Stable Scheduling in Transactional Memory

Costas Busch[1], Bogdan S. Chlebus[1], Dariusz R. Kowalski[1], and Pavan Poudel[2]

[1] Augusta University, Augusta, Georgia, USA
{kbusch,bchlebus,dkowalski}@augusta.edu
[2] ATGWORK, Norcross, Georgia, USA
poudelpavan@gmail.com

Abstract. We study computer systems with transactions executed on a set of shared objects. Transactions arrive continually subject to constraints that are framed as an adversarial model and impose limits on the average rate of transaction generation and the number of objects that transactions use. We show that no deterministic distributed scheduler in the queue-free model of transaction autonomy can provide stability for any positive rate of transaction generation. Let a system consist of m shared objects and an adversary be constrained such that each transaction may access at most k shared objects. We prove that no scheduler can be stable if a generation rate is greater than $\max\{\frac{2}{k+1}, \frac{2}{\lfloor\sqrt{2m}\rfloor}\}$. We develop a centralized scheduler that is stable if a transaction generation rate is at most $\max\{\frac{1}{4k}, \frac{1}{4\lceil\sqrt{m}\rceil}\}$. We design a distributed scheduler in the queue-based model of transaction autonomy, in which a transaction is assigned to an individual processor, that guarantees stability if the rate of transaction generation is less than $\max\{\frac{1}{6k}, \frac{1}{6\lceil\sqrt{m}\rceil}\}$. For each of the schedulers we give upper bounds on the queue size and transaction latency in the range of rates of transaction generation for which the scheduler is stable.

Keywords: Transactional memory, shared object, dynamic transaction generation, adversarial model, stability, latency.

1 Introduction

Threads that execute concurrently need to synchronize access to shared objects to avoid conflicts. Traditional low-level thread synchronization mechanisms such as locks and barriers are prone to deadlock and priority inversion, among multiple vulnerabilities. The concept of transactional memory has emerged as a high-level abstraction of the functionality of multiprocessor systems, see Herlihy and Moss [17] and Shavit and Touitou [21]. The idea is to designate blocks of program code as 'transactions' to be executed atomically. Transactions are executed speculatively, in the sense that if a transaction aborts due to synchronization conflicts or failures, then the transaction's execution is rolled back to be restarted later. A transaction commits if there are no conflicts or failures, and its effects become visible to all processes. If multiple transactions concurrently attempt to access the same object, then this creates a conflict for access and

could trigger aborting some of the involved transactions. The synchronization conflict between the transactions is handled by contention managers, also known as schedulers, see Hendler and Suissa-Peleg [16] and Spear et al. [22]. Schedulers determine an execution schedule for transactions striving to avoid conflicts for access to shared objects.

The adversarial models of generating transactions that we use are inspired by the adversarial queueing theory, which has been applied to study stability of routing algorithms with packets injected continually. Routing of packets in communication networks is constrained by properties of networks, like their topology and capacities of links or channels. In the case of transactional memory, executing multiple transactions concurrently is constrained by the requirement that a transaction needs to have an exclusive access to each object it wants to interact with in order to be executed succesfully.

A computer system includes a fixed set of shared objects. Transactions are spawned continually. The system is synchronous in that an execution of an algorithm scheduling transactions is structured into rounds. It takes one round to execute a transaction successfully. Multiple transactions can be invoked concurrently, but a transaction requires exclusive access to each object that it needs to interact with in order to be executed successfully. If multiple transactions accessing the same object are invoked at a round then all of them are aborted. The arrival of threads with transactions is governed by an adversarial model with parameters bounding the average generation rate and the number of transactions that can be generated at one round. Processed transactions may be additionally constrained by imposing an upper bound on the number of objects a transaction needs to access.

The task for such a computer system is to eventually execute each generated transaction, while striving to minimize the number of pending transactions at any round and the time a pending transaction spends waiting for execution. Once a transaction is generated, it may need to wait to be invoked. It is a scheduling algorithm that manages the timings of invocations of pending transactions. We consider both centralized and distributed schedulers.

There are two models of generating transactions which specify the autonomy of individual transactions. In the queue-free case, for each transaction there is a corresponding autonomous processor responsible for its execution. A distributed scheduler in the queue-free model is executed by the processors that attempt to invoke transactions on shared objects. In the queue-based model, there is a fixed set of processors, and each thread with a transaction is assigned to a processor. A distributed scheduler in the queue-based model is executed by the processors that communicate through the shared objects by performing transactions on them. A centralized scheduler is not affected by constraints on autonomy of each transaction, since all pending transactions are managed en masse. The schedulers we consider are deterministic, in that they do not resort to randomization.

