RUBIC: Online Parallelism Tuning for Co-located Transactional Memory Applications

Amin Mohtasham
amohtasham@gsd.inesc-id.pt

João Barreto
joao.barreto@tecnico.ulisboa.pt

INESC-ID Lisboa/ Instituto Superior Técnico, Universidade de Lisboa, Portugal

ABSTRACT

With the advent of Chip-Multiprocessors, Transactional Memory (TM) emerged as a powerful paradigm to simplify parallel programming. Unfortunately, as more cores become available in commodity systems, the scalability limits of a wide class of TM applications become more evident.

Hence, online parallelism tuning techniques were proposed to adapt the optimal number of threads of TM applications. However, state-of-the-art solutions are exclusively tailored to single-process systems with relatively static workloads, exhibiting pathological behaviors in scenarios where multiple multi-threaded TM processes contend for the shared hardware resources.

This paper proposes RUBIC, a novel parallelism tuning method for TM applications in both single and multi-process scenarios that overcomes the shortcomings of the preciously proposed solutions. RUBIC helps the co-running processes adapt their parallelism level so that they can efficiently space-share the hardware.

When compared to previous online parallelism tuning solutions, RUBIC achieves unprecedented system-wide fairness and efficiency, both in single- and multi-process scenarios. Our evaluation with different workloads and scenarios shows that, on average, RUBIC enhances the overall performance by 26% with respect to the best-performing state-of-the-art online parallelism tuning techniques in multi-process scenarios, while incurring negligible overhead in single-process cases. RUBIC also exhibits unique features in converging to a fair and efficient state.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

Keywords

concurrency control; feedback-driven systems; resource allocation; software transactional memory

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935770

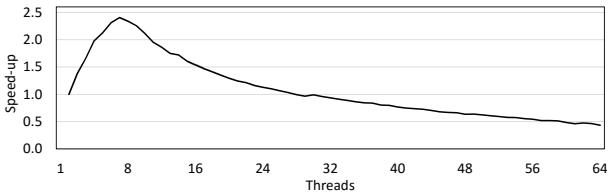


Figure 1: Intruder's throughput deteriorates after 7 parallel threads.

1. INTRODUCTION

Transactional Memory (TM) has emerged as a powerful paradigm to simplify parallel programming. Building on the abstraction of atomic transactions, and freeing the programmer from the complexity of conventional synchronization schemes, TM has proven capable of simplifying the development and verification of concurrent programs by avoiding the pitfalls of manual, lock-based synchronization, enhancing code reliability and boosting productivity. After intense research during the last decade, TM has reached the maturity required for mainstream adoption. Notable evidences of this trend include the software implementations in the gcc compiler [1], the upcoming version of the C++ standard [2], and best-effort hardware TM support in mainstream and high-end processors from major chip manufacturers.

Unfortunately, as more and more cores become available in commodity systems, the scalability limits of a wide class of TM applications become more evident. Taking STAMP [3] as a well-known benchmark suite, most applications reach a performance peak after a certain number of threads; to make things worse, many such applications even observe performance drops after such a peak [4]. For example, Figure 1 shows Intruder achieves the highest speed-up with only 7 parallel threads on a 4-socket 64-core machine, without using any explicit thread placement policy, and on average of 50 runs. As the number of threads goes beyond the peak point, the performance deteriorates such that, at 64 threads, the throughput becomes less than half of the sequential execution's throughput.

The limited scalability of the parallel workloads on modern multi/many-core processors implies 3 important observations: (i) running a parallel workload with as many threads as the number of hardware (h/w) contexts can lead to a significant waste of the hardware resources; (ii) there must be a mechanism that finds the optimal number of threads for a given workload and maximizes the speed-up;

(iii) to fully utilize the hardware, co-locating multiple TM parallel applications will become increasingly common [5, 6].

Recently, a number of works have proposed online techniques that aim at finding optimal parallelism level in *malleable* TM applications [7, 8, 4, 9]. Malleable applications are flexible applications and can set their parallelism level prior to or during their execution [10]. These constitute the vast majority of the applications that are found in the reference TM benchmarks [3, 11]. Online solutions usually employ online feedback-driven techniques together with hill-climbing techniques to learn about the workload and find its optimal parallelism level. In contrast to offline approaches, online solutions are particularly appealing because they are well-suited for workloads that are not known a priori or dynamic execution environments.

Nevertheless, the current state-of-the-art in online parallelism tuning techniques is designed for and evaluated in single-process scenarios. Although such techniques are implicitly *assumed* to also work well in multiple-process situations, our experimental results (presented in Section 4) show that they actually exhibit alarmingly poor performance when multiple TM processes run concurrently in the same machine. This pathological behavior is explained by some fundamental issues that arise when one considers a system of multiple parallel processes instead of a single one, as follows.

Oversubscription: a system becomes oversubscribed when its total number of computationally-intensive software (s/w) threads exceeds the number of h/w contexts. In an oversubscribed system, the overall performance degrades, mainly due to frequent context switches and increased cache trashing. Apart from this, a TM-application, specially, suffers from oversubscription due to prolonged transaction execution time and increased conflict likelihood, hence losing throughput[12]. In a single-process scenario, a parallelism tuning technique can easily avoid oversubscription by putting a hard limit on the number of s/w threads. However, in a multi-process environment, things are not that straightforward, as all the co-running processes need to somehow coordinate to keep the total number of s/w threads below the oversubscription point.

System-wide optimization: for a single process, the obvious goal is to simply maximize some performance metric (e.g., throughput, speed-up, etc.). However, in a multi-process system, the main goal is to optimize the whole system's performance. With such a system, the overall performance is usually defined as an aggregate function of each process' performance. Considering the limited hardware resources (i.e. h/w contexts), to maximize a given overall performance function, some processes may be required to give up some of their resources to the other processes (e.g. to those that achieve higher speed-up). *Sacrificing* a single for the sake of a whole is not aligned with the goals of the current parallelism tuning methods, where each process greedily tries to maximize its performance, regardless of the system's overall performance (Section 2 elaborates on this behavior).

