Bounds On Contention Management Algorithms

Johannes Schneider[1], Roger Wattenhofer[1]

Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland

Abstract

We present two new algorithms for contention management in transactional memory, the deterministic algorithm *CommitRounds* and the randomized algorithm *RandomizedRounds*. Our randomized algorithm is efficient: in some notorious problem instances (e.g., dining philosophers) it is exponentially faster than prior work from a worst case perspective. Both algorithms are (i) local and (ii) starvation-free. Our algorithms are local because they do not use global synchronization data structures (e.g., a shared counter), hence they do not introduce additional resource conflicts which eventually might limit scalability. Our algorithms are starvation-free because each transaction is guaranteed to complete. Prior work sometimes features either (i) or (ii), but not both. To analyze our algorithms (from a worst case perspective) we introduce a new measure of complexity that depends on the number of actual conflicts only. In addition, we show that even a non-constant approximation of the length of an optimal (shortest) schedule of a set of transactions is NP-hard – even if all transactions are known in advance and do not alter their resource requirements. Furthermore, in case the needed resources of a transaction varies over time, such that for a transaction the number of conflicting transactions increases by a factor k , the competitive ratio of any contention manager is $\Omega(k)$ for $k < \sqrt{m}$, where m denotes the number of cores.

1 Introduction

Designing and implementing concurrent programs is one of the biggest challenges a programmer can face. Transactional memory promises to resolve a couple of the difficulties by ensuring correctness and fast progress of computation at the same time. Transactions have been in use for database systems for a long time. They share several similarities with transactional memory. For instance, in case of a *conflict* (i.e. one transaction demanding a resource held by another) a transaction might get aborted and all the work done so far is lost, i.e. the values of all accessed variables will be restored (to the ones prior to the execution of the transaction).

The difficulty lies in making the right decision when conflicts arise. This task is done by so-called contention managers. They operate in a distributed fashion, that is to say, a separate instance of a contention manager is available for every thread, operating independently. If a transaction A stumbles upon a

desired resource, held by another transaction B, it asks its contention manager for advice. We consider three choices for transaction A: (i) A might wait or help B, (ii) A might abort B or (iii) abort itself. An abort wastes all computation of a transaction and might happen right before its completion. A waiting transaction blocks all other transactions trying to access any resource owned by it.

Our contributions are as follows: First, we show that even coarsely approximating the makespan of a schedule is a difficult task. (Informally, the makespan is the total time it takes to complete a set of transactions.) This holds even in the absence of an adversary. However, in case an adversary is able to modify resource requirements such that the number of conflicting transactions increases by a factor of k, the length of the schedule increases by a factor proportional to k. Second, we propose a complexity measure allowing more precise statements about the complexity of a contention management algorithm. Existing bounds on the makespan, for example, do not guarantee to be better than a sequential execution. However, we argue that since the complexity measure only depends on the number of (shared) resources overall, it does not capture the (local) nature of the problem well enough. In practice, the total number of (shared) resources may be large, though each single transaction might conflict with only a few other transactions. In other words, a lot of transactions can run in parallel, whereas the current measure only guarantees that one transaction runs at a time until commit. Third, we point out weaknesses of widely used contention managers. For instance, some algorithms schedule certain sets of transactions badly, while others require all transactions – also those facing no conflicts – to modify a global counter or access a global clock. Thus the amount of parallelism declines more and more with a growing number of cores. Fourth, we state and analyze two algorithms. Both refrain from using globally shared data. From a worst-case perspective, the randomized algorithm *RandomizedRounds* improves on existing contention managers drastically (exponentially) if for each transaction the number of conflicting transactions is small. In an extended version of this paper [14] we also show that to achieve a short makespan (from a worst case perspective) it is necessary to detect and handle all conflicts early, i.e. for every conflict a contention manager must have the possibility to abort any of the conflicting transactions.

2 Related Work

Transactional memory was introduced in the nineties [8, 16]. In 2003 the FSTM system was proposed [6] and also the Dynamic STM (DSTM)[7] for dynamic data structures was described, which suggests the use of a contention manager as an independent module. After these milestones, a lot of systems have been proposed. An overview of design issues from a practical point of view can be found in [3].

