Transactional Contention Management as a Non-Clairvoyant Scheduling Problem

Hagit Attiya* Leah Epstein† Hadas Shachnai‡ Tami Tamir§

ABSTRACT

The transactional approach to contention management guarantees atomicity by making sure that whenever two transactions have a conflict on a resource, only one of them proceeds. A major challenge in implementing this approach lies in guaranteeing progress, since transactions are often restarted.

Inspired by the paradigm of *non-clairvoyant* job scheduling, we analyze the performance of a contention manager by comparison with an *optimal*, clairvoyant contention manager that knows the list of resource accesses that will be performed by each transaction, as well as its release time and duration. The realistic, non-clairvoyant contention manager is evaluated by the *competitive ratio* between the last completion time (makespan) it provides and the makespan provided by an optimal contention manager.

Assuming that the amount of exclusive accesses to the resources is non-negligible, we present a simple proof that every work conserving contention manager guaranteeing the pending commit property achieves an $O(s)$ competitive ratio, where s is the number of resources. This bound holds for the GREEDY contention manager studied by Guerraoui et al. [2] and is a significant improvement over the $O(s^2)$ bound they prove for the competitive ratio of GREEDY. We show that this bound is tight for any deterministic contention manager, and under certain assumptions about the transactions, also for randomized contention managers.

When transactions may fail, we show that a simple adaptation of GREEDY has a competitive ratio of at most $O(ks)$, assuming that a transaction may fail at most k times. If a transaction can modify its resource requirements when reinvoked, then any deterministic algorithm has a competitive ratio $\Omega(ks)$. For the case of unit length jobs, we give (almost) matching lower and upper bounds.

*Computer Science Department, The Technion, Haifa 32000, Israel. E-mail: hagit@cs.technion.ac.il.

†Department of Mathematics, University of Haifa, Haifa 31905, Israel. E-mail: lea@math.haifa.ac.il.

‡Computer Science Department, The Technion, Haifa 32000, Israel. E-mail: hadas@cs.technion.ac.il.

§School of Computer Science, The Interdisciplinary Center, Herzliya 46150, Israel. E-mail: tami@idc.ac.il.

PODC'06, July 22-26, 2006, Denver, Colorado, USA.

Categories and Subject Descriptors

D.1.3 [**Software**]: programming techniques—*Concurrent Programming*; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; G.2.3 [**Mathematics of Computing**]: Discrete Mathematics—*Applications*

General Terms

Algorithms

Keywords

scheduling, transactions, software transactional memory, concurrency control, contention management

1. INTRODUCTION

Conventional methods for multi-processor synchronization rely on mutex locks, semaphores and condition variables to manage the contention in accessing shared resources. The perils of these methods are well-known: they are inherently non-scalable and prone to failures. An alternative approach to managing contention is provided by transactional synchronization. As in database systems [12], a *transaction* aggregates a sequence of resource accesses that should be executed atomically by a single thread. A transaction ends either by *committing*, in which case, all of its updates take effect, or by *aborting*, in which case, no update is effective.

The transactional approach to contention management [4] guarantees atomicity by making sure that whenever a conflict occurs, only one of the transactions involved can proceed. A transaction J is in *conflict* when it tries to access a resource R previously modified by some *active* (*pending*) transaction J', that has neither committed nor aborted yet. When this happens, one of the transactions—J or J'—is aborted and its effects are cleared. The aborted transaction is later *restarted* from its very beginning. This guarantees that committed transactions appear to execute sequentially, one after the other, without interference.

A major challenge in implementing a contention manager lies in guaranteeing *progress*. This requires choosing which of

the conflicting transactions (J or J') to abort so as to ensure that work eventually gets done, and all transactions commit. (It is assumed that a transaction that runs without conflicting accesses commits with a correct result; this is guaranteed, for example, by *obstruction free* transactions [4].) Quantitatively, the goal is to maximize the throughput, measured by minimizing the *makespan*—the total time needed to complete a finite set of transactions.

Rather than taking an ad-hoc approach to this problem, we observe that it can naturally be formulated in the parlance of the *non-clairvoyant* job scheduling paradigm, suggested by Motwani et al. [7]. A non-clairvoyant scheduler does not know the characteristics of a job a priori, and is evaluated in comparison with an *optimal*, clairvoyant scheduler that knows all the jobs' characteristics in advance.

We adapt the non-clairvoyant model to our setting, by viewing each transaction as a job and assuming that its resource needs are not known in advance. An optimal contention manager, denoted OPT, knows the accesses that will be performed by each transaction, as well as its release time and duration. The quality of a non-clairvoyant contention manager is measured by the ratio between the makespan it provides and the makespan provided by OPT. This ratio is called the *competitive ratio* of the contention manager.

Under a natural assumption that the amount of exclusive accesses to the resources is non-negligible[1], taking this approach allows us to present a simple and elegant proof that every contention manager with the two following properties achieves an $O(s)$ competitive ratio, where s is the number of resources.