Fairness: when it comes to resource allocation, establishing fairness immediately becomes a major issue. While, intuitively, in a fair system, all processes must receive *enough* resources to make some progress, defining and quantifying fairness is far from trivial. There exist several definitions for fairness and choosing each one affects the system's performance differently. For instance, one can define fairness as *equality*, while another actually finds equality unfair and allocates more resources to processes that contribute more to the system's overall performance (a.k.a. proportional fairness). In the context of resource allocation, for a given definition of fairness, the system's performance function is defined in such a way that it reflects the fairness requirements of that system as well [13, 14]. Nonetheless, as the state-of-the-art parallelism tuning methods do not consider maximizing any system-wide performance function, the fairness characteristics of such methods, when applied in multi-process environments, is totally unknown.

This paper proposes RUBIC, a highly adaptive online parallelism tuning method for malleable TM applications that overcomes all of the aforementioned shortcomings of the current tuning techniques in single and multi-process environments. We believe this is a key breakthrough, as it enables elastic systems composed of multiple TM-based applications that adaptively space-share h/w contexts amongst themselves, efficiently and fairly, at no extra cost with regard to the state-of-the-art techniques.

Inspired by the well-studied problem of flow/congestion control in data networks, RUBIC employs a sophisticated online feedback loop control within each running process that makes unilateral decisions only based on that process' local observations. That is to say that our solution does not rely on a central entity, nor does it require any sort of communication between running processes.

Our experimental results show that RUBIC can substantially boost the system's overall performance when compared to solutions previously proposed to single-process settings. At the same time, RUBIC is highly responsive to changes in the system and its convergence to an efficient and fair allocation is impressively fast, unlike the state-of-the-art methods.

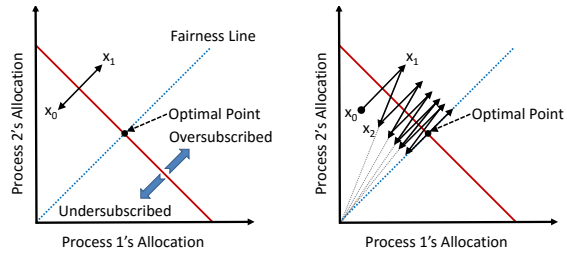
Although this paper focuses on TM workloads, our solution is not limited to TM applications. In fact, any malleable parallel application can benefit from RUBIC.

The rest of this paper is structured as follows. Section 2 elaborates on the shortcomings of the state-of-the-art methods and illustrates the rationale behind RUBIC. Section 3 describes RUBIC in detail. Section 4 shows how our proposed method acts in practice and how it outperforms the state-of-the-art. Finally, Section 5 surveys related work and Section 6 concludes.

2. THE NEED FOR SPEED AND FAIRNESS

In this section, we explain why the existing solutions, originally designed for single-process environments, are not suitable for multi-process systems. Then, we describe the rationale behind RUBIC to understand what makes RUBIC an adequate solution for parallelism tuning in multi-process environments.

Online parallelism tuning techniques usually employ a feedback loop to control the parallelism. A controller loop constantly monitors a performance metric (e.g. throughput) and makes the appropriate decision regarding the new parallelism level (i.e. number of s/w threads), based on the received feedback from the last decision. A decision can be either increasing or decreasing the parallelism level. Depending on the decision, the new level L_N is obtained through the functions f_{INC} and f_{DEC}, which are used for increasing and decreasing the parallelism level, respectively. Both functions accept the current level, L_C, as the input parameter.



(a) AIAD (b) AIMD

Figure 2: The convergence behavior of a system with two malleable processes, using different tuning schemes. AIMD converges to the optimal point.

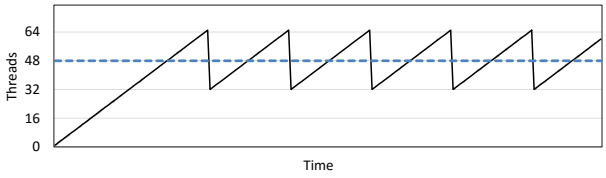


Figure 3: The expected behavior of an AIMD-based model ($\alpha = 0.5$) on a 64-core machine (the dashed line represents the average thread count).

The choice of functions f_{INC} and f_{DEC} defines the behavior of a parallelism tuning technique.

Existing solutions rely on simple linear functions for both increasing and decreasing purposes [4, 7]. At each step, the parallelism level is *additively* increased or decreased by a constant value Δl, which is usually set to 1. Therefore, we refer to this model as *AIAD* (additive-increase/additive-decrease).

AIAD gives the controller excellent capabilities to gradually explore the solution space and to find the optimal parallelism level in a single-process environment. However, in the following, we question the capabilities of AIAD-based schemes in multi-process environments, and show why such models do not converge to a system-wide fair and efficient state.

2.1 Additive vs. Multiplicative Decrease

Figure 2a shows the behavior of AIAD in a two-process system. Starting from an arbitrary point X_0, which is in the undersubscribed region, both processes additively increase their parallelism level. Therefore, the system's state moves along, at an angle of 45°, to point X_1, which is above the oversubscription line, and the system becomes oversubscribed. As a result, both processes shall notice some performance loss and additively reduce their parallelism level. Consequently, the system's state moves back, along at an angle of 45°, to the starting point, X_0, and the same additive increase phase happens. Therefore, the system's state oscillates between points X_0 and X_1 and never converges to the optimal point, at which the resources would be fairly allocated to both processes.

Inspired by congestion control techniques in data networks [15], we can solve the convergence issue by replacing the additive decrease phase with a *multiplicative decrease* (MD) one. In other words, instead of decreasing the parallelism level by one, the new level is obtained by multiplying the current level by a positive constant factor, α, where $0 < \alpha < 1$. We refer to this model as *AIMD*.

Figure 2b shows how switching from AIAD to AIMD affects the convergence behavior of the system. Starting from the same point, X_0, similarly to AIAD, the system's state moves to point X_1. However, as the system becomes oversubscribed, both processes multiplicatively reduce their parallelism level and the system's state moves toward the line joining X_1 and the origin to point X_2, which is below the oversubscription line. This cycle repeats and, eventually, the system's state oscillates around the optimal point, which is the goal.

Therefore, in order to have multiple processes converge to a fair and efficient state, the linear decrease function that is used by the state-of-the-art solutions should be replaced by a multiplicative decrease one.

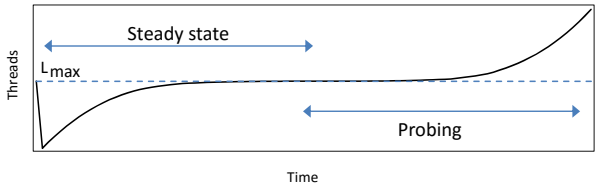


Figure 4: A cubic growth function includes a steady state phase (below L_{max}) and a probing phase (above L_{max}).

