ELSEVIER
Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 00 (2010) 1–14
Procedia Computer Science

Bounds On Contention Management Algorithms[1]

Johannes Schneider, Roger Wattenhofer
{jschneid, wattenhofer}@tik.ee.ethz.ch
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 parallel threads.

Keywords: scheduling, transactions, transactional memory, concurrency control, contention management

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

[1]This paper is an extension of [13].

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, e.g. a shared object, 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 analyze 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. We also show that achieving a short makespan (from a worst case perspective) requires 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

In [7] Dynamic STM (DSTM) targeted towards dynamic data structures was described, which suggests the use of a contention manager as an independent module. Most proposed contention managers have been assessed by specific benchmarks only[12, 10], and not analytically. 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. [3, 10]. In [10] an algorithm called SizeMatters is introduced, which gives higher priority to the transaction that has modified more (shared) memory. We show that from a worst-case perspective Polka and SizeMatters may perform exponentially worse than *RandomizedRounds*. In [6, 16] load adaption strategies have been investigated, i.e. algorithms have been proposed that alter the number of cores that perform computations depending on previously occured conflicts.

The first analysis of a contention manager named Greedy was given in [5], using time stamps to decide in favor of older transactions. [5] guarantees 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 parallel threads. 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.

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 [15]. Our algorithm *RandomizedRounds* essentially computes a $O(\max\{\Delta, \log n\})$ coloring for a graph with maximum degree Δ.

Contention management is related to online scheduling. In contrast to contention management, most scheduling algorithms are centralized and assume known conflicts. For illustration, in [4] the competitive ratios of scheduling algorithms are given for conflicting jobs. Their algorithms are non-distributed and on arrival of a new job J all conflicting jobs of J are known all at once, taking effect immediately, without change. Furthermore, the completion of a job cannot create new conflicts. In our model a conflict between two transactions happens when both access the same resource, which is not necessarily directly at their start. Additionally, dynamic data structures change their structure when modified and thus a transaction might access different resources due to the commit of another transaction, which might introduce new conflicts. Therfore, it is difficult to reliably predict conflicts, since they might change any time.

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* $P_1, ..., P_m$.[2] 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. A thread running on processor P_i consists of a sequence of transactions $T_0^{P_i}, T_1^{P_i}, T_2^{P_i}, ...$ The sequence is executed sequentially on the same processor P_i, 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.[3] 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 all reads are visible. In fact, we prove in Section 4.4 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 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. It decides how to resolve the conflict by making 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.

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

[3]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.

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.

Usually *conflicts* are handled in a *lazy* or *eager* way. We assume that conflicts are handled eagerly, i.e. once a transaction tries to access a resource that is held by another transaction. For lazy conflict handling a conflict is dealt with once a conflicting transaction tries to commit. Depending on the scenario, experimental evaluation showed that one or the other way leads to a shorter makespan. Even for "typical" cases neither consistently outperforms the other. 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. 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. We consider both eager and lazy conflict handling.

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 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 can be exponentially worse than the optimal for certain graphs. The claim follows from a straight forward reduction to coloring.

Theorem 1. *If the optimal schedule requires time k, it is NP-hard to compute a schedule of makespan less than* $\max(k \cdot n^{1-1/\log^{\epsilon} n}, n)$ *(for any constant* $\epsilon > 0$*), even if the conflict graph is known and transactions do not change their resource requirements.*

Proof. Assume all accesses to resources are writes. There are n transactions of unit length, running on n processors, each transaction requires its resources on start up. Consider a coloring of the conflict graph $G = (S, E)$. Every set $C_i \subseteq S$ of transactions of color i forms an independent set (i.e., no nodes in C_i are neighbors) and thus all transactions in C_i can execute in parallel without facing any conflicts. The makespan of an optimal schedule is equal to the chromatic number $\chi(G)$, i.e., the minimum number of colors that is needed to color graph G. If this was not the case then the independent sets IS_i of the allegedly faster schedule of length l with $l < \chi(G)$ colors, formed a coloring with $C_i = IS_i$ and l colors. In [8] it was shown that computing an optimal coloring given complete knowledge of the graph is NP-hard. Even worse, computing an approximation within a factor of $n^{1-1/\log^{\epsilon} n}$ (for any constant $\epsilon > 0$ and $k \leq n^{1/\log^{\epsilon} n}$[4]) is NP-hard as well. □