The contributions. We show first that no deterministic distributed scheduler in the queue-free model of transaction autonomy can provide stability for any positive rate of transaction generation. Let a computer system consist of m

Scheduler	Lower bound	Upper bound
distributed queue-free	stability impossible	
centralized	$\rho > \max\{\frac{2}{k+1}, \frac{2}{\lfloor\sqrt{2m}\rfloor}\}$	$\rho \leq \max\{\frac{1}{4k}, \frac{1}{4\lceil\sqrt{m}\rceil}\}$
distributed queue-based		$\rho < \max\{\frac{1}{6k}, \frac{1}{6\lceil\sqrt{m}\rceil}\}$

Table 1. A summary of the ranges of rates of transaction generation for which deterministic schedulers are stable. The used notations are as follows: m is the number of shared objects, k is the maximum number of shared objects accessed by a transaction, and ρ is the rate of transaction generation. Upper bounds limit transaction generation rates for which stability is achievable. Lower bounds limit transaction generation rates for which stability is not possible. A lower bound for centralized schedulers holds a priori for distributed queue-based schedulers.

shared objects and the adversary be constrained such that each transaction needs to access at most k of the shared objects. We show that no scheduler can be stable if a generation rate is greater than $\max\{\frac{2}{k+1}, \frac{2}{\lfloor\sqrt{2m}\rfloor}\}$. We develop a centralized scheduler that is stable if the transaction generate rate is at most $\max\{\frac{1}{4k}, \frac{1}{4\lceil\sqrt{m}\rceil}\}$. We design a distributed scheduler, in the queue-based model of transaction autonomy in which a transaction is assigned to an individual processor, that guarantees stability if the rate of transaction generation is less than $\max\{\frac{1}{6k}, \frac{1}{6\lceil\sqrt{m}\rceil}\}$. For each of the two schedulers we develop, we give upper bounds on the queue size and transaction latency in the range of rates of transaction generation for which the scheduler is stable. Table 1 gives a summary of the ranges of rates of transaction generation for which deterministic schedulers are stable.

Related work. Scheduling transactions has been studied for both shared memory multi-core and distributed systems. Most of the previous work on scheduling transactions considered an offline case where all transactions are known at the outset. Some previous work considered online scheduling where a batch of transactions arrives one by one and the performance of an online scheduler is compared to a scheduler processing the batch offline. No previous work known to the authors of this paper addressed dynamic transaction arrivals with potentially infinitely many transactions to be scheduled in a never-ending execution.

Attiya et al. [4] and Sharma and Busch [19],[20] considered transaction scheduling in distributed systems with provable performance bounds on communication cost. Transaction scheduling in a distributed system with the goal of minimizing execution time was first considered by Zhang et al. [25]. Busch et al. [8] considered minimizing both the execution time and communication cost simultaneously. They showed that it is impossible to simultaneously minimize execution time and communication cost for all the scheduling problem instances in arbitrary graphs even in the offline setting. Specifically, Busch et al. [8] demon-

strated a tradeoff between minimizing execution time and communication cost and provided offline algorithms that separately optimizw execution time and communication cost. Busch et al. [9] considered transaction scheduling tailored to specific popular topologies and provided offline algorithms that minimize simultaneously execution time and communication cost. Distributed directory protocols have been designed by Herlihy and Sun [18], Sharma and Busch [19], and Zhang et al. [25], with the goal to optimize communication cost in scheduling transactions. Zhang and Ravindran [23] provided a distributed dependency-aware model for scheduling transactions in a distributed system that manages dependencies between conflicting and uncommitted transactions such that they can commit safely. This model has the inherent tradeoff between concurrency and communication cost. Zhang and Ravindran [24] provided cache-coherence protocols for distributed transactional memory based on a distributed queuing protocol. Attiya et al. [3] and Attiya and Milani [5] studied competitive performance of schedulers compared to the clairvoyant one. Busch et al. [10] studied online algorithms to schedule transactions for distributed transactional memory systems where transactions residing at nodes of a communication graph operate on shared, mobile objects.

Adversarial queuing is a methodology to capture stability of processing incoming tasks without any statistical assumptions about task generation. It provides a framework to develop worst-case bounds on performance of deterministic distributed algorithms in a dynamic setting. This approach to study routing algorithms in store-and-forward networks was proposed by Borodin et al. [7], and continued by Andrews et al. [2]. Adversarial queuing has been applied to other dynamic tasks in communication networks. Bender et al. [6] considered broadcasting in multiple-access channels with queue-free stations in the framework of adversarial queuing. Chlebus et al. [13] proposed to investigate deterministic distributed broadcast in multiple access channels performed by stations with queues in the adversarial setting. This direction was continued by Chlebus et al. [12] who studied the maximum throughput in such a setting. Anantharamu et al. [1] considered packet latency of deterministic broadcast algorithms with injection rates less than 1. Chlebus et al. [11] studied adversarial routing in multiple-access channels subject to energy constraints. Garncarek et al. [14] investigated adversarial stability of memoryless packet scheduling policies in multiple access channels. Garncarek et al. [15] studied adversarial communication through channels with collisions between communicating agents represented as graphs.