PROPERTY 1. *A contention manager is* work conserving *if it always lets a maximal set of non-conflicting transactions run.*

Note that work conserving contention managers can be efficiently implemented in our model. In general, being work conserving requires to solve the *maximum independent set* (IS) problem, which is NP-hard and hard to approximate. However, in our model, a job that is ready for execution requests a *single* resource in its first action; therefore, the associated conflict graph is a collection of disjoint cliques, on which IS is easily solved by picking one member from each clique.

PROPERTY 2. *A contention manager obeys the* pending commit *property [2] if, at any time, some running transaction will execute uninterrupted until it commits.*

Both properties are guaranteed by the GREEDY contention manager, proposed by Guerraoui et al. [2]: Jobs are processed greedily whenever possible. Thus, a maximal independent set of jobs that are non-conflicting over their first-requested resources are processed each time. When a transaction begins, it is assigned a unique *timestamp* (which remains fixed across re-invocations), so that earlier ("older") transactions have smaller timestamps. Assume transaction J accesses a resource modified by another pending transaction J'; if J is earlier than J' (has smaller timestamp) then J' aborts, otherwise, J waits for J' to complete.[2] (Special accommodation is given to waiting transactions, see [2].) The GREEDY contention manager is decentralized and relies only on local information, carried by the transactions involved in the conflict.

Our result is a significant improvement over the $O(s^2)$ upper bound previously known for GREEDY (see [2]). Simulations [2, 3] show that this contention manager performs well in practice; our analysis indicates that these, and in fact even better, results are expected. We remark that our upper bound for GREEDY allows transactions with arbitrary release times (which are unknown in advance to the contention manager) and arbitrary durations. In contrast, the analysis of Guerraoui et al. relies on the assumption that transactions are available at the beginning of the execution and have equal duration.

We show that our analysis is asymptotically tight, by proving that no work-conserving online contention manager can achieve a better competitive ratio. This lower bound holds even if the contention manager is centralized and does not guarantee the pending commit property, and even if all the transactions have the same duration and are all available at time $t = 0$. For *randomized* contention managers, a lower bound of $\Omega(s)$ holds if transactions can modify their resource needs when they are re-invoked (after being aborted or if they run at a different time).

We also study what happens when transactions may *fail* (not as a result of a conflict). Guerraoui et al. [3] assume that a transaction may fail at most k times, for some $k \geq 1$, and show a contention manager FTGREEDY that has competitive ratio $O(ks^2)$. We improve on their result and show that the competitive ratio is at most $O(ks)$. If a transaction can modify its resource requirement when re-invoked, or if it is run at a later time, then any deterministic algorithm has a competitive ratio $\Omega(ks)$.

Finally, for the special case of unit length jobs, we give (almost) matching lower and upper bounds. We present a randomized algorithm whose competitive ratio is $O(\max\{s, k \log k\})$. This is within logarithmic factor from the lower bound of $\Omega(\max(s, k))$, which holds for *any* (deterministic or randomized) algorithm. The algorithm uses a technique of partition into phases as a function of the number of pending jobs. The probability that a pending job will try to run at a given time increases as the number of jobs in the system drops.

Previous work on non-clairvoyant scheduling assume that the jobs are not available together at the start and that the job's duration is not known when it arrives. In contrast, the optimal scheduler knows the set of jobs, their release times and their duration from the beginning. Motwani et al. [7] allow preemption and assume that a preempted job resumes its execution from where it was stopped; moreover, their schedulers are centralized. In contrast, in our analysis, an aborted job is restarted from its beginning; moreover, we mostly study decentralized contention managers. Edmonds et al. [1] study scheduling of jobs that arrive together, but their characteristics and resource needs change during their execution. Irani and Leung [5] consider decentralized schedulers but assume unit-length jobs that are executed without interruption.

Kalyanasundaram and Pruhs [6] consider the case where the processors (running the jobs) may fail and study the makespan and the average response time of on-line algorithms in comparison with an optimal off-line scheduler.

[1] We formalize this in Section 2.

[2] This resembles classical deadlock prevention schemes [8] (see [11, Ch. 18]).

Their results do not allow preemption, and clearly, do not account for the added cost of re-invocations.

Herlihy et al. [4] suggest a generic implementation of a contention manager. (We follow the description of Scherer and Scott [9], who also evaluate a wide variety of contention managers in [10].) With each resource, we associate the identity of the transaction that most recently modified it.[3] Each transaction has a status field indicating whether it is committed, aborted, or still active. This way, a transaction accessing a resource can easily verify whether it is "locked" by another pending transaction, and decide how to proceed—perhaps using additional data stored for each transaction. All contention managers the fit this generic description are work conserving. Scherer and Scott [9] provide a comprehensive survey of contention managers; more recent work is described in [2, 10].

2. MODEL AND PROBLEM STATEMENT

Consider a set of $n \geq 1$ *transactions* (often called *jobs* below) $J_1, \ldots, J_n$ and a set of $s \geq 1$ shared *resources* $R_1, \ldots, R_s$. Each transaction is a sequence of actions, each of which is an access to a single resource. The transaction starts with an action and may perform local computation (not involving access to resources) between consecutive actions. A transaction completes either with a *commit* or an *abort*. The *duration* of transaction J_i is denoted d_i.

