CPU-core Frequency Scaling for Efficient Thread Scheduling in Transactional Memories

Pierangelo Di Sanzo
DIAG – Sapienza, University of Rome
Email: disanzo@dis.uniroma1.it

Bruno Ciciani
DIAG – Sapienza, University of Rome
Email: ciciani@dis.uniroma1.it

***Abstract*—Transaction Memory systems may suffer from performance degradation when the concurrency level grows. The transaction abort rate caused by high concurrency may be detrimental to energy efficiency as well. Thread scheduling techniques, which proactively block some threads to optimize the concurrency level, help to reduce these phenomena. In this paper, we show that the efficiency of mechanisms used by tread schedulers for blocking/unblocking concurrent threads can be improved using CPU-core frequency scaling options offered by modern hardware systems. Particularly, we study a low-frequency busy waiting approach, in which blocked threads scale down the frequency of CPU-cores where they are running. We compare this approach with two commonly used approaches by thread schedulers, and we demonstrate that it achieves the best results in term of both performance and energy efficiency.**

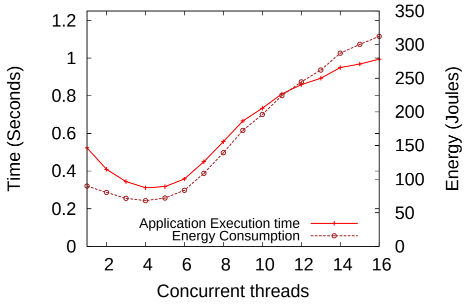

Fig. 1. Application Execution Time and Energy Consumption of Intruder.

I. Introduction

Over the last years, multi-core systems have become mainstream computing platforms. Consequently, the need for developing concurrent and parallel applications that can effectively exploit the hardware parallelism of these systems is increasingly common. Transactional Memories (TMs) simplify the problem of thread synchronization when developing concurrent applications. They allow programmers to mark as a transaction the code blocks executing shared data accesses that require to be executed in isolation (e.g. critical sections). At run-time, TMs guarantee transactions to be executed atomically without interferences on share data accesses with other concurrent transactions.

TMs use an optimistic approach for taking advantage of multi-core architectures. Transactions are allowed to execute in parallel, and conflicts on shared data accesses are resolved after that they occur by aborting and restarting one conflicting transaction. However, the performance of TM systems does not grow indefinitely while increasing the thread parallelism. Indeed, when the concurrency level excessively grows, the performance may even (drastically) decrease. This phenomenon is due to the high transaction conflict rate and to contention on shared hardware resources. Figure 1 shows some example data of an experiment executed with the Intruder application (included in the STAMP benchmark suite [1] for TMs) with a number of concurrent threads from 1 up to 16. The experiment was executed with TinySTM [2] on a 16-core machine. Results show that the application execution time decreases (i.e. the system performance increases) while incrementing the number of concurrent threads up to 5. After, its starts increasing due to the high concurrency level. In the same figure, we also report data related to the energy consumption of the machine during each run of the application. The shape of the energy curve is similar to the application execution time. Indeed, intuitively, the (wasted) energy used by CPU-cores to process (aborted) transactions is proportional to the (waste) time for processing them.

In TM systems, the phenomenon of performance loss due to high concurrency can be partially prevented by bounding the number of threads to the number of available CPU-cores [3]. However, such as shown in Figure 1, the performance degradation may arise even when there are less concurrent threads than CPU-cores.

One solution to cope with this problem relies on thread scheduling. In such a case, the TM system is equipped with a *scheduler*, which, based on a given scheduling strategy, can (temporarily) block some threads in order to reduce the concurrency level. Various thread scheduling techniques for TMs have been proposed in literature. They use alternative scheduling strategies for deciding *when* and *how long* threads should be blocked. For example, some techniques use feedback-based control strategies (e.g. [4], [5], [6]). Other techniques rely on performance models of TM applications (e.g. [7], [8], [9]).