Most proposed contention managers have been assessed by specific benchmarks only, and not analytically. A comparison of contention managers based on benchmarks can be found in [12, 10]. The experiments yield best performance for randomized algorithms, which all leave a (small) chance for arbitrary large completion time. Apart from that, the choice of the best contention manager

varies with the considered benchmark. Still, an algorithm called Polka [12] exhibits good overall performance for a variety of benchmarks and has been used successfully in various systems, e.g. [2, 10]. In [10] an algorithm called SizeMatters is introduced, which gives higher priority to the transaction that has modified more (shared) memory. In an enhanced version of this paper [14] we show that from a worst-case perspective Polka and SizeMatters may perform exponentially worse than *RandomizedRounds*. In [4] the effects of selfishness among programmers on the makespan is investigated for various contention managers from a game theoretic perspective.

The first analysis of a contention manager named Greedy was given in [5], using time stamps to decide in favor of older transactions. Variants of time-stamping algorithms had been known previously (also in the field of STM [12]). However, [5] guaranteed that a transaction commits within bounded time and that the competitive ratio (i.e. the ratio of the makespan of the schedule defined by an online scheduler and by an optimal offline scheduler, knowing all transactions in advance) is $O(s^2)$, where s is the number of (shared) resources of all transactions *together*. The analysis was improved to $O(s)$ in [1]. In contrast to our contribution, access to a global clock or logical counter is needed for every transaction which clearly limits the possible parallelism with a growing number of cores. In [11] a scalable replacement for a global clock was presented using synchronized clocks. Unfortunately, these days most systems come without multiple clocks. Additionally, there are problems due to the drift of physical clocks.

Also in [1] a matching lower bound of $\Omega(s)$ for the competitive ratio of any (also randomized) algorithm is proven, where the adversary can alter resource requests of waiting transactions. We show that, more generally, if an adversary can reduce the possible parallelism (i.e., the number of concurrently running transactions) by a factor k, the competitive ratio is $\Omega(k)$ for deterministic algorithms and for randomized algorithms the expected ratio is $\Omega(\min\{k, \sqrt{m}\})$, where m is the number of cores. In the analysis of [1] an adversary can change the required resources such that instead of $\Omega(s)$ transactions only $O(1)$ can run in parallel, i.e. all of a sudden $\Omega(s)$ transactions write to the same resource. Though, indeed the needed resources of transactions do vary over time, we believe that the reduction in parallelism is rarely that high. Dynamic data structure such as (balanced) trees and lists usually do not vary from one extreme to the other. Therefore our lower bound directly incorporates the power of the adversary.

Furthermore, the complexity measure is not really satisfying, since the number of (shared) resources in total is not correlated well to the actual conflicting transactions an individual transaction potentially encounters. As a concrete example, consider the classical dining philosophers problem, where there are n unit length transactions sharing n resources, such that transaction T_i demands resource R_i as well as $R_{(i+1) \bmod n}$ exclusively. An optimal schedule finishes in constant time $O(1)$ by first executing all even transactions and afterwards all odd transactions. The best achievable bound by any scheduling algorithm using the number of shared resources as complexity measure is only $O(n)$. Furthermore, with our more local complexity measure, we prove that for a wide variety of scheduling tasks, the guarantee for algorithm Greedy is linearly worse, whereas our randomized algorithm *RandomizedRounds* is only a factor $\log n$ off the optimal, with high probability.

We relate the problem of contention management to coloring, where a large amount of distributed algorithms are available in different models of communication and for different graphs [9, 13, 15]. Our algorithm *RandomizedRounds* essentially computes a $O(\max\{\Delta, \log n\})$ coloring for a graph with maximum degree Δ.

For further related work in respect to online scheduling, transactional memory systems and coloring, see [14].

3 Model

A set of *transactions* $S_T := \{T_1, ..., T_n\}$ sharing up to s *resources* (such as memory cells) are executed on m *processors* $P_1, ..., P_m$.[1] For simplicity of the analysis we assume that a single processor runs one *thread* only, i.e., in total at most m threads are running concurrently. If a thread running on processor P_i creates transactions $T_0^{P_i}, T_1^{P_i}, T_2^{P_i}, ...$ one after the other, all of them are executed sequentially on the same processor, i.e., transaction $T_j^{P_i}$ is executed as soon as $T_{j-1}^{P_i}$ has completed, i.e. committed.