As an approximation it seems reasonable to schedule transactions M, such that M is a maximum independent set (MaxIS, i.e an independent set of maximum cardinality) in $G = (S, E)$. Once all transactions in M have committed, the next MaxIS is scheduled. Iteratively scheduling a MaxIS yields a 4-approximation for the average response time or equivalently for the minimum sum of the transactions completion times [2]. Unfortunately, approximating the MaxIS problem within a factor of n^c for $c > 0$ is NP-hard [8]. Instead of a MaxIS one could try to schedule a maximal independent set (MIS, i.e., an independent set not extendable by adding a transaction). This yields a $O(\Delta \cdot t_S^{max})$ approximation. The factor t_S^{max} is a bound on how long it takes at most until the next MIS can be scheduled. So, how to obtain a MIS without any knowledge about the conflict graph? The well-known distributed algorithm by Luby [9] computes a MIS with high probability (i.e., $1 - \frac{1}{n}$) in time $O(t_S^{max} \cdot \log n)$. Unfortunately, it requires the degree of each transaction. Our Algorithm *RandomizedRounds* works for dynamic conflict graphs, does not need any information about them and can also be bounded by $O(t_S^{max} \cdot \log n)$. Thus, the total approximation ratio is $O(\Delta \cdot t_S^{max} \cdot \log n)$. In fact, for conflict graphs where no new edges (conflicts) are added, it can be improved to $O(\max\{\Delta, \log n\} \cdot t_S^{max})$.

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

[4]In case $k \cdot n^{1-1/\log^{\epsilon} n} \geq n$ all transactions execute sequentially, thus the makespan is always at most n

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.

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* $\Omega(\min\{k, \sqrt{m}\})$.

Proof. We run m transactions on m parallel threads. In the initial conflict graph each transaction faces only one conflict and all transactions have the same duration t. Thus, we have $m/2$ pairs $\{U, T\} \subseteq S_T$ of conflicting transactions. For each pair $\{U, T\}$ both transactions read the same resource R_{UT} on start-up and write it before their commits. Therefore, if two conflicting transactions start within time $t - \epsilon$ for some constant $\epsilon > 0$, both must have read R_{UT} and only one of them can commit while the other must abort. For every pair $\{U, T\}$ we can choose one transaction and let it change its resource demands after an abort, i.e. any (chosen) aborted transaction will write to resource R on startup until k transactions write to R. Thus, if k aborts take place, any schedule will be of length at least kt.

The scheduled transactions are known for a deterministic algorithm. Therefore, we can fix the transactions' resource requirements (before the start of the algorithm) such that (enough) aborts happen. Assume the algorithm schedules $x > 2$ transactions at at time. Since, the algorithm has no information about the conflicts, at least $x/3$ can be made to abort (in case three transactions are scheduled concurrently, two transactions can commit and one has to abort). We can set the aborted transactions, such that at least $\min\{x/3, k\}$ transactions write to the same resource R on startup. Thus, either a deterministic strategy schedules at most two transactions at a time or at least 1/3 of the transactions are aborted and therefore we can choose $\min\{m/3, k\}$ of them and let them write to the same resource R on startup. Therefore, the total time for a deterministic manager is $\min\{m/3, k\} \cdot t$. The optimal contention manager being aware of all conflicts finishes within time $2 \cdot t$.