Irrespectively of the adopted strategy, a common problem of thread schedulers is to minimize the impact on performance of operations required for thread blocking/unblocking. This issue has been evidenced, e.g., in [6] and [7]. An approach in which a thread to be blocked sleeps for a pre-determined time (e.g. by calling the *sleep(time)* POSIX/Unix-Like standard function) is generally unsuitable. Indeed, thread schedulers typically use on-line adaptive strategies, thus it is not possible to foresee when a thread should wake up in the future. If the sleep time is too long, the thread may wake up too late. Conversely, too short sleep times may lead to unnecessary overhead due to repetitive sleeping and waking up operations. We note that a scheduler could block even many threads over the total number of running threads, and that a thread could be blocked for a long time. Thus, the overhead associated to the above-mentioned operations could be relevant.

Two alternative approaches to block threads are commonly used by thread schedulers for TMs. Both of them have been analysed in [6] and [7]. The first approach is based on busy waiting, and the second one on semaphores. These approaches impose a trade-off between performance and energy consumption, as also confirmed by the results of the experimental study that we present in Section III. Specifically, the first one improves performance at the cost of energy efficiency. Indeed, it can be implemented by simply using per-thread synchronization flags, on which each thread spins when blocked. This avoids the overhead of any system call, and threads can immediately run when unblocked. However, it is energy-consuming, given that the CPU-core is kept busy by the blocked thread. With the second approach, the performance may be penalized due to the overhead associated to system calls for operating on semaphores. However, given that blocked threads can sleep until the semaphore is busy, the energy consumption of unused CPU-cores can be reduced due to the spontaneous activation of the per-core frequency scaling mechanism [10] offered by recent operating systems (that exploit the dynamic frequency scaling technologies of recent processors [11]).

To cope with the disadvantages of the above-mentioned approaches, we studied a low-frequency busy-waiting based solution, which exploits the chance for a thread to dynamically scale down/up the frequency of the CPU-core where it is running. Today, mainstream multi-core hardware systems offer the option to scale down/up the frequency of an individual CPU-core, which can be actuated via the Operating System support. Based on this facility, in our approach, the thread by itself scales down the frequency of the associated CPU-core upon entering a blocking phase. Then, when unblocked, it scales up the frequency and proceeds along its execution path. We remark that, as we already discussed, in TM applications it is generally convenient to bound the number of active threads to the number of CPU-cores to prevent performance loss (e.g. see [3]). Under this condition, it is possible to assign each thread to a different CPU-core (e.g. using CPU affinity functions, such as *sched_setaffinity* on Linux/UNIX OS), whose frequency can be regulated by the assigned thread. However, our approach is not limited by this condition. Indeed, it can be easily implemented even for the case where there are more threads than CPU-cores. More generally, it can be implemented for the case it is required to correctly restore the frequency of CPU-cores under scenarios with thread reschedule (e.g. when some thread of the application is context-switched off the CPU-core in favour of other threads of the same application, or threads of other applications/services running on top of the machine). In Section IV, we discuss a possible extension of the implementation of our approach to cope with these scenarios.

We conducted an experimental study to evaluate the efficiency of our approach in terms of both performance and energy consumption. We executed experiments using TinySTM equipped with a thread scheduler, and some applications of the STAMP benchmark suite. Results show that the low-frequency busy waiting mechanism that we implemented achieves the best, in terms of both performance and energy efficiency, of the two approaches that we described above. Finally, we note that, although we evaluated this approach in the case of TM applications, it could be used to optimize blocking/unblocking operations in thread scheduling techniques also for other kind of concurrent and parallel applications.

In the next section, we enter into details of the implementations we used to compare the different approaches. The experimental results are presented in Section III. In Section, we present the IV solution to cope with execution scenarios where frequency restore is required after thread reschedule. Finally, related work is discussed in Section V.

II. Implementations of the Evaluated Approaches

We consider a TM application running with N concurrent threads. The TM system is equipped with a thread scheduler that uses some scheduling strategy to adaptively change the number of threads k, with $k \leq N$, that are kept running along the application execution.

We assume that each thread can be assigned to a different CPU-core. A *coordinator thread* is in charge of blocking/unblocking threads according to the scheduling strategy ([1]). Each thread, before starting a new transaction, checks if it should be blocked or not.