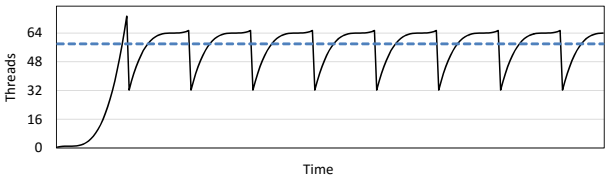


Figure 5: The expected behavior of a CIMD-based model ($\alpha = 0.5, \beta = 0.1$) on a 64-core machine. CIMD has a fast convergence due to its initial probing phase and achieves higher average utilization than AIMD (the dashed line represents the average thread count).

2.2 Additive vs. Cubic Increase

Multiplicative decrease in parallelism level solves the convergence issue. Nevertheless, it introduces a new problem to the system: the oscillations around the optimal point are expensive and cause overall undersubscription.

Figure 3 shows the behavior of a single highly-scalable process, using an AIMD controller ($\alpha = 0.5$), running on a 64-core machine. Each time the parallelism level exceeds 64, the controller detects some performance drop and multiplicatively decreases the parallelism level. Therefore, the new parallelism level is set to 32 (or close to it). At this point, the system is half-loaded and the controller, again, incrementally increases the parallelism level until it goes beyond 64 and the system oscillates around the optimal point (i.e. 64).

In this example, the average parallelism level is 48, depicted by the dashed-line in Figure 3, which means 25% of the h/w cores (16 out of 64) are left unused. In other words, the system is undersubscribed rather than operating at full capacity, due to using a multiplicative decrease function.

To enhance the system utilization, we can resort to a new growth function: a cubic function that was initially used in the context of congestion/flow control techniques in data networks [16]. More specifically, the growth function is obtained by the following equation:

$$L_{cubic} = L_{max} + \beta\big(\Delta t_{max} - \sqrt[3]{L_{max} \times \alpha/\beta}\big)^3 \qquad (1)$$

where L_{max} depicts the last parallelism level at which a performance loss was observed and Δt_{max} denotes the time since that performance loss; α is a constant multiplication factor that reduces the parallelism level to αL_{max} (as in AIMD); and, finally β is a constant scaling factor that controls the growth rate of the parallelism level.

Figure 4 shows the growth behavior of a cubic function after an MD phase at L_{max}, based on Equation (1). The parallelism level increases very fast upon an MD phase. Then, its growth slows down as it gets closer to L_{max}; becoming almost zero when the parallelism level reaches L_{max}. This phase is the *steady state* phase. During this phase, the controller tries to stabilize the parallelism level and to keep it close to L_{max} (i.e. near the oversubscription point).

As the time passes and the parallelism level exceeds L_{max}, the controller increases the growth rate and starts the *probing* phase. The probing phase indicates the possibility of free resources in the system. During this phase, the controller looks for a new oversubscription point, by taking longer and longer steps.

Figure 5 shows the expected behavior of a single and highly-scalable process, using a controller with *cubic-increase-/multiplicative-decrease* (CIMD) functions, on a 64-core machine. At the beginning, L_{max} is set to 1. As the parallelism level gets away from 1, the probing phase starts until the first performance loss, due to oversubscription, is observed around 64 threads. At this point, L_{max} is set to the observed oversubscription level, an MD phase occurs, and the steady state phase keeps the parallelism level close to L_{max} (e.g., 64). In this example, the average parallelism level is close to 60, which implies the system utilization is enhanced from 75% with AIMD to 94% with cubic growth.

Based on what we discussed in this section, we claim that to enable the fair and efficient parallelism tuning in both single- and multi-process environments, the growth and reduction functions used in the state-of-the-art solutions must be revised. More specifically, we propose to replace the AIAD scheme used in the current solutions with a CIMD scheme.

3. PARALLELISM TUNING WITH RUBIC

In this section, we present RUBIC: the first parallelism tuning solution that is tailored for both single- and multi-process environments.

We assume each process includes a pool of s/w worker threads and runs a malleable parallel workload. The process' parallelism level can be tuned by activating or blocking the threads inside the thread-pool. As soon as a s/w thread completes its current task, it picks a new task from a task queue, until all tasks have been completed. The worker thread can then terminate, or block until there are new tasks available. In order to access shared data, tasks use transactions that ensure safe synchronization of concurrent accesses to the data.

3.1 The RUBIC Algorithm

Each process runs a monitoring thread that is responsible for running the RUBIC controller. This thread is created with a higher priority than other threads inside the thread-pool. This ensures that the monitoring thread gets to perform its duty even when the system is oversubscribed.

The controller periodically measures the process' throughput (e.g., commit-rate) and updates the T_c variable, in a loop. At the end of each iteration, the value of T_c is stored in T_p, to be used in the next round (i.e., iteration).

We assume the time between two measurements is long enough so an active worker thread can finish at least one task within that time. To measure the throughput, each worker thread maintains a local counter. Upon finishing a task, a worker thread increases its counter by one. At each round, the monitoring thread reads all the thread-local counters from the thread-pool and calculates T_c. This determines the process throughput in the period that is just completing.

It is worth noting that, for each thread-local counter variable, only that thread can update such counter; the monitoring thread only reads from those counters. Therefore, maintaining the thread-specific throughput counters is very lightweight; no atomic instructions are necessary.

After each measurement, the controller compares T_c with T_p. Should T_c be greater than or equal to T_p, the controller increases the parallelism level according to the cubic growth function in Equation 1. Conversely, in the case of a performance loss, a multiplicative reduction function calculates the new level.

Algorithms 1 and 2 show how the monitoring thread and the other threads inside the thread-pool co-operate in order to change the parallelism level.

To change a process' parallelism level, we augment each worker thread's metadata with a semaphore and a unique integer identifier $tid \in [0..S-1]$, where S is the size of that process' thread-pool. The number of active threads is maintained in a process-wide global variable, named L_{RUBIC}. The monitoring thread adjusts the parallelism level by changing the value of L_{RUBIC}.