Assume a randomized algorithm schedules a set $X > 4 \cdot \sqrt{m}$ of transactions at a time. Clearly, if the algorithm chooses transactions in a non-uniform manner, i.e. the chance that a pair $\{U, T\} \subseteq S_T$ is scheduled together is larger than a pair $\{V, T\} \subseteq S_T$ the adversary can make use of this knowledge. Thus, the algorithm is best off by treating all transactions equally. The chance that a transaction $T \in X$ does not face a conflict is given by $(1 - 1/m)^{|X|-1}$. The chance that none of the transactions in X is involved in a conflict is given by $(1 - 1/m)^{|X|-1} \cdot (1 - 1/m)^{|X|-2} \cdot (1 - 1/m)^{|X|-3} \cdot \ldots \cdot 1 = (1 - 1/m)^{\sum_{i=1}^{|X|-1} i} = (1 - 1/m)^{|X| \cdot (|X|-1)/2} \leq (1 - 1/m)^{8m} \leq 1/e^8$. Assume two transactions $\{U, T\}$ conflict. The algorithm must decide on one of the transactions to abort. Assume it aborts U with probability $p \geq 1/2$. Then the adversary lets U be the transaction that chooses resource R on startup. The overall chance that out of X transactions with $|X| > 4 \cdot \sqrt{m}$ one transaction aborts and chooses resource R on startup is $1/2 \cdot (1 - 1/e^8)$. If $k \cdot 4 \cdot \sqrt{m}$ transactions run in parallel then we expect at least a constant fraction of them to abort. Thus, either the algorithm schedules less than $4 \cdot \sqrt{m}$ at a time or we expect in total up to $\Omega(\min\{k, \sqrt{m}\})$ transactions to choose the same resource on startup.

□

4.4. Visible vs. Invisible Reads

We show: If an optimal contention manager is employed for a set of transactions, which do not alter their resource requirements over time, a system using visible reads can be linearly faster than a system with invisible reads. This is due to the fact that for invisible reads all aborts might take place without the influence of a contention manager, since read-write conflicts might not be handled by a contention manager, but can simply force a transaction to abort. It underlines the importance of detecting all conflicts and resolving them by a contention manager.

Theorem 3. *The competitive ratio of a system employing invisible reads is a factor* $\Omega(n)$ *worse than a system using visible reads, if both make use of an optimal contention manager.*

Proof. Suppose we have n processors and schedule $2 \cdot n$ transactions, i.e., transactions $T_0^{P_i}$ and $T_1^{P_i}$ on processor P_i. All transactions $T_0^{P_i}$ with $0 \leq i \leq n-1$ start at the same time, read resource R_i on startup and have duration $n + 2 \cdot i \cdot \epsilon$. Transactions $T_1^{P_i}$ with $0 \leq i \leq n-1$ write to all resources R_j $i < j \leq n-1$ on startup and have duration ϵ.

For invisible reads transaction $T_0^{P_0}$ commits after time n and $T_0^{P_1}$ after time $n+\epsilon$. A transaction $T_0^{P_i}$ with $1 \leq i \leq n-1$ will abort at time $n + 2 \cdot i \cdot \epsilon$. After time $2 \cdot (n + 2 \cdot \epsilon)$ transaction $T_0^{P_2}$ commits and ϵ time units later $T_1^{P_2}$. Again all transactions $T_0^{P_i}$ with $2 \leq i \leq n-1$ abort. Thus, all transactions $T_0^{P_i}$ with $0 \leq i \leq n-1$ execute sequentially. The time it takes until all transactions have committed is lower bounded by $\Omega(n^2)$.

For visible reads, the contention manager decides to give all transactions $T_0^{P_i}$ with $0 \leq i \leq n-1$ higher priority. Afterwards all transactions $T_1^{P_i}$ with $0 \leq i \leq n-1$ execute sequentially. Therefore, the makespan equals $n + 3 \cdot n \cdot \epsilon = O(n)$. □

4.5. Competitive Ratio of Algorithm Greedy

The next theorem states that for certain problem instances algorithm Greedy [5] executes a large fraction of transactions entirely sequentially, even if a large amount of them could be run in parallel. In contrast to the lower bound in [1], our lower bound holds even if transactions do not modify their resource requirements after an abort (i.e. the adversary must not alter the demanded resources of a transaction). In algorithm Greedy each transaction gets a unique time stamp on start up and keeps it until commit. In case of a conflict, the older transaction proceeds. The younger aborts, if it has already acquired the resource needed by the older transaction, otherwise it waits. A waiting transaction is always aborted.