The *duration* of transaction T is denoted by t_T and refers to the time T executes until commit without contention (or equivalently, without interruption). The length of the longest transaction of a set S of transactions is denoted by $t_S^{max} := \max_{K \in S} t_K$. If an adversary can modify the duration of a transaction arbitrarily during the execution of the algorithm, the competitive ratio of any online algorithm is unbounded: Assume two transactions T_0 and T_1 face a conflict and an algorithm decides to let T_0 wait (or abort). The adversary could make the opposite decision and let T_0 proceed such that it commits at time t_0. Then it sets the execution time T_0 to infinity, i.e., $t_{T_0} = \infty$ after t_0. Since in the schedule produced by the online algorithm, transaction T_0 commits after t_0 its execution time is unbounded. Therefore, in the analysis we assume that t_T is fixed for all transactions T.[2] We consider an *oblivious adversary* that knows the (contention management) algorithm, but does not get to know the randomized choices of the algorithm before they take effect.

Each transaction consists of a sequence of operations. An operation can be a read or write access of a shared resource R or some arbitrary computation. A value written by a transaction T takes effect for other transactions only after T commits. A transaction either successfully finishes with a commit after executing all operations and acquiring all modified (written) resources or unsuccessfully with an abort anytime. A resource can be acquired either once it is used for the first time or at latest at commit time. A resource can be read in parallel by arbitrarily many transactions. A read of transaction A of resource R is *visible*, if another transaction B accessing R after A is able to detect that A has already read R. We assume that conflicts that all reads are visible. In fact, we prove in [15] that systems with invisible readers can be very slow. To perform a write, a resource must be acquired exclusively. Only one transaction at a time

[1] Transactions are sometimes called jobs, and machines are sometimes called cores.

[2] In case the running time depends on the state/value of the resources and therefore the duration varied by a factor of c, the guarantees for our algorithms (see Section 6) would worsen only by the same factor c.

can hold a resource exclusively. This leads to the following types of *conflicts*: (i) Read-Write: A transaction B tries to write to a resource that is read by another transaction A. (ii) Write-Write: A transaction tries to write to a resource that is already held exclusively (written) by another transaction, (iii) Write-Read: A transaction tries to read a resource that is already held exclusively (write) by another transaction. A contention manager comes into play if a conflict occurs and decides how to resolve the conflict. It can make a transaction wait (arbitrarily long), or abort, or assist the other transaction. We do not explicitly consider the third option. Helping requires that a transaction can be parallelized effectively itself, such that multiple processors can execute the same transaction in parallel with low coordination costs. In general, it is difficult to split a transaction into subtasks that can be executed in parallel. Consequently, state of the art systems do not employ helping. If a transaction gets aborted due to a conflict, it restores the values of all modified resources, frees its resources and restarts from scratch with its first operation. A transaction can request different resources in different executions or change the requested resource while waiting for another transaction.

We assume that a transaction notices a conflict once it actually occurs and a contention manager is called right away, i.e. *eagerly*.[3] A transaction keeps a resource locked until commit, i.e. *no early release*. By introducing additional writes in our examples, any transaction indeed cannot release its resources before commit.

A *schedule* shows for each processor P at any point in time whether it executes some transaction $T \in S_T$ or whether it is idle. The *makespan* of a schedule for a set of transactions S_T is defined as the duration from the start of the schedule until all transactions S_T have committed. We say a schedule for transactions S_T is *optimal*, if its makespan is minimum possible. We measure the quality of a contention manager in terms of the makespan. A contention manager is *optimal*, if it produces an optimal schedule for every set of transactions S_T.

4 Lower Bounds

Before elaborating on the problem complexity of contention management, we introduce some notation related to graph theory and scheduling. We show that even coarse approximations are NP-hard to compute. In [14] we give a lower bound of $\Omega(n)$ for the competitive ratio of algorithms Polka, SizeMatters and Greedy, which holds even if resource requirements remain the same over time.