When using the first busy waiting approach (the one without frequency scaling, we refer to as *basic* busy waiting approach), a thread checks an its own flag. The coordinator thread sets to 0 all flags of threads that are allowed to run (i.e. of threads that must not block), otherwise they are set to 1. Thus, a thread spins while its flag is equal to 1.

With the second approach, upon checking, a thread executes a *wait-for-zero* operation on a its own semaphore, whose value is set to 1 by the coordinator thread when the thread must block. Conversely, it is set to 0 when the thread is allowed to run. Upon the *wait-for-zero* operation, if the semaphore value is equal to 1, the thread transits into the sleeping state. It is woken up by the operating system when the value of the semaphore becomes 0.

With the low-frequency busy waiting approach, each thread uses the same its own flag, as with the basic busy waiting approach, which is modified by the coordinator thread. However, upon checking, if the thread finds the flag to be 1, it

[1]The coordinator thread could be a dedicated thread or one of the concurrent application threads, e.g. selected with a round robin strategy.

scales down the frequency of the CPU-core it is assigned to. Then, it continues spinning at low-frequency, as determined by the lower CPU-core speed. When the flag is set to 0 by the coordinator thread, the thread scales up the frequency and runs again.

III. Experimental Study

To compare the different approaches, we conducted a set of experiments with TinySTM and three applications of STAMP benchmark suite, including Intruder, Kmeans and Yada. We selected these applications since they show very different workload profiles, spanning from short to long transactions, from low to high contention level and from low to high time spent in transactions (see [1] for details). All experiments were executed on top of a 16-core HP ProLiant server with 2GHz AMD Opteron 6128 processors, 64 GB of RAM and Linux operating system (kernel version 2.7.32-5-amd64).

As for semaphores, we used the standard implementation of POSIX Semaphore APIs on Linux. As for frequency scaling operations, we used Linux CPUfreq subsystem [10]. With the basic busy waiting approach and the semaphore-based approach, we activated the governor ”ondemand”, which allows the operating system to decide when scaling down/up the frequency of each CPU-core depending on the individual utilization. With the low-frequency busy waiting approach, we activated the governor ”userspace”, which allows users/applications to modify the frequency of each CPU-core. This can be done by updating (pseudo) files of the Linux virtual file system under the path ”/sys/devices/system/cpu/...”. In our experiments, we used two frequency levels: 2GHz when the CPU-core was busy, and 0.8GHz when the CPU-core was idle or when a thread executes the scaling down operation. These levels were the maximum and the minimum one supported by the HP server of our study.

To assess the different approaches independently of the effectiveness of a specific thread scheduling strategy, we preventively executed some experiments using a static scheduling approach. Specifically, the number of non-blocked threads was pre-established and was not modified by the scheduler along the entire application execution. After, we executed other experiments using an adaptive scheduler based on an on-line strategy used in various scheduling techniques, such as those proposed in [5], [6], [12]. The strategy is inspired by the Hill Climbing search. The scheduler performs continues increments and decrements of the number of blocked threads in order to find the configuration for which the application throughput (in terms of transactions per time unit) is maximum. In short, the scheduler runs a continuous loop. For each step of the loop, if in the previous step it incremented (decremented) the number of blocked threads and the throughput has increased, it continues to increment (decrement) this number, otherwise the number is decremented (incremented). We call this scheduler *H-Scheduler*.

In our experiments, we configured H-Scheduler to completed a single step each time the coordinator thread executed 1000 consecutive transactions. After each step, once modified the number of threads to keep running, H-Scheduler randomly selects the specific threads to be blocked. Similarly, with the static scheduler, the specific threads to be blocked are randomly selected periodically, each time that the coordinator thread executed 1000 consecutive transactions.