Formally, an *execution* is a finite sequence of *timed actions*. Each action is taken by a single transaction and it is either a *read* to some resource R, a *write* to R, a *commit*, or an *abort*. We assume that the amount of exclusive accesses to the resources performed by $J_1, \ldots, J_n$ is non-negligible, more formally, the total duration of *write* actions is at least $\alpha \sum_{i=1}^{n} d_i$, where $\alpha \in (0, 1]$ is some constant. The times are nonnegative, non-decreasing real numbers. As an example, consider the execution described in Figure 1, $W(R_i)$ denotes write to R_i. Time advances horizontally from left to right.

Note that a transaction may request different resources in different executions. In the above example, when J_3 starts, its first request is for R_3. Later, when J_3 is reinvoked, its first request is for R_1.

A transaction is *pending* after its first action, which must be a read or a write, until its last action, which is a commit or abort; it takes no further actions after a commit or an abort. It is assumed that the times associated with actions of one transaction are increasing, namely, two actions of the same transaction cannot occur at the same time.

For a scheduling algorithm A and a set, $\mathcal{S}$, of jobs, $\mathsf{makespan}(\mathrm{A},\mathcal{S})$ denotes the completion time of all jobs under A, that is the latest time at which any job of A is completed. $\mathcal{S}$ is omitted when the set of jobs is clear from the context. For randomized algorithms we use $\mathsf{makespan}(\mathrm{A},\mathcal{S})$ to denote the expected latest completion time of any job.

We also assume that each transaction may access *different* resources in different invocations. While the online algorithm does not know these accesses until they occur, an optimal offline algorithm, denoted OPT, knows the sequence of accesses of the transaction to resources in each execution.

We make the following simple observation on the decisions of OPT.

[3]The implementation also maintains *before* and *after* information for rolling back an aborted transaction, an issue outside the scope of our paper.

CLAIM 1. *There is an algorithm* OPT *that achieves the minimum makespan and schedules each job exactly once.*

PROOF. Any execution with minimum makespan can be modified so as to remove all partial executions. Clearly, this does not increase the makespan, and provides the above property. □

3. THE GREEDY ALGORITHM HAS $O(S)$-COMPETITIVE MAKESPAN

The greedy algorithm GREEDY, suggested in [2], schedules a maximal independent set of jobs (i.e., jobs that are non-conflicting over their first-requested resources). When a set of jobs is running, and some of these jobs are conflicting over some resource, R_j, GREEDY grants access to the "oldest" job among them, i_o. If i_o needs to perform *write*, then all other jobs are aborted; if it performs *read*, any other "reader" can access R_j too. The algorithm guarantees the pending commit property: at any time in the execution, at least one job (the oldest) is guaranteed to complete its execution without being aborted.

THEOREM 1. GREEDY *is* $O(s)$*-competitive.*

PROOF. Consider the sequence of *idle* time intervals, $I_1, \ldots, I_k$ in which no job is running under GREEDY, and the sequence of time intervals $I'_1, \ldots, I'_\ell$ in which no job is running under OPT. We first prove that there exists an optimal schedule in which the total idle time is at least the total idle time of GREEDY. Formally,

CLAIM 2. $\sum_{j=1}^{k} |I_j| \leq \sum_{j=1}^{\ell} |I'_j|$.

PROOF. By definition, GREEDY is idle at a certain time only after completing all jobs available at that time. Let $I_1 = [t_1, t_2]$; this imply that during time interval $[0, t_1]$, GREEDY is busy processing some set of jobs S. The processing of S is completed at time t_1, and the next job is released at time t_2. There exists an optimal schedule that completes the (sub)instance S at time at most t_1, is idle till t_2, and possibly has additional idle intervals during $[0, t_1]$. Such an optimal schedule exists, since GREEDY completes all jobs in S by time t_1 and no job is available till time t_2. Since we are interested in a schedule which minimizes the makespan, it is even possible to simply adopt the schedule of GREEDY without violating the optimality of the schedule.

Therefore, there exists an optimal schedule with total idle time at least $t_2 - t_1 = |I_1|$ till time t_2. Continuing the same way, for each prefix of idle intervals, we get that for any j, $1 \leq j \leq k$, there exists an optimal schedule with total idle time at least $\sum_{i=1}^{j} |I_i|$ till the end of I_j. In particular, for $j = k$ this gives the statement of the claim. □

By assumption, a job accesses at least one resource at any time during its execution. Consider the set of *write* actions of all transactions. If $s+1$ jobs or more are running concurrently, the pigeonhole principle implies that at least two of them are accessing the same resource. Thus, at least one out of $s+1$ writing jobs will be aborted. Claim 1 implies that no job is aborted in an execution of OPT, implying that at most s writing jobs are running concurrently during time intervals that are not idle under OPT, that is, outside $I'_1, \ldots, I'_\ell$. Thus, the makespan of OPT satisfies:

$$\mathsf{makespan}(\mathrm{OPT}) \geq \sum_{j=1}^{\ell} |I'_j| + \frac{\alpha \sum_{i=1}^{n} d_i}{s}.$$

J_1:	W(R_1)	W(R_2)	Commit					
J_2:	W(R_2)	W(R_1)- Abort		W(R_2)	W(R_1)	Commit		
J_3:	W(R_3)	W(R_2)- Abort		W(R_1)	W(R_2)-Abort	W(R_3)	W(R_2)	Commit

Figure 1: A possible execution.