4.1 Notation

We use the notion of a *conflict graph* $G = (S, E)$ for a subset $S \subseteq S_T$ of transactions executing concurrently, and an edge between two conflicting transactions. The neighbors of transaction T in the conflict graph are denoted by N_T and represent all transactions that have a conflict with transaction T in G. The degree d_T of a transaction T in the graph corresponds to the number of neighbors

[3] Even for "typical" cases neither eager nor lazy conflict handling consistently outperforms the other.

in the graph, i.e., $d_T = |N_T|$. We have $d_T \leq |S| \leq \min\{m, n\}$, since at most m transactions can run in parallel, and since there are at most n transactions, i.e., $|S_T| = n$. The maximum degree Δ denotes the largest degree of a transaction, i.e., $\Delta := \max_{T \in S} d_T$. The term t_{N_T} denotes the total time it takes to execute all neighboring transactions of transaction T sequentially without contention, i.e., $t_{N_T} := \sum_{K \in N_T} t_K$. The time $t^+_{N_T}$ includes the execution of T, i.e., $t^+_{N_T} = t_{N_T} + t_T$. Note that the graph G is highly dynamic. It changes due to new or committed transactions or even after an abort of a transaction. Therefore, by d_T we refer to the maximum size of a neighborhood of transaction T that might arise in a conflict graph due to any sequence of aborts and commits. If the number of processors equals the number of transactions ($m = n$), all transactions can start concurrently. If, additionally, the resource requirements of transactions stay the same, then the maximum degree d_T can only decrease due to commits. However, if the resource demands of transactions are altered by an adversary, new conflicts might be introduced and d_T might increase up to $|S_T|$.

4.2 Problem complexity

If an adversary is allowed to change resources after an abort, such that all restarted transactions require the same resource R, then for all aborted transactions T we can have $d_T = \min\{m, n\}$. This means that no algorithm can do better than a sequential execution (see lower bound in [1]).

We show that even if the adversary can only choose the initial conflict graph and does not influence it afterwards, it is computationally hard to get a reasonable approximation of an optimal schedule. Even, if the whole conflict graph is known and fixed, the best approximation of the schedule obtainable in polynomial time is exponentially worse than the optimal. The claim follows from a straight forward reduction to coloring. For the proof we refer to [14].

Theorem 1 *If the optimal schedule requires time k, it is NP-hard to compute a schedule of makespan less than $k^{\frac{\log k}{25}}$ (for sufficiently large constants), even if the conflict graph is known and transactions do not change their resource requirements.*

4.3 Power of the adversary

We show that if the conflict graph can be modified, the competitive ratio is proportional to the possible change of a transaction's degree. Initially, a contention manager is not aware of any conflicts. Thus, it is likely to schedule (many) conflicting transactions. All transactions that faced a conflict (and aborted) change their resources on the next restart and require the same resource. Thus they must run sequentially. The contention manager might schedule transactions arbitrarily – in particular it might delay any transaction for an arbitrary amount of time (even before it executed the first time). The adversary has control of the initial transactions and can state how they are supposed to behave after an abort (i.e. if they should change their resource requirements). During the execution, it cannot alter its choices. Furthermore, we limit the power of the adversary as follows: Once the degree of a transaction T has increased by a factor of k, no

new conflicts will be added for T, i.e. all initial proposals by the adversary for resource modifications augmenting the degree of T are ignored from then on.

The proof of the following theorem can be found in[14].

Theorem 2 *If the conflict graph can be modified by an oblivious adversary such that the degree of any transaction is increased by a factor of k, any deterministic contention manager has competitive ratio $\Omega(k)$ and any randomized has $\Omega(\min\{k, \sqrt{m}\})$.*

5 Algorithms

Our first algorithm *CommitRounds* (Section 5.1) gives assertions for the response time of individual transactions, i.e., how long a transaction needs to commit. Although we refrain from using global data and we can still give guarantees on the makespan, the result is not satisfying from a performance point of view, since the worst-case bound on the makespan is not better than a sequential execution. Therefore we derive a randomized algorithm *RandomizedRounds* (Section 5.2) with better performance.

Algorithm Commit Rounds (*CommitRounds*)

On conflict of transaction T^{P_i} with transaction T^{P_j}:
$c_{P_i}^{max} := \max\{c_{P_i}^{max}, c_{P_j}^{max}\}$
$c_{P_j}^{max} := c_{P_i}^{max}$
if $c_{P_i} < c_{P_j} \vee (c_{P_i} = c_{P_j} \wedge P_i < P_j)$
then Abort transaction T^{P_j}
else Abort transaction T^{P_i}
end if

After commit of transaction T^P:
$c_P^{max} := c_P^{max} + 1$
$c_P := c_P^{max}$

5.1 Deterministic algorithm *CommitRounds*

The idea of the algorithm is to assign priorities to processors, i.e. a transaction T^P running on a processor P inherits P's priority, which stays the same until the transaction committed. When T commits, P's priority is altered, such that any transaction K having had a conflict with transaction T will have higher priority than all following transactions running on P. For the first execution of the first transaction on processor P_i, the variable $c_{P_i}^{max}$ and c_{P_i} are initialized with 0. We refer to the pseudocode of algorithm *CommitRounds* for details and to [14] for a more detailed textual description.