In the rest of this Section, we show performance and energy data we collected with TinySTM without scheduling support (that we call ”baseline”) and with TinySTM for the cases of both static and on-line scheduler, for each one of the three approaches. Each benchmark application was run while varying the number of concurrent threads between 2 and 16

A. Results with static scheduling

In the experiments with static scheduling, results that we achieved with the different benchmark applications are similar and provide the same insights. Thus, for brevity, we show only results with Intruder. In Figure 2 we show results with TinySTM without scheduling support and with the static scheduler for configurations with 2, 5 and 8 non-blocked threads (we note that, when the number of concurrent threads is less or equal to the pre-established number of non-blocked threads, the scheduler does not actually block any thread). By the plots, both the application execution time and the energy consumption with the baseline rapidly grow with more than 5 concurrent treads. Conversely, the static scheduler avoids, or in the worst case reduces, this phenomenon with all three approaches. We note that, with the workload configuration of Intruder that we used in our experiments, 5 corresponds to the static number of threads providing the best performance. Thus, it represents the optimal static scheduling configuration. Conversely, 2 and 8 threads correspond to suboptimal scheduling configurations.

With the optimal static scheduling configuration (see central plots in Figure 2), the basic busy waiting approach and the low-frequency busy waiting one can preserve the maximum performance achievable with the baseline up to 16 concurrent threads. The performance with the semaphore-based approach deteriorates, on average, of about 10% with more than 5 concurrent treads. As for energy consumption, the best results are archived with the low-frequency busy waiting approach. Indeed, although the number of blocked threads grows while increasing the number of concurrent threads, the scaled down frequencies of CPU-cores keep the energy consumption close to the minimum level. With the semaphore-based approach, the energy consumption is slightly higher. With the basic busy waiting approach, the energy consumption constantly grows with more than 5 threads.

In the cases of 2 and 8 non-blocked threads (see left and right plots in Figure 2, respectively), the suboptimal scheduling configurations lead to worse results with all approaches with respect to the optimal scheduling configuration. In any case, data show that the low-frequency busy waiting approach still achieves the best results, in terms of both performance and energy consumption, with respect to the other approaches. This outcome shows that it is convenient to use the low-frequency

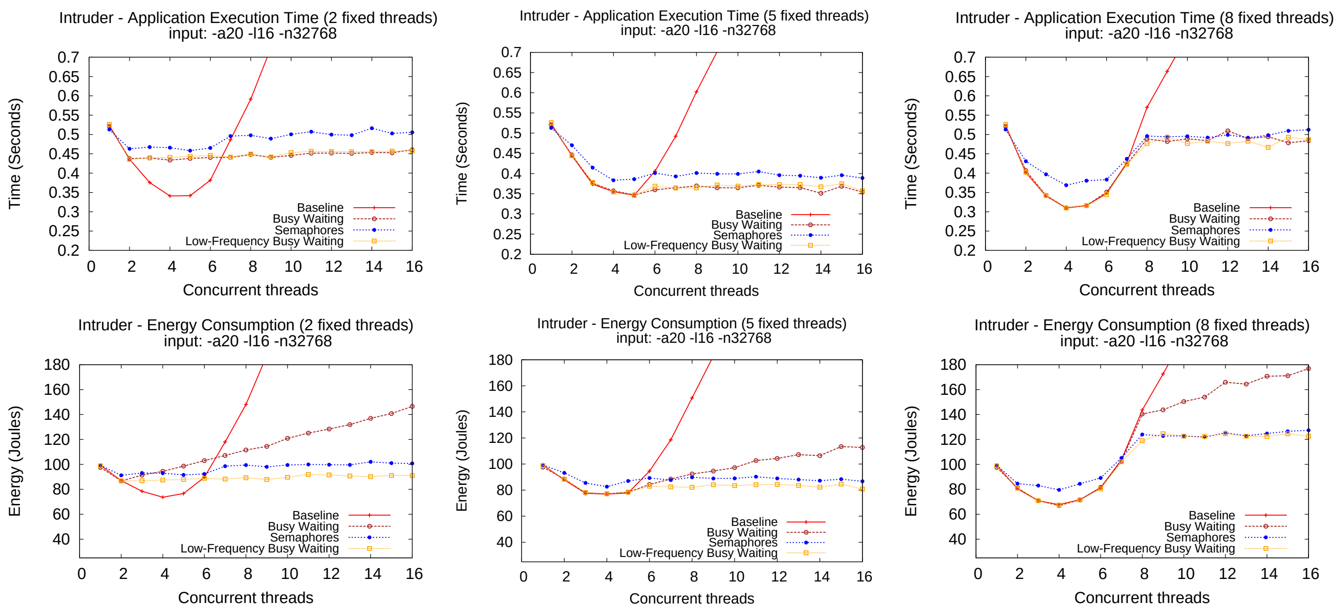

Fig. 2. Application Execution Time and Energy Consumption of Intruder with static scheduling.

busy waiting independently of the ability of a scheduler to find the best configuration.