On the other hand, whenever GREEDY is not idle, at least one of the jobs that are processed will be completed. Hence, the makespan of GREEDY satisfies:

$$\mathsf{makespan}(\text{GREEDY}) \leq \sum_{j=1}^{k} |I_j| + \sum_{i=1}^{n} d_i.$$

The theorem follows. □

We remark that the same proof holds for any work conserving contention manager that guarantees the pending commit property.

4. $\Omega(S)$ LOWER BOUNDS FOR CONTENTION MANAGERS

4.1 A Lower Bound for Fixed First-request

In the following, we give a matching lower bound to the upper bound derived in Section 3 for GREEDY.

THEOREM 2. *Any work-conserving contention deterministic manager is $\Omega(s)$-competitive.*

PROOF. Assume that s is even and denote $s = 2k$.

The proof uses an execution with $ks = s^2/2$ unit length jobs, described in Table 1: Each job j requests a pair of resources (R_{j_1}, R_{j_2}), such that R_{j_1} is the resource required to begin the transaction, and R_{j_2} is an additional resource requested by the job in order to complete its execution and is not known in advance (the table shows the indices (j_1, j_2)). All jobs are released and available at time $t = 0$. An online algorithm knows only the first resource request of each job, therefore, the input is in fact a set of ks jobs, such that for every resource i, $1 \leq i \leq s$, exactly $k = s/2$ jobs request R_i for their execution to start. The second resource in each pair will be determined by the adversary during the execution of the algorithm in a way that will force many of the jobs to abort.

Consider a work-conserving contention manager. Being work-conserving, it must select to process a set of s non-conflicting jobs, each requesting a different resource as its first requested resource. The adversary will then determine the second resource of each of these jobs according to a single column in Table 1. Specifically, the first phase of s jobs is described by the first column of the Table, that is, in order to complete their execution, jobs whose first access is to a resource with an odd index, request at time $1-\varepsilon$ the resource R_2 as their second resource, while jobs whose first access is to a resource with an even index, request at time $1-\varepsilon$ the resource R_1 as their second resource. Clearly, at most two of these jobs can complete their execution, while all other $s-2$ jobs must abort.

In general, in phase t, the algorithm selects an independent set of s jobs, and the adversary determines their second requested resource at time $t+1-\varepsilon$ to be either R_{2t-1} or R_{2t}, as described by column t of the table. Once again, only two jobs from this column can complete their execution, while all other jobs must abort.

All aborting jobs request R_1 in any subsequent execution. This implies that after the first k time-slots, during which at most two jobs run in parallel, no parallelism is possible. The resulting makespan is therefore $ks - 2k = (s-2)s/2$.

We now show that there exists an optimal schedule with makespan s: Note that each diagonal directed from left to right in the upper or the lower half of the table consists of k independent jobs that require exactly all the resources. Formally, for every odd value $z \in \{1, 3, \ldots, 2k-1\}$, $I_z(up)$ is the set of jobs in the upper half for which (R_{j_1}, R_{j_2}) have the form $(r, (r+z) \ mod \ 2k)$, where $r = 2\ell + 1$, $0 \leq \ell \leq k-1$. The sets in the lower half, $I_z(low)$, are similarly defined, only that $r = 2\ell$, $1 \leq \ell \leq k$. For example $I_1(up) = \{(1,2), (3,4), \ldots (2k-1, 2k)\}$ and $I_{2k-5}(low) = \{(6,1), (8,3), \ldots, (4, 2k-1)\}$.

An optimal contention manager runs all k jobs forming each of these sets simultaneously; the makespan of this schedule is the number of sets, that is, $2k = s$, implying a competitive ratio of $(s-2)/2$. □

4.2 A Lower Bound for Variable First-request

Consider now a generalized model in which, as before, any job j arriving to the system requests its first resource; however, while waiting to start its execution, j may modify the request to another resource, thus the online algorithm knows the first resource requested by any job only when this job starts running. For this model, we show a lower bound of $\Omega(s)$ for *any* (deterministic or randomized) algorithm.

THEOREM 3. *Any randomized algorithm in the model where the first request for any resource is time dependent has competitive ratio of $\Omega(s)$.*

PROOF. We first prove a lower bound of $\Omega(s)$ for an arbitrary online deterministic algorithm, and then show how to adapt it to randomized algorithms. Let $s' = \lfloor s/2 \rfloor$.

In our execution, each job will have a single request for a resource. It reveals the information regarding the resources it is going to need each time it restarts. Thus, the resource requests are time dependent.