Theorem 4. *Algorithm Greedy [5] has competitive ratio of* $\Omega(n)$ *even if transactions do not alter their resource requests over time.*

Proof. Consider the dining philosophers problem (see Section 2) and assume eager conflict handling. Suppose all transactions have unit length and transaction i requires its first resource R_i at time 0 and its second $R_{(i+1) \bmod n}$ at time $1 - i \cdot \epsilon$. Since the algorithm is deterministic, we know the time stamp of each transaction. Let transaction i have the i^{th} oldest time stamp. At time $1 - i \cdot \epsilon$ transaction $i+1$ with $i \geq 1$ will get aborted by transaction i and only transaction 1 will commit at time 1. After every abort transaction i restarts ϵ time units before transaction $i-1$.

Since transaction $i-1$ acquires its second resource $(i-1)\cdot\epsilon$ time units before its termination, transaction $i-1$ will abort transaction i at least $i-1$ times. Thus, after $i-1$ aborts transaction i can commit. The total time until the algorithm is done is bounded by the time transaction n stays in the system, i.e., $\sum_{i=1}^{n}(1-i\cdot\epsilon)=\Omega(n)$. An optimal schedule requires only $O(1)$ time.

For lazy conflict handling, we let transaction i have duration 1 and require its second resource at time $\frac{1}{2}$. Let all transactions start with their first operation at the same time. Just before commit every transaction i acquires resource i at the same time and also each transaction i with $i<n$ aborts transaction $i+1$ concurrently. The transaction with the oldest time stamp commits. All other transactions start again at the same time and the process repeats. □

4.6. Competitive Ratio of Algorithm SizeMatters

SizeMatters[10] decides a conflict in favor of the transaction, that has accessed (read or written) more unique bytes, i.e. an access to a memory cell is only counted once during an execution. Thus, if the same byte is accessed multiple times, the overall increase of the priority is only 1. The priority is reset to 0 on restart. After a threshold c of restarts, it reverts to time-stamp. Unfortunately, the authors in do not explain how the time-stamps are chosen. Thus, we assume that a transaction running on processor P_i gets the i^{th} smallest time-stamp.

Theorem 5. *Algorithm SizeMatters [10] has competitive ratio of $\Omega(n)$ even if transactions do not alter their resource requests over time.*

Proof. We use the same transactions as in the proof of Theorem 4 and say that an access of resource R_i equals an access to $n-i$ bytes. The rest is analogous to the proof of Theorem 4. Transaction i will always have larger priority than $i+1$, independent of whether the priority is calculated using time-stamps or the number of accessed bytes. □

4.7. Competitive Ratio of Algorithm Polka

Algorithm Polka works as follows: A transaction increases its priority by one for every acquired object until commit (it keeps its priority on abort). A transaction with higher priority can abort a lower priority one. If a transaction with lower priority wants a resource held by a transaction with higher priority, Polka waits for a number of intervals given by the difference in priority between the two conflicting transactions. The length of interval i has mean 2^i according to a fixed distribution chosen by the algorithm designer. For instance, assume transaction A wants a resource held by B and the difference in priorities is 2. After having tried to access the resource the first time, transaction A waits for a (random) time interval with mean 2^1. Then it tries to access the resource again. If it fails, it waits for a time interval with mean 2^2. If it was not able to access the resource again, transaction B is aborted, frees the resource and A can access it.