At the application initialization, the parallelism level is set to minimum (1 thread). This means that only the first worker thread (i.e., $tid = 0$) is allowed to run tasks at the beginning and the rest must be blocked. Also, all thread semaphores are initialized to 0 (lines 2-5 in Algorithm 1).

Before acquiring a task from the task queue, each thread compares its tid with the global variable L_{RUBIC}. If tid is greater than L_{RUBIC}, it means the thread must be blocked. Hence, the thread waits on its semaphore, which was initialized to 0, and blocks (lines 8-10 in Algorithm 1). Otherwise, the thread acquires a task, if any, and executes.

It is noteworthy that the normal task acquisition flow does not include any system calls and the $Wait$ call only happens when a thread must block.

3.2 The Increase Function

The lines 6-15 in Algorithm 2 show how RUBIC increases the parallelism level, based on Equation 1 in Section 2.2.

In the round immediately after a cubic growth, the controller runs an additive increase phase and increments the parallelism level just by one. This enables the controller to compare two adjacent levels and to make more accurate decisions. The variable *growth* is used to interleave cubic and linear growth phases, in Algorithm 2.

Algorithm 1 Applied changes to the legacy instrumentation

```
1: procedure Initialization
2:     L_RUBIC = 1
3:     for each thread do
4:         thread.semaphore = 0
5:     end for
6: end procedure

7: procedure AcquireTask(thread)
8:     if (thread.tid ≥ L_RUBIC) then
9:         Wait(thread.semaphore)
10:    end if
11:    //The rest remains unchanged...
12: end procedure
```

Finally, the controller signals the blocked worker threads that must be awakened in this round. It also stores the value of T_c in T_p to be used in the next round (lines 20-23 in Algorithm 2).

3.3 The Reduction Function

Should the throughput drop, RUBIC decreases the parallelism level (lines 25-35 in Algorithm 2).

The reduction function also interleaves two phases: a multiplicative phase and a linear one. This hybrid technique is used to avoid unnecessary multiplicative phases as much as possible. The main rationale behind this is as follows. A performance drop can be due to: (i) passing the workload's optimal parallelism level in the growth phase; or (ii) dynamic changes in the running workload or available hardware resources. This happens when, for example, a new process joins the system.

Whenever there is a performance drop, RUBIC first assumes the slowdown is caused by passing the workload's optimal parallelism level. Therefore, it linearly decreases the parallelism (lines 31-32), instead of a multiplicative decrease.

If a linear decrease does not produce the desired outcome, and the performance loss still persists in the next round, a multiplicative decrease happens (lines 26 to 29).

The variable *reduction* enables the interleaving between the two phases above.

4. EVALUATION

In this section we evaluate RUBIC. The purpose of this evaluation is to thoroughly analyze the behavior of our technique in static/dynamic and single-/multi-process environments. The main examined features are: performance, efficiency, fairness and adaptability. Throughout our experiments, we show how RUBIC outperforms the existing solutions in all the above features.

4.1 The System's Performance Function

As stated earlier in Section 1, in a multi-process environment, one must consider two main factors for optimal resource allocation: (i) the system's overall performance, rather than each process' performance alone; and (ii) fairness to ensure all processes get some fair share of the system and no process starves.

Therefore, we must define the system's overall performance function so that it reflects both aforementioned features. To do so, we resort to Nash's solution for the bargaining problem (NSBP) [13].

Algorithm 2 Adaptive parallelism algorithm in the monitoring thread

```
1: growth ← CUBIC, reduction ← LINEAR, T_p ← 0
2: while running do
3:     Sleep(TIME_PERIOD)
4:     T_c ← CalculateCurrentThroughput()
5:     if (T_c ≥ T_p) then
6:         L_p ← L_RUBIC
7:         if (growth = CUBIC) then
8:             Δt_max ← Δt_max + 1
9:             L_cubic ←
10:                L_max + β(Δt_max − ∛(L_max × α/β))^3
11:            L_RUBIC ← max(L_cubic, L_RUBIC + 1)
12:            growth ← LINEAR
13:        else
14:            L_RUBIC ← L_RUBIC + 1
15:            growth ← CUBIC
16:        end if
17:        if (T_p ≠ 0) then
18:            reduction ← LINEAR
19:        end if
20:        for each thread where L_p ≤ tid < L_RUBIC do
21:            Signal(thread.semaphore)
22:        end for
23:        T_p ← T_c
24:    else
25:        Δt_max ← 0
26:        if (reduction = MULTIPLICATIVE) then
27:            L_max ← L_RUBIC
28:            L_RUBIC ← αL_RUBIC
29:            reduction ← LINEAR
30:        else
31:            L_RUBIC ← L_RUBIC − 2
32:            reduction ← MULTIPLICATIVE
33:        end if
34:        growth ← LINEAR
35:        T_p ← 0
36:    end if
37: end while
```

For a process ρ, running a parallel workload ω, we define the speed-up function of the process ρ ($S_\rho(\omega)$) as the ratio between the process's obtained throughput ($T_\rho(\omega)$), and the measured throughput of a sequential execution of the workload ω, in a single process configuration ($T_{seq}(\omega)$). In other words, $S_\rho(\omega) = \frac{T_\rho(\omega)}{T_{seq}(\omega)}$.

Based on NSBP, in a system with N running processes, where each one is running workload ω_ρ, the system's overall performance is defined as the product of all process' speed-up (i.e. $\prod_{\rho=1}^{N} S_\rho(\omega_\rho)$).

For example, it can be easily shown that in a contended system running identical processes, equally sharing the hardware between the processes maximizes the system's overall performance.

4.2 The Efficiency Function

We also define a process's efficiency (E) as the ratio between its achieved speed-up (S) and its parallelism level (L) (i.e. the number of active threads)[17]. Thus, for a process ρ, running a workload ω: $E_\rho(\omega) = \frac{S_\rho(\omega)}{L_\rho(\omega)}$.

Similarly to the system's performance function, we define the systems's total efficiency as the product of all running process, which is expressed as: $\prod_{\rho=1}^{N} E_\rho(\omega_\rho)$

4.3 Allocation Policies

We compare our results to those gathered by different policies referred as *Greedy*, *EqualShare*, *F2C2* [4], and *EBS* [7]. EBS and F2C2 are almost identical and they both implement an AIAD-based controller. The only difference is that,

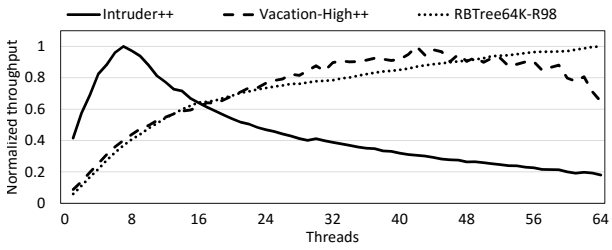


Figure 6: The scalability graph of the evaluated workloads.