At first, there are $2s'$ sets of unit-length jobs $A_1, ..., A_{2s'}$. Each set contains $2s'$ jobs, where all jobs in one set A_i initially request resource i. For some of the jobs this is changed later on. Consider the situation after s' time units.

We define an offline contention manager OFF. Partition each set A_i into B_i and C_i. The set C_i contains all jobs in A_i that the algorithm completes by time s'. Add additional jobs from A_i to C_i until $|C_i| = s'$. Let $B_i = A_i - C_i$. OFF runs the jobs of B_i during time units $1, 2, ..., s'$. Since each set B_i requested a different resource, at time s', OFF completes all these jobs. However, the online algorithm did not run any of these jobs yet. Starting at time s', the jobs in $B_2, B_3, ..., B_s$ request resource R_1 when they start running. Thus, all waiting requests need to use the same resource, and the on-line algorithm needs $2s'^2$ additional time units

	1	2	3	...	k
1	$(1,2)$	$(1,4)$	$(1,6)$	...	$(1,2k)$
2	$(3,2)$	$(3,4)$	$(3,6)$	...	$(3,2k)$
3	$(5,2)$	$(5,4)$	$(5,6)$	...	$(5,2k)$
.	.	.	.	...	.
.	.	.	.	...	.
k	$(2k-1,2)$	$(2k-1,4)$	$(2k-1,6)$	...	$(2k-1,2k)$
1	$(2,1)$	$(2,3)$	$(2,5)$	...	$(2,2k-1)$
2	$(4,1)$	$(4,3)$	$(4,5)$	...	$(4,2k-1)$
3	$(6,1)$	$(6,3)$	$(6,5)$	...	$(6,2k-1)$
.	.	.	.	...	.
.	.	.	.	...	.
k	$(2k,1)$	$(2k,3)$	$(2k,5)$	...	$(2k,2k-1)$

Table 1: The set of jobs used in the proof of Theorem 2.

to complete them. In contrast, OFF needs only s' additional time units, since it can now run the C_i jobs for all i in s' time units in parallel. We get that the algorithm completes all jobs at time $s'+2s'^2$, whereas OFF completes all jobs by time $2s'$. This gives a lower bound of $\frac{1}{2}+s' = \Omega(s)$.

Assume now that the algorithm is randomized. Instead of defining C_i as before, let C_i be the set of s' elements of A_i with the highest probability to be run by the algorithm and complete by time s'. Let $B_i = A_i - C_i$, i.e., the jobs with smallest probabilities to terminate successfully by time s'. As in the deterministic case, OFF runs all jobs of all B_i until time s' and afterwords all jobs of C_i. Also, all jobs of B_i request only resource 1 if they are run starting from time s' or later.

Let X_i (respectively Y_i) be the number of elements of B_i (C_i) that have been completed by the algorithm by time s'. It holds that $E(X_i) \leq \frac{s'}{2}$. To see this, we use the linearity of expectation and get $E(X_i) \leq E(Y_i)$ and since $X_i + Y_i \leq s'$ we have $E(X_i)+E(Y_i) \leq s'$. Thus, the expected number of elements from all B_i's that are still waiting to be scheduled by the algorithm is at least $\frac{2s'^2}{2} = s'^2$. It follows that the expected makespan is at least $s' + s'^2$, and we get a lower bound of $\Omega(s)$ for the randomized case as well. □

Note that the proof of this theorem uses $O(s)$ jobs, while the proof of Theorem 2 uses $O(s^2)$ jobs.

We remark that GREEDY is s-competitive also in this generalized model. The proof of Theorem 1 makes no assumption on the identities of the requested resources, i.e., a job may modify its resource request as long as it has not started running; also, if a job was aborted and is waiting to be restarted, it may initially ask for some resource, and later modify its request.

5. HANDLING FAILURES

Consider a system in which jobs may fail; if a job j running at time t fails, the contention manager subsequently needs to restart the execution of j [3]. We assume that at most k failures may occur for any job, for some $k \geq 1$. Indeed, for any job j, GREEDY may run j almost to completion k times, and then restart its execution due to a failure. This stretches the processing time of j to $(k+1)d_j$. In contrast, an optimal offline algorithm may avoid the execution of a job j when j may fail. This implies:

THEOREM 4. *If each job may fail at most k times, then* GREEDY *is $O(ks)$-competitive.*

For this model, we show a lower bound of $\Omega(ks)$ for *any* deterministic algorithm.

THEOREM 5. *Assume that the first request of a job for a resource is time dependent, and each job may fail at most k times, for some $k \geq 1$, then any deterministic algorithm has competitive ratio $\Omega(ks)$.*

PROOF. Define the sets A_i, B_i and C_i as in the proof of Theorem 3. The sequence is the same until time $2s'$ ($= 2\lfloor s/2 \rfloor$) at which OFF completes all jobs. After this time, we define failure times as follows. Consider the schedule of the algorithm. If a running job already failed k times then it is not interrupted; otherwise, it fails just before completion. Thus, all jobs except for at most $2s'^2+s'$ fail exactly k times. Since the failure of any job occurs almost upon completion, the remaining $2s'^2 - s'$ jobs are completed only after $(k+1)(2s'^2 - s')$ additional times units. We get a total of $(k+1)(2s'^2-s')+2s'$ time slots, and a lower bound of $\Omega(ks)$. □

When requests are time dependent, we obtain a lower bound also for randomized algorithms.