Theorem 6. *Algorithm Polka has at least competitive ratio $\Omega(n)$.*

Proof. Consider eager conflict handling and the probability that the back off time X_B is more than n time units. First, assume $p(X_B\geq n)\geq\frac{1}{n}$. Assume n transactions of unit length run on n processors. Each transaction i faces only one conflict on startup, i.e., transaction 1 with transaction 2, transaction 2 with transaction 3 etc. Therefore, directly after startup half the transactions will acquire a resource and they have priority 1, whereas the rest will wait for an interval of random length with mean 2. The probability that no transaction waits for n time units is $(1-\frac{1}{n})^{\frac{n}{2}}\leq\frac{1}{\sqrt{e}}$. Therefore the expected schedule is at least $n\cdot(1-\frac{1}{\sqrt{e}})$. An optimal schedule is of length 2.

Now assume $p(X_B \geq n) < \frac{1}{n}$ and consider two transactions T_1, T_2 of length $3 \cdot n$. Let them start simultaneously and conflict after running for time n due to resource R. Assume transaction T_1 acquires resource R and thus it has priority 1. Transaction T_2 will wait in expectation for 2 time units before aborting T_1 and increasing its priority to 1. Once T_1 aborted it will conflict again after time n with T_2. Both will have priority 1 and T_1 aborts T_2 and sets its priority to 2. The process repeats: Again T_2 will execute for n time units and then wait in expectation for 2 time units. The chance that T_2 waits until T_1 completed, i.e., at least time n, is less than $\frac{1}{n}$. Therefore in expectation n trials of duration n are needed until transaction T_2 waits long enough. In total expected time $O(n^2)$ is needed. The optimal requires time $O(n)$.

For lazy conflict handling assume that there are two transactions of equal length starting at the same time. Transaction T_1 writes to resources R_1 and R_2. So does transaction T_2 but in the opposite order. Just before trying to commit transaction T_1 acquires R_1 and at the same time transaction T_2 acquires R_2. Then T_1 aborts T_2 and concurrently T_2 aborts T_1, since both have the same priority and thus do not back off before aborting another transaction. Again, both start at the same time and the scenario repeats. Therefore the system will livelock and the competitive ratio becomes unbounded. □

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 Algorithm *CommitRounds* 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.

Furthermore, transaction T will inform every transaction (more precisely, processor) with which T conflicts, that it should set its priority (after a commit) such that transaction K can abort it. To do so every processor P maintains two variables: (i) Variable c_P represents the priority, such that the smaller c_P the higher transaction T's priority, and (ii) Variable c_P^{max} holds the next priority for a transaction running on processor P. In case a conflict occurs between transactions T^{P_i} and T^{P_j}, the transaction running on the processor P with smaller c_P proceeds. In case both processors have the same value ($c_{P_i} = c_{P_j}$), the transaction running on the processor with smaller identifier obtains the resource. The variable $c_P^{max} + 1$ is the next value for c_P, i.e., on commit we increment c_P^{max} and set $c_P := c_P^{max}$. The value of $c_{P_i}^{max}$ should be such that after a commit of a transaction running on P_i, the next transaction running on P_i should have lower priority than any transaction running on some processor P_j, that got previously aborted by the committed transaction executed on P_i, i.e. $c_{P_i} > c_{P_j}$. Thus, on every conflict we set $c_{P_i}^{max} := c_{P_j}^{max} := \max\{c_{P_i}^{max}, c_{P_j}^{max}\}$. Additionally, once the transaction running on P_i commits, we increment c_P^{max} and set $c_P := c_P^{max}$. 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.

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, if one transaction chooses from $[1, \lfloor \frac{m}{2} \rfloor]$ instead of $[1, m]$, it doubles the chance of succeeding in a round.