Algorithm Randomized Rounds (*RandomizedRounds*)

procedure Abort(transaction T, K)
  Abort transaction K
  K waits for T to commit or abort before restarting
end procedure

On (re)start of transaction T:
  $x_T :=$ random integer in $[1, m]$

On conflict of transaction T with transaction K:
  if $x_T < x_K$ **then** Abort(T, K)
    else Abort(K, T)
  end if

5.2 Randomized algorithm *RandomizedRounds*

For our randomized algorithm *RandomizedRounds* a transaction chooses a discrete number uniformly at random in the interval $[1, m]$ on start up and after every abort. In case of a conflict the transaction with the smaller random number proceeds and the other aborts. The routine Abort(transaction T, K) aborts transaction K. Moreover, K must hold off on restarting until T committed or aborted.

To incorporate priorities set by a user, a transaction simply has to modify the interval from which its random number is chosen. For example, choosing from $[1, \lfloor \frac{m}{2} \rfloor]$ instead of $[1, m]$ doubles the chance of succeeding in a round.[4]

6 Analysis

We study two classic efficiency measures of contention management algorithms, the makespan (the total time to complete a set of transactions) and the response time of the system (how long it takes for an individual transaction to commit).

6.1 Deterministic Algorithm *CommitRounds*

Theorem 3 *Any transaction will commit after being in the system for a duration of at most* $2 \cdot m \cdot t_{S_T}^{max}$.

Proof. When transaction T^{P_i} runs and faces a conflict with a transaction T^{P_j} having lower priority than T^{P_i} i.e., $c_{P_i} < c_{P_j}$ or $c_{P_i} = c_{P_j}$ and also $P_i < P_j$, then T^{P_j} will lose against T^{P_i}. If not, transaction T^{P_j} will have $c_{P_j}^{max} \geq c_{P_i}^{max} \geq c_{P_i}$ after winning the conflict. Thus at latest after time $t_{S_T}^{max}$ one of the following two scenarios will have happened: The first is that T^{P_j} has committed and all transactions running on processor P_j later on will have $c_{P_j} > c_{P_j}^{max} \geq c_{P_i}^{max} \geq c_{P_i}$. The second is that T^{P_j} has had a conflict with another transaction T^{P_k} for which

[4] Any interval yields the same guarantees on the makespan as long as the number of distinct possible (random) values is at least m, i.e., the maximal number of parallel running jobs.

will also hold that $c_{P_k}^{max} \geq c_{P_i}^{max}$ after the conflict. Thus after time $t_{S_T}^{max}$ either a processor has got to know $c_{P_i}^{max}$ (or a larger value) or committed knowing $c_{P_i}^{max}$ (or a larger value). In the worst-case one processor after the other gets to know $c_{P_i}^{max}$ within time $t_{S_T}^{max}$, taking time at most $m \cdot t_{S_T}^{max}$ and then all transactions commit one after the other, yielding the bound of $2 \cdot m \cdot t_{S_T}^{max}$.

6.2 Randomized Algorithm *RandomizedRounds*

To analyze the response time, we use a complexity measure depending on local parameters, i.e., the neighborhood in the conflict graph (for definitions see Section 4.1).

Theorem 4 *The time span a transaction T needs from its first start until commit is $O(d_T \cdot t_{N_T^+}^{max} \cdot \log n)$ with probability $1 - \frac{1}{n^2}$.*

Proof. Consider an arbitrary conflict graph. The chance that for a transaction T no transaction $K \in N_T$ has the same random number given m discrete numbers are chosen from an interval $[1, m]$ is: $p(\nexists K \in N_T | x_K = x_T) = (1 - \frac{1}{m})^{d_T} \geq (1 - \frac{1}{m})^m \geq \frac{1}{e}$. We have $d_T \leq \min\{m, n\}$ (Section 4.1). The chances that x_T is at least as small as x_K of any transaction $K \in N_T$ is $\frac{1}{d_T+1}$. Thus the chance that x_T is smallest among all its neighbors is at least $\frac{1}{e \cdot (d_T+1)}$. If we conduct $y = 32 \cdot e \cdot (d_T + 1) \cdot \log n$ trials, each having success probability $\frac{1}{e \cdot (d_T+1)}$, then the probability that the number of successes X is less than $16 \cdot \log n$ becomes (using a Chernoff bound): $p(X < 16 \cdot \log n) < e^{-2 \cdot \log n} = \frac{1}{n^2}$

The duration of a trial, i.e., the time until T can pick a new random number, is at most the time until the first conflict occurs, i.e., the duration t_T plus the time T has to wait after losing a conflict, which is at most $t_{N_T}^{max}$. Thus the duration of a trial is bounded by $2 \cdot t_{N_T^+}^{max}$.