THEOREM 6. *Assume that the first request of a job for a resource is time dependent, and each job may fail at most k times, for some $k \geq 1$, then any (deterministic or randomized) algorithm has competitive ratio $\Omega(\max\{s,k\})$.*

PROOF. Assume that $k \geq 5$, otherwise the deterministic bounds can be applied. A lower bound of $\Omega(s)$ follows from Theorem 3. To prove a lower bound of k consider an input with two jobs j_1 and j_2, each having (a different) one of the two sets of failure times: $\{1, \frac{3}{2}, 2, \frac{5}{2}, \ldots, \frac{k+1}{2}\}$ and $\{\frac{1}{2}, 1, 2, \frac{5}{2}, \ldots, \frac{k+1}{2}\}$, that is, both sets contain all multiples of $\frac{1}{2}$ (up to and including $\frac{k+1}{2}$) except one such number. The first set does not contain $\frac{1}{2}$ whereas the second one does not contain $\frac{3}{2}$. Assume that $s = 1$, thus, the issue of resources may be ignored. An offline algorithm can run the job with

the first failure times sequence at time 0, until time 1, and the other job at time 1, until time 2.

Consider an online algorithm. Let p_1 be the probability that job j_1 is running just before time $\frac{1}{2}$ and p_2 that j_2 is running. We have $p_1 + p_2 \leq 1$ (since it may be the case that no job is running). If $p_1 \leq p_2$, we assign the first failure times sequence to j_1 and the second one to j_2, and otherwise we do the opposite assignment. The only way that all jobs are completed by time 2, is that some job is completed by time 1, and thus this job needs to be running just before time $\frac{1}{2}$, and not interrupted at time $\frac{1}{2}$. The probability for that is p_1 in the first case and p_2 in the second case. However, in the first case $p_1 \leq \frac{1}{2}$ and in the second case $p_2 \leq \frac{1}{2}$, so with probability at least $\frac{1}{2}$, at least one job can run to completion only after time $\frac{k+1}{2}$. Thus, the expected completion time is at least $\frac{1}{2} \cdot 2 + \frac{1}{2} \cdot (\frac{k+1}{2} + 1) = \Omega(k)$. □

We next describe a randomized algorithm that matches this bound within a logarithmic factor for the case where all jobs require unit processing time. We start with a description of a centralized scheduler, and later explain how to make it decentralized.

Algorithm PHASES

Let N be the set of pending jobs, and $|N|$ be its size. Initially, N is the set of all jobs.

Phase 1. While $|N| > 2k$ repeat the following steps.

Choose randomly and uniformly a permutation of the n jobs, and assign the jobs in this order to run (one job at a time) in the next n time units. The algorithm is oblivious to aborts of jobs, and keeps the schedule unchanged even if it becomes idle. Update N.

Phase 2. For $j = 1, \ldots, \lceil 3 \log_2 k \rceil$ repeat the following steps.

Choose randomly and uniformly an assignment of the pending jobs, to the $2k$ time slots (such that each job receives one random time slot among the $2k$ slots, and some slots possibly remain idle). Assign jobs to run at most one at a time, according to the assignment, in the next $2k$ time units. Update N.

Phase 3. While $|N| > 0$ repeat the following steps.

Select a pending job from N and schedule it at every integer time point until it runs to completion. Update N.

Even though the algorithm is randomized, its worst case total running time is bounded: Phase 1 terminates after at most $k + 1$ iterations, since each job can be interrupted at most k times. The same holds for Phase 2 and Phase 3. Thus, in the worst case, the algorithm completes all jobs after $O(nk + k^2)$ time units.

Next, we analyze the expected running time of the algorithm.

THEOREM 7. *The competitive ratio of* PHASES *is at most* $O(\max\{s, k \log k\})$.

PROOF. Our proof consists of examining the expected duration of each of the three phases. We show that the first phase consumes expected time of $O(n)$ and the second and third phases consume expected time $O(k \log k)$. Since OPT $\geq \max\{1, \frac{\alpha n}{s}\}$, this would give the competitive ratio as claimed. Note that if n is initially small, it may be the case that Phase 1 is skipped, or the other phases are skipped. Moreover, it may be the case that Phase 2 or Phase 3 are skipped, since the number of pending jobs can drop quickly in an iteration of a previous phase.

Let n_i be the number of pending jobs when Phase i starts. Phase 3 lasts at most $k + 1$ times units for every job, and thus takes at most $n_3(k + 1)$ time units.