2 Technical Preliminaries

A distributed system consists of processors and a fixed set of m shared objects. The system executes an algorithm. An execution of the algorithm is synchronous in that it is partitioned into time steps, which we call *rounds*. Intuitively, the executed algorithm spawns threads and each thread generates and executes transactions. To simplify the model of transaction generation and scheduling, we assign transactions directly to processors and disregard threads entirely.

We consider two frameworks of generating transactions. In the *queue-based* model, we assume a fixed number of processors in the system, each with a unique name. Each new transaction is generated at a specific round and assigned to one such a processor. All the transactions at a processor pending at a round make its *queue* at the round. In a *queue-free* model of transaction autonomy, each new transaction generated at a round is associated with an anonymous virtual processor that exists only for the purpose to execute this transaction and disappears after the transaction's successful execution.

The *type* of a transaction is the set of objects it may need to access during execution. To determine the type of a transaction, it suffices to read it to list all the mentioned objects. The number of objects in a transaction's type is the *weight* of this transaction and also of the type. If the types of two transactions share an object, then we say that this creates a *conflict for access* to this object, and that the transactions involved in a conflict for access to an object *collide* at this object. A set of transactions with the property that no two different transactions in the set collide at some shared object is called *conflict free*.

Scheduling transactions. Transactions are managed by a *scheduler*. This is an algorithm that determines for each round which pending transactions are invoked at this round. A transaction invoked at a round that gets executed successfully is no longer pending, while an aborted transaction stays pending at the next round. Scheduling transactions is constrained by whether this is a queue-free or queue-based model. In the queue-free model, transactions are managed en-masse and only conflicts for access to objects determine feasibility of concurrently performing a set of transactions. This means that if a pending transaction invoked at a round is not involved in conflict with any object it needs to access, for any of the transactions invoked at this round concurrently, then this transaction is executed successfully. In particular, if a set of transactions is conflict-free then all the transactions in this set can be executed together at one round. We assume conservatively that if a pending transaction invoked at a round is involved in conflict with some other transaction invoked concurrently, for an object they need to access, then all such transactions get aborted at this round and stay pending at the next round. The queue-based model is more restricted, in that the queue-free model's constraints on concurrent execution of transactions do apply, but additionally, for each processor, at most one transaction in this processor's queue can be performed at a round.

A *centralized scheduler* is an sequential algorithm that knows all the transactions pending at a round and receives instantaneous feedback from each object about committing to an invoked transaction or aborting it. Such a scheduler can invoke concurrently any set of pending transactions at a round in the queue-free model, but at most one transaction in the queue of a processor in the queue-based model.

A *distributed scheduler* is a distributed algorithm executed by all the involved processors. The processors communicate among themselves through shared objects. These are the processors that determine the distributed system in the queue-based case, and anonymous processors in queue-free case, one dedicated

processor per each transaction. If a processor invokes a transaction at a round, it receives an instantaneous feedback from each object of the type of the transaction about committing to an invoked transaction or aborting it.

Adversaries generate transactions. We consider a setting in which new transactions arrive continually to the system. The process of generating transactions is represented quantitatively by adversarial models. We study two types of adversaries corresponding to the queue-free and queue-based models. In the queue-free model, a transaction generated at a round contributes a unit to the *congestion* at the round at each object the transaction includes in its type. This is the *queue-free adversary*. In the queue-based model, a transaction generated at a round at a processor contributes a unit to the *congestion* at the round at each object the transaction includes in its type and also to the processor the transaction is generated at. This is the *queue-based adversary*.

Quantitative restrictions imposed on adversaries are expressed in terms of bounds on congestion. A queue-free adversary generates transactions with *generation rate* ρ and *burstiness component* b if, in each contiguous time interval τ of length t and for each shared object, the amount of congestion created for the object at all the rounds in τ together is at most $\rho t + b$. A queue-based adversary generates transactions with *generation rate* ρ and *burstiness component* b if, in each contiguous time interval τ of length t and for each shared object and for each processor, the amount of congestion created for the object at all the rounds in τ together is at most $\rho t + b$ and the amount of congestion created for the processor at all the rounds in τ together is at most $\rho t + b$. For these adversarial models, we assume that $\rho > 0$ is a real number and $b > 0$ is an integer. Given the parameters ρ and b, such an adversary is said to be of *type* (ρ, b). The *burstiness* of an adversary is the maximum number of transactions the adversary can generate in one round.

Performance of scheduling. A scheduler is *stable*, against a given type of adversary, if the number of pending transactions stays bounded in the course of any execution in which transactions are generated by the adversary of this type. For an object and a round number r, at most r transactions that contributed to congestion at this object can get executed in the first r rounds. It follows that no scheduler can be stable if its injection rate is greater than 1. In view of this, we consider only adversaries of types (ρ, b) in which $0 < \rho \leq 1$. A transaction's *delay* is the number of rounds between its generation and successful execution. The *latency* of a scheduler in an execution is the maximum delay of a transaction generated in the execution.