F2C2 benefits from an initial exponential growth phase for faster convergence to the optimal level. By this mechanism, the controller initially doubles the parallelism level instead of increasing it by 1. After the first performance loss, F2C2 halves the parallelism level and switches to pure AIAD until the end, as in EBS.

With Greedy, each process tries to take over all the hardware by spawning as many parallel threads as the h/w contexts.

In EqualShare, each process takes an equal share of the h/w context, regardless of its workload. This policy relies on a central entity to decide upon the number of threads in each process. It is the simplest heuristic we could use to avoid oversubscription. Nevertheless, EqualShare is naïve, since it ignores the running workloads and provides each process with the same amount of h/w contexts, even if some processes actually do not require those many contexts.

Finally, in our implementation of RUBIC, the α and β constants, in Equation (1) from Section 2, are set to 0.8 and 0.1, respectively, to obtain the best results.

4.4 Methodology and Workloads

We evaluated the policies in a 4-socket machine consisting of AMD Opteron 6272 CPUs, each with 16 cores, where each core runs one thread, resulting in 64 h/w contexts and total memory of 32GB, running Linux kernel 3.2.0-58.

The monitoring thread measures the commit-rate every 10 milliseconds.

We have implemented all the policies and incorporated them into the RSTM transactional framework [18], using SwissTM [19] as the underlying TM runtime. We consider three different benchmarks: Vacation and Intruder from the STAMP benchmark suite [3] and the Red-Black-Tree micro benchmark (with 64K elements and 98% look-up operations). These well-known benchmarks are chosen to represent applications from a wide scalability spectrum, ranging from poorly to highly scalable workloads.

Furthermore, for the sake of simplicity, the thread placement is left to the OS's default policy.

Figure 6 shows the scalability graph of each benchmark. In this graph, we measure the throughput in terms of transaction commit-rate as the number of parallel threads grows from 1 to 64. To make the results from each benchmark comparable, the values are normalized relatively to the peak throughput that was measured in each workload.

It should be noted that the choice of the host machine, underlying parallelism runtime and the benchmark does not affect the conclusions we draw next. This observation stems from the fact that our techniques only depend on the scalability curve defined by each running process. The only requirement is that the scalability graph of the workloads must monotonically increase until its peak point.

The metric we use to calculate throughput is commit-rate, measured by the number of commits per second for each process. We use the throughput values to calculate the performance and the efficiency metrics that are previously defined in sections 4.1 and 4.2.

Each experiment lasts for 10 seconds and performance results are the average of 50 repeated experiments to minimize the evaluation noise.

We run the experiments in both pairwise and single-process settings. In the pairwise execution, two co-running processes run distinct workloads, with the same starting time. This results into three pairs of workloads. In a single process setting, a single process runs either of the workloads.

4.5 Performance Results

We now present the obtained performance results of all allocation policies.

4.5.1 Pairwise Execution

Figure 7 depicts the system's overall metrics for the pairwise execution of the workloads.

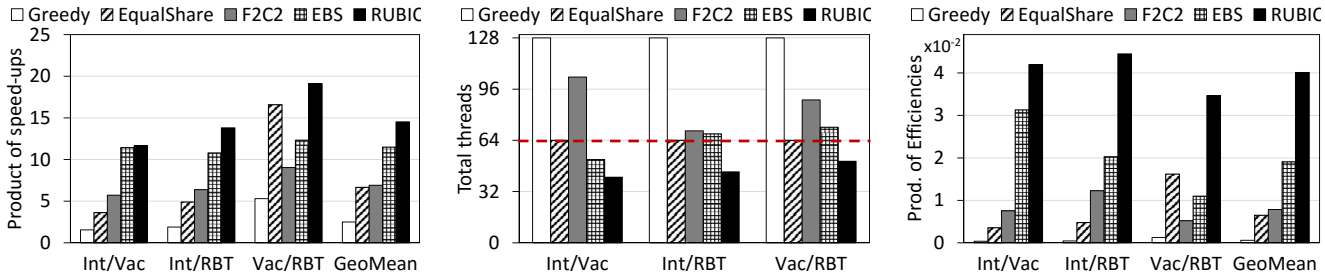
Figure 7a shows the system's total speed-up with each workload pair and the geometric average of all three pairwise experiments. It is evident that for all workload pairs and also on average, RUBIC achieves the best results, while Greedy proves to be the worst policy. The poor performance of Greedy is due to ignoring the running workloads and oversubscribing the system, as expected.

EqualShare performs poorly too, as it also neglects the running workloads. This is especially evident in the workload pairs that include *Intruder*, where the 32 h/w contexts allocated to this workload cause severe performance loss for the running process and the system's overall performance, consequently. EqualShare performs better with the *Vac/RBT* pair. This is because both running workloads scale up to 32 threads. Hence, no workload suffers from performance loss, due to over-allocation. Nonetheless, even with this workload pair, EqualShare is not the best policy, as it still suffers from ignoring the running workloads and improper resource allocation.

Surprisingly, unlike what we initially expected, EBS and F2C2 perform very differently. With all workload pairs, EBS outperforms F2C2. Furthermore, with the *Int/Vac* pair, EBS's performance is comparable to RUBIC's. This is because, with this pair, running both workloads at their peak parallelism level does not oversubscribe the system. Therefore, the greedy characteristic of EBS neither oversubscribes the system nor leads to unfair resource allocation. However, this situation is not the case for the other pairs, where RUBIC outperforms EBS because of EBS's unfair allocation and oversubscription.