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

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 $c^{max}_{P_j} \geq c^{max}_{P_i} \geq c_{P_i}$ after winning the conflict. Thus, at latest after time $t^{max}_{S_T}$ 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 $c_{P_j} > c^{max}_{P_j} \geq c^{max}_{P_i} \geq c_{P_i}$. The second is that T^{P_j} has had a conflict with another transaction T^{P_k} for which will also hold that $c^{max}_{P_k} \geq c^{max}_{P_i}$ after the conflict. After time $t^{max}_{S_T}$ either a processor has got to know $c^{max}_{P_i}$ (or a larger value) or committed knowing $c^{max}_{P_i}$ (or a larger value). In the worst-case one processor after the other gets to know $c^{max}_{P_i}$ within time $t^{max}_{S_T}$, taking time at most $m \cdot t^{max}_{S_T}$ and then all transactions commit one after the other, yielding the bound of $2 \cdot m \cdot t^{max}_{S_T}$. □

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

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 $\frac{1}{d_T+1}$. The chance that x_T is smallest among all its neighbors is at least $\frac{1}{e \cdot (d_T+1)}$. If we conduct $y = 64 \cdot e \cdot (d_T+1) \cdot \log n$ 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): $p(X < 16 \cdot \log n) < e^{-4 \cdot \log n} = \frac{1}{n^4}$

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 $t^{max}_{N_T}$. The duration of a trial is bounded by $2 \cdot t^{max}_{N^+_T}$. □

Theorem 9. *If n transactions* $S = \{T^{P_0}, ..., T^{P_n}\}$ *run on n processors, then the makespan of the schedule by algorithm* RandomizedRounds *is* $O(\max_{T \in S_T}(d_T \cdot t^{max}_{N^+_T}) \cdot \log n) + t_{last}$ *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 8 to show that $p(\exists K \in S$ finishing after $O(\max_{T \in S}(d_T \cdot t^{max}_{N^+_T}) \cdot \log n) < \frac{1}{n}$. In the proof of Theorem 8, we showed that for the probability $p(E_T)$ of the event E_T for any transaction T holds: $p(E_T) = p(T$ finishes before $O(d_T \cdot t^{max}_{N^+_T} \cdot \log n) > 1 - \frac{1}{n^4}$. Since $O(d_T \cdot t^{max}_{N^+_T} \cdot \log n) \leq O(\max_{T \in S}(d_T \cdot t^{max}_{N^+_T}) \cdot \log n)$ we have $p(E_T) \geq p(T$ finishes before $O(\max_{T \in S}(d_T \cdot t^{max}_{N^+_T}) \cdot \log n) > 1 - \frac{1}{n^4}$. The probability $p(\text{not } E_{T_1} | E_{T_2})$ for two transactions T_1 and T_2 that given E_{T_2} has occurred, E_{T_1} does not happen, is at most $\frac{2}{n^4}$, since in the worst case E_{T_1} and E_{T_2} are maximal negatively correlated. That is to say, all outcomes for E_{T_1} that are excluded due to the occurrence E_{T_2} would cause E_{T_1} to occur.

Since E_{T_2} only excludes outcomes of probability $1/n^4$ its impact on the probability of E_{T_1} is also only $1/n^4$. In the same manner, we have for $p(\text{not } E_{T_1}|E_{T_2} \wedge E_{T_3}) \leq 3/n^4$, since due to E_{T_2} a fraction $1/n^4$ of all possibilities for E_{T_1} to occur are excluded and due to E_{T_3} the same amount. In general for $p(\text{not } E_{T_1}|E_{T_2} \wedge ... \wedge E_{T_n}) \leq 1/n^4 + (n-1)/n^4 \leq 1/n^3$. Thus, the probability that no transaction out of all n transactions exceeds the bound of $O(\max_{T\in S}(d_T \cdot t^{max}_{N^+_T}) \cdot \log n)$ is $(1 - \frac{1}{n^3})^n \geq 1 - \frac{1}{n}$. □

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 9 (and also the one of Theorem 8) 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 [14]. The idea of the proof is as follows: After time $d_T \cdot t^{max}_{N^+_T}$ we can show that the new maximum degree d'_T is at most $c \cdot d_T$ for a constant $c < 1$, i.e. it is reduced by a constant factor. This is because every transaction has constant probability to commit if it runs d_T times. To again reduce d'_T by a constant factor requires time $d'_T \cdot t^{max}_{N^+_T})$ time, i.e. a factor c less as before. Thus, the total time until the degree is less than one is given by $\sum_{i=0}^{O(\log n)} c^i d_T \cdot t^{max}_{N^+_T}) = O(\log n \cdot t^{max}_{N^+_T}))$.