Proposition 1. *No deterministic distributed scheduler for a system with one shared object can be stable against a queue-free adversary of type* $(\rho, 2)$*, for any constant* $\rho > 0$*.*

In view of Proposition 1, we will consider a centralized deterministic scheduler for the queue-free model.

3 A Lower Bound

We show that no scheduler can handle dynamic transactions if a generation rate is sufficiently high with respect to the number of shared objects m and an upper bound k on the weight of a transaction. If a and b are integers where $a \leq b$ then let $[a, b]$ denote the set of integers $\{a, a+1, \ldots, b\}$.

Lemma 1. *For an integer $n > 0$, there is a family of sets $A_1, A_2, \ldots, A_{n+1}$, each a subset of $[1, \frac{n(n+1)}{2}]$, such that every set A_i has n elements, any two sets A_i and A_j, for $i \neq j$, share an element, and each element of $[1, \frac{n(n+1)}{2}]$ belongs to exactly two sets A_i and A_j, for $i \neq j$.*

We give a lower bound on generation rate to keep scheduling stable.

Theorem 1. *A queue-free adversary of type (ρ, b) generating transactions for a system of m objects such that each transaction is of weight at most k can make a scheduling algorithm unstable if injection rate ρ satisfies $\rho > \max\{\frac{2}{k+1}, \frac{2}{\lfloor\sqrt{2m}\rfloor}\}$.*

Proof. Let the m objects be denoted as $o_1, o_2, \ldots, o_m$. Suppose first that $\frac{k(k+1)}{2} \leq m$. The transactions to be generated will use only the objects $o_1, o_2, \ldots, o_s$, where $s = \frac{k(k+1)}{2}$. Let us take the family of sets $A_1, A_2, \ldots, A_{k+1}$ as in Lemma 1, in which n is set to k. We will use a fixed set of transactions $T_1, T_2, \ldots, T_{k+1}$ defined such that transaction T_i uses object o_j if and only if $j \in A_i$. In particular, each transaction uses k objects. The adversary generates these transactions listed in order $L_0, L_1, L_2, \ldots$, where L_{i-1} is the ith transaction generated and L_i is a transaction identical to $T_{1+i \bmod (k+1)}$. Consider a round $r+1$. Let i be the highest index of a transaction L_i generated by round r. Then in round $r+1$ the adversary generates transactions that make a maximal prefix of the sequence $L_{i+1}, L_{i+2}, \ldots$ such that the total number of transactions generated by round $r+1$ satisfies the constraints on objects' congestion of type (ρ, b). The adversary may generate no transaction at a round and it may generate multiple transaction at a round. For example, the adversary generates exactly the transactions $L_0, \ldots, L_{b-1}$ simultaneously in the first round.

By Lemma 1, at most one transaction can be executed at a round. The $k+1$ transactions $T_1, T_2, \ldots, T_{k+1}$ require $k+1$ rounds to have each one executed, one transaction per round. Discounting for the burstiness of generation, which is possible due to the burstiness component b in the type (ρ, b), these transactions can be generated with a frequency of at most one new transaction generated per round if the execution is to stay stable.

The group of transactions $T_1, \ldots, T_{k+1}$ together contribute 2 to the congestion of each used object, by Lemma 1. If an execution is stable then the inequality $\rho(k+1) \leq 2$ holds. This gives a bound $\rho \leq \frac{2}{k+1}$ on the generation rate of an adversary if the execution is stable. In case $\rho > \frac{2}{k+1}$, the adversary can generate at least one transaction at every round, and for each round r it can generate two transactions at some round after r. Such an execution is unstable, because at most one transaction among $T_1, \ldots, T_{k+1}$ can be executed in one round.

Next, consider the case $\frac{k(k+1)}{2} > m$. Let n be the greatest positive integer such that $\frac{n(n+1)}{2} \leq m$. We use a similar reasoning as in the case $\frac{k(k+1)}{2} \leq m$, with the family of sets $A_1, A_2, \ldots, A_{n+1}$ as in Lemma 1. In particular, we use a set of transactions $T_1, T_2, \ldots, T_{n+1}$ defined such that transaction T_i uses an object o_j if and only if $j \in A_i$. The rules of generating new transactions by the adversary are similar. We obtain the inequality $\rho \leq \frac{2}{n+1}$ by the same argument. The inequality $\frac{n(n+1)}{2} \leq m$ implies $n+1 = \lfloor \frac{1}{2}(1+\sqrt{1+8m}) \rfloor$, by algebra. We have the estimates $\frac{2}{n+1} = \frac{2}{\lfloor \frac{1}{2}(1+\sqrt{1+8m}) \rfloor} \leq \frac{2}{\lfloor \sqrt{2m} \rfloor}$. If $\rho > \frac{2}{\lfloor \sqrt{2m} \rfloor}$ then also $\rho > \frac{2}{n+1}$. It follows that if the adversary is of a type (ρ, b) such that $\rho > \frac{2}{\lfloor \sqrt{2m} \rfloor}$, then this adversary generating transactions at full power can generate at least one transaction at every round, and for each round r it can generate two transactions at some round after r. This makes the queue of transactions grow unbounded.