Consider Phase 2. Since $n_2 \leq 2k$, each iteration admits an assignment of all jobs to time slots. Consider a specific job scheduled in an iteration. This job may be assigned to any of the $2k$ time slots starting at integer times with equal probability. However, out of these slots, at most k can prevent a successful completion of the job. Thus, the job is completed in a given iteration with probability at least $\frac{1}{2}$. We next compute an upper bound on the probability that the algorithm reaches Phase 3. The probability of a given job to be pending, even after $\lceil 3 \log_2 k \rceil$ iterations of Phase 2, is at most $(\frac{1}{2})^{\lceil 3 \log_2 k \rceil} \leq \frac{1}{k^3}$. Using the sum of probabilities as an upper bound, the probability that at least one job is left for Phase 3 is upper bounded by $\frac{n_2}{k^3} \leq \frac{2k}{k^3} = \frac{2}{k^2}$. Thus, with probability at most $1 - \frac{2}{k^2}$, the algorithm does not reach Phase 3, and thus, the overall running time for Phases 2 and 3 is at most $2k \cdot \lceil 3 \log_2 k \rceil$. With probability at most $\frac{2}{k^2}$, Phase 3 will require at most $(k+1)n_3 \leq 2k(k+1)$ additional time units. This gives expected additional time of at most $\frac{4(k+1)}{k} < 5$. We get for Phases 2 and 3 an expected total running time of $O(k \log k)$.

Finally, consider Phase 1. Let X_i be a random variable which denotes the length of iteration i of this phase. Clearly $X_1 = n$. We claim that $E(X_i) \leq \frac{X_{i-1}}{2}$ for $i \geq 2$. Similarly to Phase 2, each job has equal probability to be assigned to each time slot, and since $n > 2k$ during this phase, the probability of a job to run to completion during iteration $i-1$ is at least $\frac{1}{2}$. Since this holds for any value of X_i, and due to linearity of expectation, we conclude that $E(X_i) \leq \frac{E(X_{i-1})}{2}$. Using induction we can prove that $E(X_i) \leq \frac{1}{2^{i-1}} n_1$. Let t be the number of iterations in Phase 1, as we saw above, this number is no larger than $k + 1$. The length of Phase 1 is then at most $\sum_{i=1}^{t} E(X_i) \leq 2n$. This completes the proof. □

A Decentralized Implementation of PHASES

We describe a decentralized implementation of Algorithm PHASES, assuming a synchronized system. Crucial to the algorithm is the assumption that pending jobs are aware of $|N|$ (the number of pending jobs), at the beginning of each iteration of Phase 1, and at the end of each of the first two phases. (This can be achieved by collecting global information.) Initially, $|N| = n$.

As before, Phase 1 ends when fewer than $2k$ jobs remain. The length of each iteration $i \geq 1$ in this phase is equal to the number of remaining jobs at the beginning of this iteration, denoted m_i. In iteration i of Phase 1, any job which has not completed and did not fail yet in this iteration, runs in the next time slot with probability $\frac{1}{m_i}$. If more than one job run in some slot, and the jobs are conflicting over resources, then one of the jobs in the set is selected to run, randomly and uniformly. The set N is updated at the end of each iteration (jobs need only know its size).

Jobs follow Phase 2 in a similar manner, except that the

length of each iteration in this phase is $2k$, and each of the remaining jobs runs in the next slot (in any iteration) with probability $\frac{1}{2k}$; the number of iterations in Phase 2 is y, that will be determined later.

In Phase 3, jobs start running in time slots that are integral multiples of $k+1$. Each of the remaining jobs starts running in the next scheduling point. If several jobs have a conflict on some resource, then the oldest job wins the next $k+1$ time slots, while all other jobs need to restart.

We next analyze the algorithm and show that it is a decentralized implementation of algorithm PHASES, where the expected running time increases by a constant factor. Specifically, we show that the expected length of Phase 1 is $O(n)$, while the expected length of Phases 2 and 3 is $O(k \log k)$.

Consider Phase 1. Suppose that some job ℓ tries to run in slot j of iteration i. The probability that no other job attempts to run in this slot is at least

$$(1 - \frac{1}{m_i})^{m_i - 1} \geq \frac{m_i}{m_i - 1} e^{-2} \geq e^{-2}.$$

If job ℓ runs alone in some slot in iteration i and does not fail, then ℓ completes in this iteration. To lower bound the probability that job ℓ completes in iteration i, let $Good_i$ denote the set of (at least) $m_i - k$ time slots that are good for ℓ in iteration i, i.e., if ℓ runs in any of these slots then it does not fail. Also, let A_j^i be the event "In iteration i, job ℓ runs for the first time in slot j, and conflicts with no other job in this slot," then the probability that ℓ completes in iteration i is at least

$$\begin{aligned}
\sum_{j \in Good_i} Prob(A_j^i) &\geq \sum_{j \in Good_i} (1 - \frac{1}{m_i})^{j-1} \cdot \frac{1}{m_i} \cdot \frac{1}{e^2} \\
&\geq \sum_{j=k+1}^{m_i} (1 - \frac{1}{m_i})^{j-1} \frac{1}{e^2 m_i} \\
&= (1 - \frac{1}{m_i})^k \frac{1 - (1 - \frac{1}{m_i})^{m_i - k}}{e^2} \\
&\geq (1 - \frac{1}{m_i})^{m_i/2} \frac{1}{e^2} (1 - (1 - \frac{1}{m_i})^{m_i/2}) \\
&\quad \text{since } k \leq m_i/2 \\
&\geq \frac{1}{e^3}(1 - \frac{1}{\sqrt{e}}) \\
&\quad \text{since } e^{-1} \leq (1 - \tfrac{1}{m_i})^{m_i/2} \leq e^{-1/2}
\end{aligned}$$