B. *Results with the on-line adaptive scheduler*

Figure 3 shows the results with H-Scheduler for all benchmark applications. We note that with the baseline, even in the case of Kmeans and Yada, both the application execution time and the energy consumption rapidly grow after that the number of concurrent threads overcomes the optimal value (i.e. 3 for Kmeans and 7 for Yada). H-Scheduler partially reduces this phenomenon. We remark that H-Scheduler uses an on-line adaptive strategy, thus we can expect that it does not achieve results as good as the optimal static scheduling configuration. Also, the overhead of the online adaptive strategy is responsible of the slightly increment of the application execution time with the basic busy waiting and the low-frequency busy waiting approach with respect to the baseline (it can be observed with Intruder between 2 and 5 concurrent threads).

Overall, results show that the advantages with the low-frequency busy waiting approach are, on average, more evident in these scenarios. Indeed, the differences of the application execution times with respect to the semaphore-based approach are higher than in the case of static scheduling, particularly with Intruder. This is due to the overhead generated by the operations on semaphores, whose rate is higher due to the on-line adaptive strategy. The application execution times with the low-frequency busy waiting approach with respect to the basic busy waiting approach are similar for all applications. As for the energy efficiency, the semaphore-based approach reduces, on average, the energy consumption with respect to the basic busy waiting approach. On the other hand, the best results are still achieved with the low-frequency busy waiting approach. The advantages provided by this approach are particularly evident when the number of concurrent thread is high (e.g with 16 concurrent threads for Intruder and Yada).

Conclusively, experimental data of our study show that the low-frequency busy waiting approach provides the best results in term of both application execution time and energy efficiency compared to the other two approaches. Further, these advantages exist independently of the workload profile, which is notably different for the three applications used in our study, and independently of the effectiveness of the scheduler.

IV. Hints on how to cope with execution scenarios with thread reschedule

In this section, we discuss a possible extension to implement our approach to cope with execution scenarios where it is required to correctly restore the frequency of CPU-cores as a consequence of thread reschedule. Frequency restore may be required when, e.g., a blocked thread of the TM application has scaled down the frequency of a CPU-core, and then it is context-switched off the CPU-core before scaling up the frequency (i.e. before being unblocked). In this case, another thread of the same TM application, or a thread of other applications/services running on the same machine, would run at low CPU-core frequency if rescheduled on the same CPU-core. The solution that we describe also allows to cope with scenarios where a TM application is executed with more concurrent threads than CPU-cores, which necessarily leads threads of the TM application to be context-switched. The solution is based on a kernel-scheduler hooking mechanism, where a custom callback function is invoked as soon as a thread is rescheduled. Such a mechanism is used, e.g., in [13], [14]. In our case, when a thread is rescheduled on a given CPU-core, the frequency of the CPU-core can be scaled