4 A Centralized Scheduler

We present a scheduling algorithm that processes all transactions pending at a round. The algorithm is centralized in that it is aware of all the pending transactions while selecting the ones to be executed at a round. Throughout this Section we assume the queue-free model of autonomy of individual transactions, and the corresponding queue-free adversarial model of transaction generation.

The centralized scheduler identifies a conflict-free set of transactions pending execution that is maximal with respect to inclusion among all pending transactions. This is accomplished by first ordering all pending transaction on the time of generation and then processing them greedily one by one in this order. The word 'greedily' in this context means passing over a transaction only when its type includes an object that belongs to the type of a transaction already selected for execution at the current round.

The algorithm is called CENTRALIZED-SCHEDULER, its pseudocode is given in Figure 1. The variable `Pending` denotes a list of all pending transactions. At the beginning of a round, all newly generated transactions are appended to the tail of this list. The list is processed in the order from head to tail, which prioritizes transactions on their arrival time, such that those generated earlier get processed before these generated later. The transactions already selected for execution are stored in the set `Execute`. If a transaction in `Pending` is processed, it is checked for conflicts with transactions already placed in the set `Execute`. If a processed transaction does not collide with any transaction already in `Execute` then it is removed from `Pending` and added to `Execute`, and otherwise it is passed over. After the whole list `Pending` have been scanned, all the transactions in `Execute` get executed concurrently. No invoked transaction is aborted in the resulting execution, because conflicts of transactions are avoided by the process to add transactions to the set `Execute`.

To assess the efficiency of executing transactions, let us partition an execution of the algorithm CENTRALIZED-SCHEDULER into contiguous *milestone intervals*

```
Algorithm Centralized-Scheduler

initialize Pending ← an empty list
for each round do
    append all transactions generated in the previous round at the tail of list Pending
    initialize Execute ← an empty set
    if Pending is nonempty then
        repeat
        (a) entry ← first unprocessed list item on Pending, starting from head towards tail
        (b) if entry is conflict-free with all the transactions in Execute then
                remove entry from Pending and add it to set Execute
        until entry points at the tail of list Pending
    execute all the transactions in Execute
```

Fig. 1. A pseudocode of the algorithm scheduling all pending transactions en masse. Transactions pending execution are stored in a list `Pending` in the order of generation, with the oldest at the head. The set `Execute` includes transactions to execute at a round. It is selected in a greedy manner, prioritizing older transactions over newer and avoiding conflicts for access to shared objects.

of rounds, denoted $I_1, I_2, I_3, \ldots$, such that the length of each interval equals $4b \cdot \min\{k, \lceil\sqrt{m}\rceil\}$ rounds.

The following invariant holds for all milestone intervals of an execution.

Lemma 2. *If a generation rate satisfies* $\rho \leq \max\{\frac{1}{4k}, \frac{1}{4\lceil\sqrt{m}\rceil}\}$*, then there are at most* $2bm$ *pending transactions at the first round of a milestone interval, and all these transactions get executed by the end of the interval.*

Algorithm Centralized-Scheduler is stable and has bounded transaction latency for suitably low transaction generation rates.

Theorem 2. *If algorithm* Centralized-Scheduler *is executed against an adversary of type* (ρ, b)*, such that each generated transaction accesses at most* k *objects out of* m *shared objects available and transaction-generation rate* ρ *satisfies* $\rho \leq \max\{\frac{1}{4k}, \frac{1}{4\lceil\sqrt{m}\rceil}\}$*, then the number of pending transactions at a round is at most* $4bm$ *and transaction latency is at most* $8b \cdot \min\{k, \lceil\sqrt{m}\rceil\}$*.*

Proof. To estimate the number of transactions pending at a round, let this round belong to a milestone interval I_k. The number of old transactions at any round of milestone interval I_k is at most $2mb$, by the centralized milestone invariant formulated as Lemma 2. During the interval I_k, at most $2mb$ new transactions can be generated. So $2mb + 2mb = 4mb$ is an upper bound on the number of pending transactions at the round.

To estimate transaction latency, we use the property that a transaction generated in a milestone interval gets executed by the end of the next interval, again by the centralized milestone invariant formulated as Lemma 2. This means that

transaction latency is at most twice the length of a milestone interval, which is $2 \cdot 4b \cdot \min\{k, \lceil\sqrt{m}\rceil\} = 8b \cdot \min\{k, \lceil\sqrt{m}\rceil\}$.

5 A Distributed Scheduler

We now consider distributed scheduling. Let a distributed system consist of n processors. The processors communicate among themselves through some m shared objects by invoking transactions and receiving instantaneous feedback from each involved object. Every transaction type includes at most k objects.

Each generated transaction is assigned to a specific processor and resides in its local queue while pending execution. This means we consider the queue-based model of autonomy of individual transactions, and the corresponding queue-based adversarial model of transaction generation.