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 and SizeMatters algorithms (see Sections 4.5 and 4.6). For the Polka contention management strategy, the examples used in the proof of Theorem 6 disclose an exponential gap between *RandomizedRounds* and Polka, since the makespan of Algorithm *RandomizedRounds* for both examples is within a factor of $O(\log n)$ of the optimal with high probability.

A frequently used heuristic for contention management algorithms is to base the priority of a transaction on some measure of the work it has already completed. Since algorithm *Randomize-dRounds* does not use any information about the progress of a transaction such as the number of accessed resources, it looks like *RandomizedRounds* does not follow this heuristic at all. However, we show that the probability that a transaction T has high priority increases with every transaction aborted due to T. Assume a set W of transactions has aborted due to T. Then the probability that the randomly chosen priority x_T is less than $a \in [1, m]$ is:

$$\begin{aligned} p(x_T \leq a) &= 1 - p(x_T > a) \\ &= 1 - p(x_K > a, \forall K \in (W \cup T)) \\ &= 1 - (1 - \frac{a}{m})^{|W|+1} \end{aligned}$$

This indicates that in general the more conflicts a transaction has won the higher are its chances to succeed in the next one as well.

7. Conclusions

In the quickly growing field of transactional memory, most research has been based on practical concerns on current systems, frequently neglecting future trends, such as the possibly fast growth of the number of cores per chip. Furthermore, evaluation has been often limited to a few selected scenarios, reflected in a couple of benchmarks. In this paper we have analyzed some well-known algorithms in the field of contention management. Additionally, we derived two algorithms that improve on existing algorithms. Our algorithms avoid using global data, which limits scalability. Our randomized algorithm improves on the (worst-case) performance of previous work dramatically, e.g. exponentially, if sufficient parallelism is possible. Due to the reduction to coloring a further improvement is difficult and for some scenarios computationally not feasible.

References

[1] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symp. on Principles of Distributed Computing (PODC)*, 2006.

[2] A. Bar-Noy, M. Bellare, M. M. Halldorsson, H. Shachnai, and T. Tamir. On chromatic sums and distributed resource allocation. *Information and Computation*, 140(2):183–202, 1998.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of the Int. Conferernce on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2006.

[4] G. Even, M. M. Halldorsson, L. Kaplan, and D. Ron. Scheduling with conflicts: online and offline algorithms. In *Journal of Scheduling*, 2008.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th ACM Symp. on Principles of Distributed Computing (PODC)*, 2005.

[6] D. Hasenfratz, J. Schneider, and R. Wattenhofer. Transactional memory: How to perform load adaption in a simple and distributed manner. In *Proc. of the 210 International Conf. on High Performance Computing & Simulation (HPCS)*, 2010.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symp. on Principles of Distributed Computing (PODC)*, 2003.

[8] S. Khot. Improved Inapproximability Results for MaxClique, Chromatic Number and Approximate Graph Coloring. In *20th Annual Symposium on Foundations of Computer Science (FOCS)*, 2001.

[9] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

[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 *Proceedings of the 24th ACM Symp. on Principles of Distributed Computing (PODC)*, 2005.

[13] J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *Proc. of the 20th International Symposium on Algorithms and Computation (ISAAC)*, 2009.

[14] J. Schneider and R. Wattenhofer. Coloring Unstructured Wireless Multi-Hop Networks. In *Proceedings of the 28th ACM Symp. on Principles of Distributed Computing (PODC)*, 2009.

[15] J. Schneider and R. Wattenhofer. A New Technique For Distributed Symmetry Breaking. In *Symp. on Principles of Distributed Computing(PODC)*, 2010.

[16] G. Sharma, B. Estrade, and C. Busch. Window-based greedy contention management for transactional memory. In *Proc. of the 24th Int. Symposium on Distributed Computing (DISC)*, 2010.