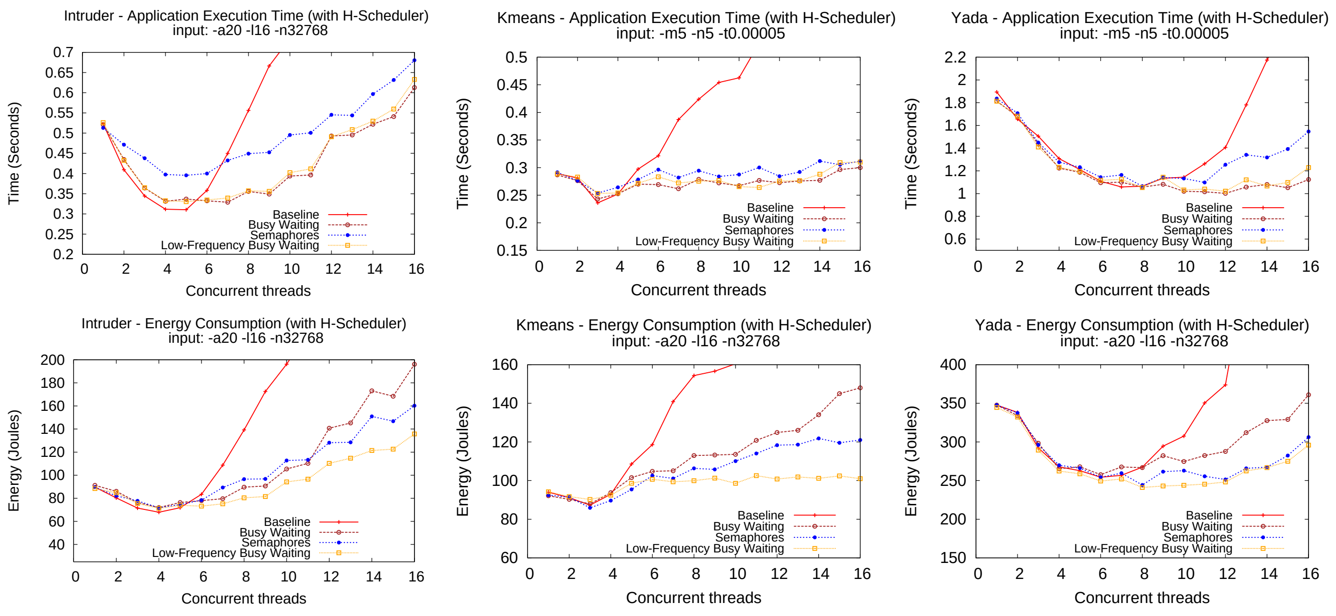

Fig. 3. Application Execution Time and Energy Consumption of Intruder, Kmeans and Yada with H-Scheduler.

up by the callback function. Further, this callback function can be implemented for a thread of the TM application such that, when the thread is rescheduled, it checks (by reading the thread'flag) if the thread is blocked. If it is not blocked and the CPU-core frequency is low, the frequency is scaled up, or vice versa. If the frequency is the correct one, it is not changed. The above-cited works showed that the overhead introduced by such call-back mechanisms, given the conventional thread schedule frequency of kernel-schedulers of mainstream operating systems, does not appreciably affect the performance of applications. Also, our experimental study showed that the overhead of operations for scaling down/up the CPU-core frequency is negligible in our approach. Thus, we expect the above described solution to be effective. We plan to integrate it in the implementation of our approach as a part of our future work.

V. Related Work

Scheduling techniques have been largely explored in the context of TMs with the aim of optimizing the application performance (e.g. in [4], [5], [6], [7], [8], [15], [9], [16]). On the other hand, energy efficiency of scheduling techniques for TMs has been poorly explored. Few studies analyse the impact on energy consumption of alternative techniques [17], [18]. Performance and energy efficiency of different TM implementations have been evaluated in [19]. Techniques for reducing the energy consumption in TMs have been studied out of the specific context of scheduling. In [20], the authors sketched out a mechanism based on clock gating [21] to reduce the energy consumption in Hardware Transactional Memories. This mechanism gates a processor when an hardware transaction aborts, and un-gates it based on the number of transaction aborts and the state of conflicting transactions. Still in context of Hardware TMs, in [22] the authors proposed alternative cache structures and contention management schemes to improve the energy efficiency of TM applications for embedded systems. Differently from the above-mentioned proposals, our approach can be easily adopted in the case of both Software and Hardware TMs, and does not require any architectural modification.