We employ a specific communication mechanism between a pair of processors. One of the processors, say s, is a sender and the other processor, say r, is a receiver. The two processors s and r communicate through a designated object o. Communication occurs at a given round. All the processors are aware that this particular round is a round of communication from s to r. Each of the participants s and r may invoke a transaction involving object o at the round, while at the same time all the remaining processors pause and do not invoke any transactions at this round.

Assume first that both s and r have pending transactions that access object o. At a round of communication, the recipient processor r invokes a transaction t_r that uses object o. If the sender processor s wants to convey bit 1 then s also invokes a transaction t_s that uses object o. In this case, both transactions t_r and t_s get aborted, so that the processor r receives the respective feedback from the system and interprets it as receiving 1. If the sender processor s wants to convey bit 0 then s does not invoke any transactions using object o at this round. In this case, transaction t_r gets executed successfully, so that r receives the respective feedback from the system and interprets it as receiving 0. This is how one bit can be transmitted successfully from a sender s to a recipient r.

That was an example of a perfect cooperation between a sender and receiver, but alternative scenarios are possible as well. Suppose that the sender s has a pending transaction using object o and wants to communicate with r but the recipient r either does not want to communicate or does not have a pending transaction using object o. What occurs is that s invokes a suitable transaction t_s which gets executed but r does not receive any information. Alternatively, suppose that the receiver r has a pending transaction using object o and wants to communicate while the sender s either does not want to communicate or does not have a pending transaction using object o. What occurs is that the receiver r invokes a suitable transaction t_r which gets executed, which the receiver r interprets as receiving the bit 0.

That communication mechanism can be extended to transmit the whole type of any transaction in the following way. The type identifies a subset of all m objects. Having a fixed ordering of the objects, the type can be represented as a

sequence of m bits, in which 1 at position i represents that the ith object belongs to the type, and 0 represents that the ith object does not belong. A processor s can transmit a transaction type to recipient r by transmitting m bits representing the type in m successive rounds while using some designated object o. We say that by this operation processor s *sends the transaction type to processor* r *via object* o. This operation works as desired assuming each of the processors has at least m transactions involving object o. If at least one of these processors either does not have m transactions involving object o or does not want to participate, then either no bits are transmitted, or the receiver r possibly receives a sequence of 0s only, which it interprets as no type of transaction successfully transmitted.

Pending transactions at a processor are grouped by their types. All pending transactions of the same type at a processor make a *block of transactions* of this type. The *weight of a block* is defined to be the weight of its type. If there are sufficiently many transactions in a block then the block and the type are said to be *large*. A boundary number defining sizes to be large is denoted by L and equals $L = (n-1)^2n^2m^2$. If the number of transactions of some type in a queue at a processor is at least kL but less than $(k+1)L$, for a positive integer k, then we treat these transactions as contributing k large blocks.

An execution of the scheduling algorithm is partitioned into epochs, and each consecutive epoch consists of three phases, labeled Phase 1, Phase 2, and Phase 3. Each phase is executed the same number of $L = (n-1)^2n^2m^2$ rounds. The algorithm is called Distributed-Scheduler and its pseudocode is given in Figure 2.

In the beginning of Phase 1, each processor v that has a large block of transactions of some type, selects one such a block, and this type then is *active* at the processor in the epoch. A processor that starts Phase 1 with an active type is called *active* in this phase. Processors store large blocks in the order of generation of their last-added transaction. Each processor chooses as active a large block that comes first in this order. The purpose of Phase 1 is to spread the information of active types of all the active processors as widely as possible. Each active processor uses transactions of its active type for communication. Such communication involves executing transactions, so a block of transactions of a given type may gradually get smaller. Once a type of a large block becomes active in the beginning of Phase 1, it stays considered as active for the durations of an epoch, even if the number of transactions in the block becomes less than L. Phase 1 assigns segments of $(n-1)n^2m^2$ rounds for each pair of processors s and r and each object o to spend with s acting as sender to r acting as receiver with communication performed via object o.

Phase 2 is spent on executing transactions in some active blocks selected such that they do not create conflicts for access to shared objects. In the beginning of Phase 2, each processor computes a selection of active large blocks of transactions to execute in Phase 2 among those learned in Phase 1. This common selection is computed greedily as follows. The active types learned in Phase 1 are ordered by the owners' names. There is a working set of active types selected for execution, which is initialized empty. The active types are considered one by one. If a

```
Algorithm DISTRIBUTED-SCHEDULER

Phase 1 : sharing information about large active blocks during L rounds
    repeat n − 1 times
        for each sender processor s and recipient processor r and object o do
            in a segment of rounds assigned for this selection of s, r, and o:
                if v is active and this is a round when s = v then
                    act as sender to transmit all relevant information to r via object o
                elseif v is active and this is a round when r = v then
                    act as recipient to receive information from s via object o
Phase 2 : executing large blocks of transactions during L rounds
    if v is active then
        select active blocks for execution among those learned in Phase 1
    if v is active and its active block got selected then
        for each among L consecutive rounds do
            if there is a transaction of the active type in the queue then
                invoke such a transaction
Phase 3 : executing remaining transactions by solo processors in L rounds
    for L consecutive rounds
        if this is a round among L/n ones assigned to v then
            if the queue is nonempty then invoke a transaction
```