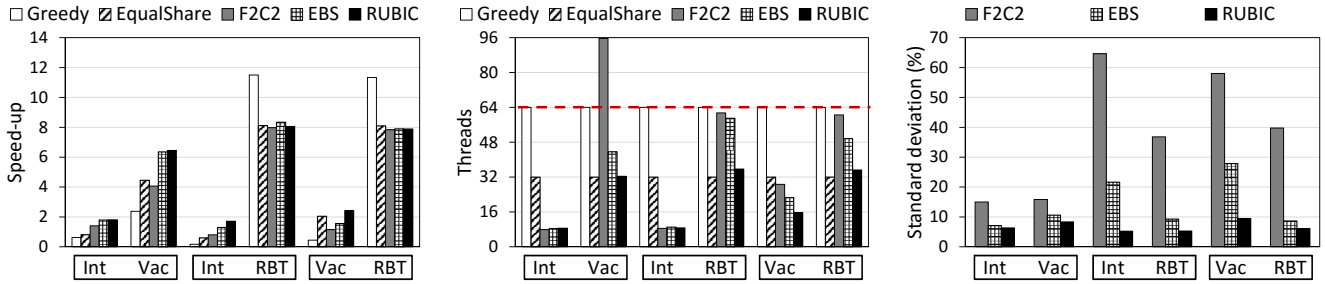
On average over all pairwise experiments, RUBIC enhances the system's overall performance by 26% and 500% with regard to the second-best policy (EBS) and the worst policy (Greedy), respectively.

The total numbers of running s/w threads in the system are depicted in Figure 7b. We expected all adaptive policies to keep the total number of threads below the oversubscription line (the dashed line). Perhaps surprisingly, only RUBIC meets our expectation. In fact, except for the *Int/Vac* pair, where EBS maintains the total number of threads be-



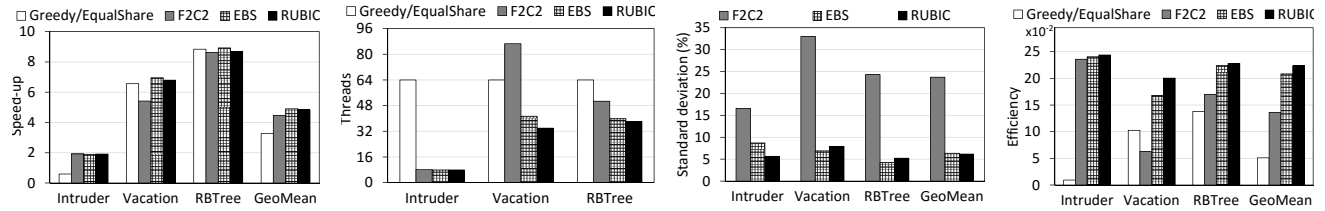
(a) Total performance (b) Total number of threads (c) Total efficiency

Figure 7: The system's overall metrics in pairwise execution of the workloads.



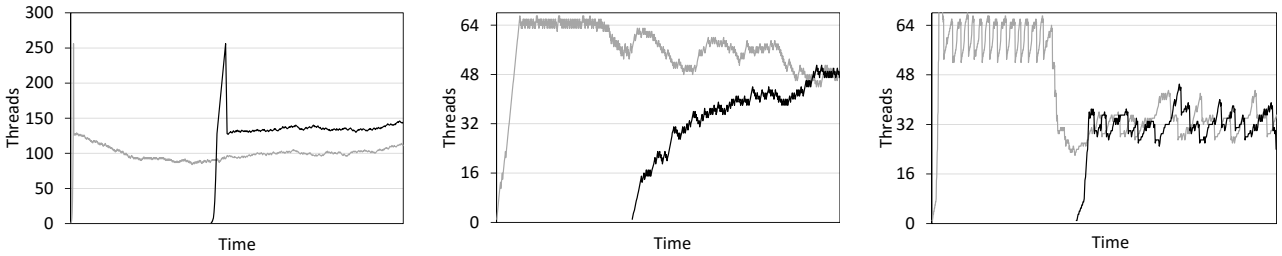
(a) Per-process speed-ups (b) Per-process threads (c) Per-process threads std. deviation

Figure 8: The system's per-process metrics in pairwise execution of the workloads (in graph (c) lower is better).



(a) Speed-up (b) No. of threads (c) Threads std. deviation (d) Process efficiency

Figure 9: Process-specific metrics for single-process execution of the workloads (in graph (c) lower is better).



(a) F2C2 (b) EBS (c) RUBIC

Figure 10: Number of active threads over time, in 2 homogeneous processes with different arrival time. With F2C2 and EBS techniques, the processes are greedy and oversubscribe the system to get more computational power. But, with RUBIC, the processes reach a fair equilibrium without oversubscribing the hardware.

low 64, EBS and F2C2 cannot keep the system's overall state below the oversubscription line. This is caused by the greedy nature of these policies, where co-running processes with scalable workloads get involved in a race over resources and oversubscribe the system.

Due to higher performance and lower resource consumption, RUBIC is theoretically expected to be the most efficient policy as well. The total efficiency results in Figure 7c confirm that expectation. As in the performance results, EBS comes after RUBIC as the second-most efficient policy, while Greedy is the least efficient one.

On average over all pairwise experiments, RUBIC proves to be 2 and 66 times more efficient than the second-most efficient policy (EBS) and the least efficient policy (Greedy), respectively.

Figure 8 demonstrates the per-process statistics of the pairwise execution of the workloads. This information helps us better understand the superior overall performance of RUBIC. In the graphs, each workload pair is specified by a rectangle and each 2-bar group represents a whole pairwise experiment.

Interestingly, in Figure 8a (and unlike Figure 7a), Greedy does not always lead to the worst speed-up. In fact, the *RBT* workload always achieves the highest speed-up with Greedy, when compared to other policies. However, taking a look at *RBT*'s counterparts in the pairwise execution, Greedy achieves very low speed-up for both *Intruder* and *Vacation* workloads. This justifies Greedy's very low overall performance.

Comparing RUBIC's per-process speed-up statistics, in Figure 8a, to EBS's, they both have comparable speed-up gains running the *Int/Vac* pair. Nevertheless, *RBT*'s counterparts always gain higher speed-up with RUBIC. This is due to the fairness characteristic of RUBIC. In other words, RUBIC slightly sacrifices a very scalable process' performance to achieve a much higher speed-up gain on a less scalable process (e.g., 1% of *RBT*'s speed-up in exchange for 10% improvement in *Intruder*). Figure 8c shows how RUBIC allocates less threads to *RBT* to reduce the pressure on its *Intruder* and *Vacation* counterparts. Consequently, both *Intruder* and *Vacation* experience a performance boost, which highly affects the system's overall performance.