Theorem 5 *If n transactions $S = \{T^{P_0}, ..., T^{P_n}\}$ run on n processors, then the makespan of the schedule by algorithm* RandomizedRounds *is $O(\max_{T \in S_T}(d_T \cdot t_{N_T^+}^{max}) \cdot \log n) + t_{last}$ with probability $1 - \frac{1}{n}$, where t_{last} is the time, when the latest transaction started to execute.*

Proof. Once all transactions are executing, we can use Theorem 4 to show that $p(\exists K \in S$ finishing after $O(\max_{T \in S} d_T \cdot t_{N_T^+}^{max}) \cdot \log n) < \frac{1}{n}$. In the proof of Theorem 4, we showed that for any transaction T: $p(T$ finishes after $O(d_T \cdot t_{N_T^+}^{max} \cdot \log n) < \frac{1}{n^2}$. Since $O(d_T \cdot t_{N_T^+}^{max} \cdot \log n) \leq O(\max_{T \in S}(d_T \cdot t_{N_T^+}^{max}) \cdot \log n)$ we have $p(T$ finishes after $O(\max_{T \in S}(d_T \cdot t_{N_T^+}^{max}) \cdot \log n) < \frac{1}{n^2}$. The chance that no transaction out of all n transactions exceeds the bound of $O(\max_{T \in S}(d_T \cdot t_{N_T^+}^{max}) \cdot \log n)$ is $(1 - \frac{1}{n^2})^n \geq 1 - \frac{1}{n}$.

The theorem shows that if an adversary can increase the maximum degree d_T by a factor of k the running time also increases by the same factor. The bound still holds if an adversary can keep the degree constantly at d_T despite committing transactions. In practice, the degree might also be kept at the same

level due to new transactions entering the system. In case, we do not allow any conflicts to be added to the initial conflict graph, the bound of Theorem 5 (and also the one of Theorem 4) can be improved to $O(\max_{T \in S_T}(\max\{d_T, \log n\} \cdot t^{max}_{N^+_T}))$, with an analogous derivation as in [15]. As explained in [14] the schedule corresponds then to a coloring using $O(\max\{\Delta, \log n\})$ colors.

Let us consider an example to get a better understanding of the bounds. Assume we have n transactions starting on n processors having equal length t. All transactions only need a constant amount of resources exclusively and each resource is only required by a constant number of transactions, i.e., d_T is a constant for all transactions T – as is the case in the dining philosophers problem mentioned in Section 2. Then the competitive ratio is $O(\log n)$, whereas it is $O(n)$ for the Greedy, Polka and SizeMatters algorithms as shown in [14].

References

1. H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC*, 2006.
2. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.
3. D. Dice and N. Shavit. Understanding Tradeoffs in Software Transactional Memory. In *Symp. on Code Generation and Optimization*, 2007.
4. R. Eidenbenz and R. Wattenhofer. Good Programming in Transactional Memory: Game Theory Meets Multicore Architecture. In *ISAAC*, 2009.
5. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, 2005.
6. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA Conference*, 2003.
7. M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
8. M. Herlihy and J. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. In *Symp. on Computer Architecture*, 1993.
9. F. Kuhn. Weak Graph Coloring: Distributed Algorithms and Applications. In *SPAA*, 2009.
10. H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *Symp. on Computer Architecture*, 2007.
11. T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *Parallel Algorithms and Architectures*, 2007.
12. W. Scherer and M. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.
13. J. Schneider and R. Wattenhofer. A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In *PODC*, 2008.
14. J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *TIK Technical Report 311, ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-311.pdf*, 2009.
15. J. Schneider and R. Wattenhofer. Coloring Unstructured Wireless Multi-Hop Networks. In *PODC*, 2009.
16. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10, 1997.