Fig. 2. A pseudocode of an epoch for a processor v. Pending transactions are dispersed among the processors. Number $L = (n-1)^2 n^2 m^2$ is the duration of each phase. In Phase 1, processors s and r use transactions from their active large blocks to implement communication. A sender processor s transmits the active type for each processor it knows about. In Phase 2, large active blocks are selected for execution in a greedy manner, with blocks ordered by the processors' names. In Phase 3, each processor gets assigned a unique exclusive contiguous segment of L/n rounds, in which to execute up to L/n transactions from its queue in a first-in first-out manner.

processed active type can be added to the working set without creating a conflict for access to an object, then the type is added to the set, and otherwise it is passed over. This computation is performed locally by each active processor at the beginning of the first round of Phase 2 and each active processor obtains the same output. The rounds of Phase 2 are spent on executing the transactions of the active blocks selected for execution. An active processor whose active large block has been selected executes pending transactions in its selected active block as long as some transactions from the block are still available in the queue or Phase 2 is over, whichever happens earlier.

Phase 3 is spent by each processor executing solo its pending transactions, those that have never been included in large blocks. Each processor is assigned a unique exclusive contiguous segment of $L/n = (n-1)^2 n m^2$ rounds to execute such transactions. Transactions are performed in the order of their adding to the queue, with those waiting longest executed before those generated later.

Let $P = \sum_{i=1}^{k} \binom{m}{i}$ be the number of possible different transaction types in a system of m shared objects such that a type includes at most k objects.

We will use the estimate $P \leq 2^{\mathrm{H}(\frac{k}{m})\,m}$, for $k \leq \frac{m}{2}$, where $\mathrm{H}(x)$ is the binary entropy function $\mathrm{H}(x) = x \lg x + (1-x)\lg(1-x)$ for $0 < x < 1$.

An execution of algorithm Distributed-Scheduler is partitioned into contiguous *milestone intervals* denoted $I_1, I_2, \ldots$. Each milestone interval consists of $2bnP \cdot \min\{k, \lceil\sqrt{m}\rceil\}$ epochs. Alternatively, a milestone interval consists of $6bnLP \cdot \min\{k, \lceil\sqrt{m}\rceil\}$ rounds, after translating the lengths of epochs into rounds.

The following Lemma 3 gives and invariant that holds for all milestone intervals of an execution of algorithm Distributed-Scheduler.

Lemma 3. *For a generation rate $\rho < \max\{\frac{1}{6k}, \frac{1}{6\lceil\sqrt{m}\rceil}\}$, and assuming the bulk of the system is sufficiently large with respect to ρ, there are at most bn^5m^3P pending transactions at a first round of every milestone interval, and all these transactions get executed by the end of the interval.*

Algorithm Distributed-Scheduler is stable and has bounded transaction latency for suitably low transaction generation rates.

Theorem 3. *If algorithm* Distributed-Scheduler *is executed against an adversary of type (ρ, b), such that each generated transaction accesses at most $k \leq \frac{m}{2}$ objects out of m shared objects available, and the generation rate ρ satisfies $\rho < \max\{\frac{1}{6k}, \frac{1}{6\lceil\sqrt{m}\rceil}\}$, and the bulk of the system is sufficiently large with respect to ρ, then the number of pending transactions at a round is at most $2bn^5m^3\,2^{H(\frac{k}{m})m}$ and latency is at most $12bn^5m^2\,2^{H(\frac{k}{m})m}\min\{k, \lceil\sqrt{m}\rceil\}$.*

Proof. To estimate the number of transactions pending at a round, let this round belong to a milestone interval I_k. The number of old transactions at any round of the interval I_k is at most bn^5m^3P, by the distributed milestone invariant formulated as Lemma 3. During the interval I_k, at most bn^5m^3P new transactions can be generated, again by Lemma 3, because they will become old when the next interval begins. So $2bn^5m^3P \leq 2bn^5m^3\,2^{\mathrm{H}(\frac{k}{m})m}$ is an upper bound on the number of pending transactions at any round, since $P = \sum_{i=1}^{k}\binom{m}{i} \leq 2^{\mathrm{H}(\frac{k}{m})m}$, for $k \leq \frac{m}{2}$.

To estimate the transaction latency, we use the property that a transaction generated in an interval gets executed by the end of the next interval, again by the distributed milestone invariant formulated as Lemma 3. This means that transaction latency is at most twice the length of an interval, which is $2 \cdot 6bnLP\min\{k, \lceil\sqrt{m}\rceil\}$, where $L = (n-1)^2n^2m^2$. We obtain that the latency is at most $12bn^5m^2\,2^{\mathrm{H}(\frac{k}{m})m}\min\{k, \lceil\sqrt{m}\rceil\}$.