This interesting feature of RUBIC is also known as *proportional fairness*, which is a well-studied topic in the context of resource allocation in distributed systems [14].

Furthermore, Figure 8b proves RUBIC to be the most stable adaptive policy. In this figure, the lowest standard deviation in resource allocation, across all 50 repetitions of each experiment, belongs to RUBIC. On the contrary, F2C2 is the most unstable adaptive solution.

It is worth nothing that the unexpected poor performance and low stability of F2C2 stem from the initial exponential growth phase. By this initial phase, a process can fall onto a performance plateau, where the AIAD-based technique cannot exit from, preventing the controller from ever converging. This is visible, for example, when running the workload pair *Int/Vac*, since *Vacation*'s parallelism level goes beyond the number of h/w contexts and never falls back (Figure 8c).

4.5.2 Single-process Execution

Although RUBIC originally targets multi-process environments, it is also relevant to evaluate it in single-process scenarios.

It should be noted that in single-process scenarios, EqualShare and Greedy become identical, as they both give all the hardware to a single running process.

Figure 9a depicts the achieved speed-up of all the policies, running either of the workloads. For all the workloads and also on average, RUBIC's speed-up is always comparable with the best performing policy.

Figure 9b shows that RUBIC achieves the aforementioned speed-up by allocating slightly less threads to the running process. However, RUBIC's thread allocation is always the closest one to the best performing policy (i.e. EBS).

Furthermore, on average, RUBIC is the most stable policy, by having the lowest standard deviation in its allocation results throughout the repeated experiments, depicted in Figure 9c. Nevertheless, the same figure suggests that EBS's stability is very comparable to RUBIC.

4.6 Convergence

This section analyzes the convergence behavior of RUBIC and the other parallelism tuning solutions.

To do so, we conduct the following experiment: two processes, P1 and P2, run an identical workload. However, these process have different arrival times. P2 arrives 5 seconds after P1 starts running and the whole experiment lasts for 10 seconds.

We record each process' parallelism level during the experiment. Then we see how P1 behaves before P2's arrival and how both processes behave as P2 joins the system.

To make the running processes actually fight over resources, we chose to use a conflict-free red-back-tree workload (i.e. 100% read-only transactions). This workload is highly scalable and scales up to the number of h/w contexts, in a single-process scenario. However, with 2 running processes a fair and efficient allocation state is when the processes equally share the hardware and each uses 32 s/w threads.

Figure 10 shows the behavior of different online policies in the aforementioned experiment. We see that each policy behaves very differently from the others.

In Figure 10a, F2C2 exhibits a pathological behavior: neither before, nor after P2's arrival, the system converges to an efficient state. The initial growth phase exponentially increases the parallelism level to beyond the number of h/w contexts. Then, the F2C2 controller gets stuck on a performance plateau, and very slowly decreases the parallelism level, but never converges to 64 threads. As P2 arrives, P2 falls into the same plateau. Interestingly, upon P2's arrival, P1 changes its behavior and starts slowly increasing its parallelism level, as well as P2. In fact, after P2's arrival, both processes fall into a race and greedily increase their resource usage.

In Figure 10b, EBS performs the best before P2's arrival (unlike F2C2). P1's parallelism level gradually converges to 64 threads and remains steady at that level. But, as P2 joins the system, both processes behave rather randomly and they do not converge to the optimal allocation, which is 32 threads per process.

Figure 10c depicts RUBIC's impressive convergence behavior. In the beginning, P1 runs the initial probing phase and quickly converges to 64 threads. From this point on, P2 oscillates around the optimal level (e.g., 64) until P2 joins the system. P2's cubic probing phase coincides with several multiplicative decreases from P1's side. Therefore, both processes, almost immediately, get close to 32 threads and both

start oscillating around the optimal allocation line (i.e. 32 threads), until the end of the experiment, unlike the other tuning policies.

5. RELATED WORK

While not a new topic, the problem of adaptive parallelism tuning has received stronger attention in recent years due to the emergence of multi/many-core architectures.

Didona *et. al.* [7] propose an exploration-based scaling (EBS) method to dynamically find the best parallelism level for TM applications. They use transaction commit-rate as the performance measure and try to maximize the throughput by dynamically changing the number of active threads, taking a hill-climbing approach. Recently, Ravichandran *et. al.* [4] introduced F2C2-STM, another adaptive parallelism method. F2C2-STM is similar to Didona's work, except for an initial exponential-growth phase that allows a faster convergence. We compared our results of RUBIC to those of EBS and F2C2-STM in Section 4.

Heiss *et. al.* [20] also take a similar hill-climbing approach to find the optimal parallelism level in transactional database systems.

Ansari *et. al.* [8] and Chan *et. al.* [9] suggest techniques to increase the efficiency of TM applications by limiting the wasted computational power caused by aborted transactions. These methods try to keep the ratio of committed transactions above a predefined threshold by dynamically changing the number of active threads. However, such techniques are not targeted at maximizing the system throughput neither in single-process nor in multi-process environments, as they only limit the amount of wasted work.

Mohtasham *et. al.* [21] propose an online parallelism tuning technique that is based on additive increase and multiplicative decrease phases. However, as discussed in Section 2, this method is unstable and underutilizes the system.

Pusukuri *et. al.* [22] introduce a kernel-level solution to find the near-optimal number of threads for a single parallel application. This method is based on profiling the application offline to find its best parallelism point. However, as it is offline, it is not able to cope with dynamic changes in workload or available hardware resources.

Also, there exists a body of research trying to improve the performance of TM by introducing contention management or transactional scheduling elements to a transactional runtime [23, 24, 25]. Although these approaches may share the concept of scheduling with our work in a broad sense, they focus on rescheduling transactions (threads) inside a single process, while we focus on the interferences between multiple processes. In fact such research works are orthogonal and complementary to ours.

Mohtasham *et. al.* [26] formalize and solve the resource allocation problem for co-existing transactional memory applications, by introducing FRAME. FRAME is a centralized and offline method; hence to obtain the optimal allocation, it requires thorough knowledge of the co-running workloads, which is far from trivial. Similarly, Creech *et. al.* [17] solves the resource allocation problem for co-running OpenMP processes with SCAF.

Rughetti *et. al.* [27, 28] use machine learning techniques to predict the best parallelism level in software and hardware transaction memory. These techniques need a learning phase and are not completely online. Also, due to their dependence on the learning phase, such techniques may only perform well in single-process scenarios.

