion detection sensors by multi-threading," in *Proceedings of the 5th International Conference on Information Security Applications*, ser. WISA. Springer-Verlag, 2005, pp. 188–203.

[28] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981.

[29] J. Ruppert, "A delaunay refinement algorithm for quality 2-dimensional mesh generation," *Journal of Algorithms*, vol. 18, no. 3, pp. 548–585, 1995.

[30] T. Do, S. Rawshdeh, and W. Shi, "pTop: A Process-level Power Profiling Tool," in *Proceedings of the Workshop on Power Aware Computing and Systems*, ser. HotPower. ACM, Oct. 2009.

[31] B. Sprunt, "The basics of performance-monitoring hardware," *Micro, IEEE*, vol. 22, no. 4, pp. 64–71, 2002.