6 Conclusion

We propose to study transactional memory systems with continual generation of transactions. The critical measure of quality of such systems is stability understood as having the number of pending transactions bounded from above at all times, for a given generation rate. Transactions are modeled as sets of accesses to shared objects, and it is assumed that conflicting transactions cannot

be executed concurrently. We identify a lower bound on generation rate that makes stability impossible and also develop centralized and distributed optimal scheduling algorithms that handle generation rates asymptotically equal to the lower bound.

The quality of schedulers, on a range of generation rates that guarantee stability, is further assessed by the queue size and latency. The centralized scheduler has these bounds polynomial in the parameters of the system and the adversary's type, but the distributed scheduler has the bounds exponential. It is an open question if it is possible to develop distributed scheduling with polynomial queues and latency for the region of generation rates for which stability is feasible.

Acknowledgements This work was partly supported by the National Science Foundation grant No. 2131538.

References

1. Anantharamu, L., Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Packet latency of deterministic broadcasting in adversarial multiple access channels. Journal of Computer and System Sciences **99**, 27–52 (2019)
2. Andrews, M., Awerbuch, B., Fernández, A., Leighton, F.T., Liu, Z., Kleinberg, J.M.: Universal-stability results and performance bounds for greedy contention-resolution protocols. Journal of the ACM **48**(1), 39–69 (2001)
3. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. Algorithmica **57**(1), 44–61 (2010)
4. Attiya, H., Gramoli, V., Milani, A.: Directory protocols for distributed transactional memory. In: Transactional Memory. Foundations, Algorithms, Tools, and Applications, Lecture Notes in Computer Science, vol. 8913, pp. 367–391. Springer (2015)
5. Attiya, H., Milani, A.: Transactional scheduling for read-dominated workloads. Journal of Parallel and Distributed Computing **72**(10), 1386–1396 (2012)
6. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: Proceedings of the 17th ACM Symposium on Parallel Algorithms and Architectures (SPAA). pp. 325–332 (2005)
7. Borodin, A., Kleinberg, J.M., Raghavan, P., Sudan, M., Williamson, D.P.: Adversarial queuing theory. Journal of the ACM **48**(1), 13–38 (2001)
8. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Time-communication impossibility results for distributed transactional memory. Distributed Computing **31**(6), 471–487 (2018)
9. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Fast scheduling in distributed transactional memory. Theory of Computing Systems **65**(2), 296–322 (2021)
10. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Dynamic scheduling in distributed transactional memory. Distributed Computing **35**(1), 19–36 (2022)
11. Chlebus, B.S., Hradovich, E., Jurdziński, T., Klonowski, M., Kowalski, D.R.: Energy efficient adversarial routing in shared channels. In: Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 191–200. ACM (2019)

12. Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Maximum throughput of multiple access channels in adversarial environments. Distributed Computing **22**(2), 93–116 (2009)
13. Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Adversarial queuing on the multiple access channel. ACM Transactions on Algorithms **8**(1), 5:1–5:31 (2012)
14. Garncarek, P., Jurdzinski, T., Kowalski, D.R.: Stable memoryless queuing under contention. In: Proceedings of the 33rd International Symposium on Distributed Computing (DISC). LIPIcs, vol. 146, pp. 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
15. Garncarek, P., Jurdzinski, T., Kowalski, D.R.: Efficient local medium access. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 247–257. ACM (2020)
16. Hendler, D., Suissa-Peleg, A.: Scheduling-based contention management techniques for transactional memory. In: Transactional Memory. Foundations, Algorithms, Tools, and Applications, Lecture Notes in Computer Science, vol. 8913, pp. 213–227. Springer (2015)
17. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300. ACM (1993)
18. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing **20**(3), 195–208 (2007)
19. Sharma, G., Busch, C.: Distributed transactional memory for general networks. Distributed Computing **27**(5), 329–362 (2014)
20. Sharma, G., Busch, C.: A load balanced directory for distributed shared memory objects. Journal of Parallel and Distributed Computing **78**, 6–24 (2015)
21. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2), 99–116 (1997)
22. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP). pp. 141–150. ACM (2009)
23. Zhang, B., Ravindran, B.: Brief announcement: On enhancing concurrency in distributed transactional memory. In: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC). pp. 73–74. ACM (2010)
24. Zhang, B., Ravindran, B.: Brief announcement: Queuing or priority queuing? On the design of cache-coherence protocols for distributed transactional memory. In: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC). pp. 75–76. ACM (2010)
25. Zhang, B., Ravindran, B., Palmieri, R.: Distributed transactional contention management as the traveling salesman problem. In: Proceedings of the 21st International Colloquium on Structural Information and Communication Complexity (SIROCCO 2014). Lecture Notes in Computer Science, vol. 8576, pp. 54–67. Springer (2014)