There is significant work on adaptive parallelism tuning for data-parallel languages, which shares some goals with our proposal [29]. However, there are crucial differences between such context and the one that this paper considers. Most importantly, contention across tasks within a job in data-parallel programs is rare or inexistent; whereas TM applications inherently share data, thus their threads are highly prone to interferences such as data conflicts. Moreover, where TM applications typically run a pool of active threads that is mostly stable, data-parallel programs dynamically unfold a directed acyclic graph of tasks, hence can quickly oscillate between different parallelism levels. Therefore, the assumptions underlying adaptive scheduling techniques for data-parallel languages are not directly applicable to the TM programs.

Furthermore, in the context of task scheduling and aside from the classic schedulers [30], there exists a plethora of contention-aware scheduling techniques [31, 32, 5]. Such methods try to come up with an efficient solution to map software threads to hardware cores and minimize the cross-thread interferences. These methods are orthogonal to the parallelism tuning problem. That is to say, contention-aware schedulers do not change the number of threads in the parallel processes and, given a fixed set of active threads, these methods merely aim at finding the best thread-to-core mapping. Therefore, our solution can complement such schedulers with the ability to limit the total number of active software threads in the system.

6. CONCLUSION AND FUTURE WORK

Transactional memory has proved itself as an elegant and easy-to-use alternative to traditional lock-based methods. Although TM solutions are known to be scalable on multi-core processors, due to the inherent workload contention, many TM applications cannot scale up to the ever-increasing parallelism available on commodity many-core machines. In fact, most TM workloads not only stop scaling after peaking at a certain parallelism level, but also start losing performance after that peak.

Hence, it becomes increasingly important to devise effective approaches to dynamically tune each process' parallelism in order to reach the maximum throughput.

This problem has received recent attention. However, the state-of-the-art solutions are tailored to and evaluated considering single-process scenarios, thus neglecting the dynamics of the system, the running workload and fairness in multiprocess environments.

This paper proposes RUBIC: a decentralized method for adaptive parallelism tuning for co-located transactional multi-threaded processes. Our evaluation with different workloads and scenarios shows that RUBIC enhances the system's performance compared to the state-of-the-art parallelism tuning solutions in multi-process environments, while exhibiting considerably faster convergence and higher fairness.

Although this paper focuses on the specific case of programs based on the transactional memory paradigm, RUBIC is extensible to any type of malleable application. The key insight is that, as long as there are meaningful and precise ways of measuring the throughput of each process, it can serve as the input to RUBIC. In future work, we plan to extend RUBIC to support a wider range of the target

applications, including non-transactional and non-malleable parallel applications.

7. ACKNOWLEDGMENTS

This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through the project with the reference UID/CEC/50021/2013.

8. REFERENCES

[1] W. Ruan, T. Vyas, Y. Liu, and M. Spear, "Transactionalizing legacy code: An experience report using GCC and memcached," *ACM SIGARCH Computer Architecture News*, 2014.

[2] V. Luchangco, M. Wong, H. Boehm, *et al.*, "Transactional memory support for C++," 2014.

[3] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," IISWC'08, 2008.

[4] K. Ravichandran and S. Pande, "F2C2-STM: Flux-based feedback-driven concurrency control for STMs," IPDPS'14, IEEE, 2014.

[5] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," EuroSys'14, ACM, 2014.

[6] S. Peter, A. Schüpbach, P. Barham, *et al.*, "Design principles for end-to-end multicore schedulers," HotPar'10, USENIX Association, 2010.

[7] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," in *Networked Systems*, Springer Berlin Heidelberg, 2013.

[8] M. Ansari, M. Luján, C. Kotselidis, *et al.*, "Robust adaptation to available parallelism in transactional memory applications," in *Transactions on High-Performance Embedded Architectures and Compilers III*, Springer, 2011.

[9] K. Chan, K. T. Lam, and C.-L. Wang, "Adaptive thread scheduling techniques for improving scalability of software transactional memory," PDCN'11, ACTA Press., 2011.

[10] D. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, Springer Berlin Heidelberg, 1996.

[11] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A benchmark for software transactional memory," EuroSys'07, ACM, 2007.

[12] W. Maldonado, P. Marlier, P. Felber, *et al.*, "Scheduling support for transactional memory contention management," in *ACM Sigplan Notices*, ACM, 2010.

[13] J. F. Nash Jr, "The bargaining problem," *Econometrica: Journal of the Econometric Society*, 1950.

[14] F. P. Kelly, A. K. Maulloo, and D. K. Tan, "Rate control for communication networks: shadow prices, proportional fairness and stability," *Journal of the Operational Research society*, 1998.

[15] D. M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, 1989.

[16] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, 2008.

[17] T. Creech, A. Kotha, and R. Barua, "Efficient multiprogramming for multicores with SCAF," MICRO-46, ACM, 2013.

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," TRANSACT'06, 2006.

[19] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," PLDI'09, ACM, 2009.

[20] H. Heiss and R. Wagner, "Adaptive load control in transaction processing systems," VLDB'91, Morgan Kaufmann Publishers Inc., 1991.

[21] A. Mohtasham and J. Barreto, "Brief announcement: Fair adaptive parallelism for concurrent transactional memory applications," SPAA'15, ACM, 2015.

[22] K. Pusukuri, R. Gupta, and L. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," IISWC'11, 2011.

[23] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," PODC'05, ACM, 2005.

[24] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," PODC'05, ACM, 2005.

[25] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," SPAA'08, ACM, 2008.

[26] A. Mohtasham, R. Filipe, and J. Barreto, "FRAME: Fair resource allocation in multi-process environments," ICPADS'15, IEEE, 2015.

[27] D. Rughetti, P. Di Sanzo, A. Pellegrini, B. Ciciani, and F. Quaglia, "Tuning the level of concurrency in software transactional memory: An overview of recent analytical, machine learning and mixed approaches," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, Springer, 2015.

[28] D. Rughetti, P. Romano, F. Quaglia, and B. Ciciani, "Automatic tuning of the parallelism degree in hardware transactional memory," EuroPar'14, Springer, 2014.

[29] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," PPoPP'06, ACM, 2006.

[30] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Prentice Hall Press, 2014.

[31] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, 2012.

[32] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *EuroSys'10*, ACM, 2010.