In [23], the authors present Green-TM, an energy-efficient contention manager for TMs. When a transaction gets aborted and the thread enters the back-off phase, Green-TM dynamically adapts the back-off time and decides if the tread must spin or sleep, with the aim of improving performance and energy efficiency. Further, Green-TM groups threads in two different sets: 1) threads that are likely to back off, and 2) threads that spend most of their time executing transactions. This separation aims at favouring the spontaneous activation of the operating system control mechanism for scaling the frequency of CPU-cores where the running threads are likely to back off. Differently from Green-TM, our low-frequency busy waiting approach is targeted to overcome drawbacks of both spin-based and sleep-based waiting mechanisms. Further, it directly controls the frequency of a specific CPU-core, without waiting for the activation of the operating system control, which is generally less effective given that it may require various (architecture/system dependent) pre-conditions to be verified.

VI. Conclusions

In this paper, we addressed the problem of optimizing performance and energy efficiency of thread scheduling in TMs. Particularly, we focused on the overhead associated to thread blocking/unblocking operations. We analysed the two approaches commonly used by thread schedulers proposed in literature, then we proposed a low-frequency busy waiting approach, in which threads scale down/up the frequency of the CPU-cores when they are blocked/unblocked. Results of our experimental study show that this approach is able to provide the best results in terms of both performance and energy efficiency in different execution scenarios. We also proposed a possible extension of our approach to cope with frequency restore requirements in the case of thread reschedule. We will integrate it in our implementation as a future work.

References

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Proc. 4th IEEE Int. Symposium on Workload Characterization*. IEEE, 2008, pp. 35–46.

[2] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proc. 13th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2008, pp. 237–246.

[3] R. Ennals and R. Ennals, "Software transactional memory should not be obstruction-free," Technical Report IRC-TR–06–052, Intel Research Cambridge Tech Report, Jan 2006, Tech. Rep., 2008.

[4] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, and I. Watson, "Adaptive concurrency control for transactional memory," in *In MULTIPROG 08: First Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[5] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory systems," in *Proc. International Conference on Networked Systems*, ser. NETYS. Springer, 2013.

[6] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for stms," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 927–938.

[7] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proc. 20th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2012, pp. 278–285.

[8] ——, "Dynamic feature selection for machine-learning based concurrency regulation in stm," in *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 68–75.

[9] 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 *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, Sept 2013, pp. 31–40.

[10] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[11] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, ser. HotPower'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924920.1924921

[12] K. Chan, K. Tin Lam, and C.-L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," in *Proceedings of the 10th IASTED-PDCN*. ACTA Press, 2011, pp. 91–98.

[13] A. Pellegrini and F. Quaglia, "Time-sharing time warp via lightweight operating system support," in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM PADS '15. New York, NY, USA: ACM, 2015, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2769458.2769478

[14] I. Di Gennaro, A. Pellegrini, and F. Quaglia, "Os-based numa optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Washington, DC, USA: IEEE Computer Society, 2016.

[15] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia, "Analytical/ml mixed approach for concurrency regulation in software transactional memory," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 81–91.

[16] P. Di Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, "Markov chain-based adaptive scheduling in software transactional memory," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2016. Washington, DC, USA: IEEE Computer Society, 2016.

[17] D. Rughetti, P. Di Sanzo, and A. Pellegrini, "Adaptive transactional memories: Performance and energy consumption tradeoffs," in *Network Cloud Computing and Applications (NCCA), 2014 IEEE 3rd Symposium on*. IEEE Computer Society, Feb 2014, pp. 105–112.

[18] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, "Automatic tuning of the parallelism degree in hardware transactional memory," in *Euro-Par 2014 Parallel Processing*. Springer International Publishing, 2014, pp. 475–486.

[19] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628080

[20] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero, "Clock gate on abort: Towards energy-efficient hardware transactional memory," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–8.

[21] S. Rodriguez and B. Jacob, "Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)," in *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, ser. ISLPED '06. New York, NY, USA: ACM, 2006, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/1165573.1165581

[22] C. Ferri, S. Wood, T. Moreshet, R. Iris Bahar, and M. Herlihy, "Embedded-tm: Energy and complexity-effective hardware transactional memory for embedded multicore systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 10, pp. 1042–1052, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2010.02.003

[23] S. Issa, P. Romano, and M. Brorsson, "Green-cm: Energy efficient contention management for transactional memory," in *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ser. ICPP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 550–559. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2015.64