Letting $\delta = e^{-3}(1 - 1/\sqrt{e})$, we get that

$$E[X_i] = (1 - \delta) E[X_{i-1}] \ ,$$

where X_i is a random variable denoting the length of iteration i of Phase 1 (as in the analysis of Phase 1 in algorithm PHASES). It follows that the expected length of Phase 1 is $\sum_{i \geq 1} (1 - \delta)^{i-1} n = \frac{n}{\delta}$.

For Phase 2, we set the number of iterations to be

$$y = \log(2(k+1)/\log k)/\log(1/(1-\delta)),$$

and get that its length is $O(k \log k)$.

For Phase 3, an analysis similar to Phase 1 shows that the probability that a job that started Phase 2 does not complete by the end of the phase is at most $(1 - \delta)^y < \frac{\log k}{2(k+1)}$. Therefore, the expected length of Phase 3 is at most $(1-\delta)^y \cdot 2k(k+1) = O(k \log k)$. This completes the analysis.

In the decentralized implementation of PHASES, the worst case length of Phase 1 is unbounded. The following adaptation of the algorithm results in bounding the length of phase 1 with $O(nk)$. When the phase reaches iteration $z = \log(k/2)/\log(1/1-\delta)$, *every* remaining jobs starts running in the next time slot. Conflicts are resolved as before, by random selection of one job in the conflict set. Clearly, this implies that in the next $(k+1)n$ time units all jobs complete. It can be shown that the expected length of Phase 1 with this modification remains $O(n)$. (We leave the details to the full version of the paper.) The lengths of the other two phases are bounded.

6. DISCUSSION

We adopted terminology and techniques of non-clairvoyant scheduling to analyze the behavior of transactional contention managers. Our framework allows to explore further extensions to the results presented here, e.g., to prove bounds when the amount of exclusive accesses to the resources is negligible, in particular, when there are many read-only jobs.

It would be nice to remove the assumption that jobs can modify their resource needs (made in Theorem 3 and Theorem 7).

Another problem that remains open is the optimality of work-conservative algorithms. The lower bound of $\Omega(s)$ presented in Theorem 2 is suitable only for work-conserving contention managers. Our difficulty in obtaining the same lower bound for non work-conserving algorithms may hint that such algorithms can perform better. Note that a lower bound of $\Omega(\sqrt{s})$ for non work-conserving algorithms follows from the construction in the proof of Theorem 2.

The analysis of Algorithm PHASES hinges on the fact that the probability of a job trying to execute in a phase depends on the number of pending jobs. Scherer and Scott [9] describe a practical *randomized* contention manager that flips a coin to choose between aborting the other transaction and waiting for a random time. Our analysis suggests that in order for their contention manager to be effective, it should bias the coin in a way that depends on (at least) an estimate of the number of jobs waiting to be executed.

Another interesting avenue for further research is to evaluate other complexity measures, in particular, those that evaluate the guarantees provided for each individual transaction, like the *average response or waiting time* or the *average punishment*.

Acknowledgments. We would like to thank Rachid Guerraoui and Bastian Pochon for helpful discussions, and the anonymous referees for their comments.

7. REFERENCES

[1] Jeff Edmonds, Donald D. Chinn, Tim Brecht and Xiaotie Deng, Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *J. Scheduling*, 6(3): 231-250 (2003).

[2] R. Guerraoui, M. Herlihy and S. Pochon, Toward a Theory of Transactional Contention Management. PODC 2005: 258-264.

[3] R. Guerraoui, M. Herlihy, M. Kapalka and S. Pochon, Robust Contention Management in software transactional memory. Synchronization and

Concurrency in Object-Oriented Languages (SCOOL) workshop, in conjunction with OOPSLA 2005. *http://urresearch.rochester.edu/handle/1802/2103*.

[4] Maurice Herlihy, Victor Luchangco, Mark Moir and William N. Scherer III, Software transactional memory for dynamic-sized data structures. PODC 2003: 92-101.

[5] Sandy Irani and Vitus Leung, Scheduling with Conflicts, and Applications to Traffic Signal Control. SODA 1996: 85-94.

[6] Bala Kalyanasundaram and Kirk R. Pruhs, Fault-tolerant scheduling. *SIAM Journal on Computing*, 34(3): 697 - 719 (2005).

[7] R. Motwani, S. Phillips and E. Torng, Non-Clairvoyant Scheduling. *Theor. Comput. Sci*, 130(1): 17–47 1994.

[8] Daniel J. Rosenkrantz, Richard E. Stearns and Philip M. Lewis II, System Level Concurrency Control for Distributed Database Systems. *ACM Trans. Database Syst.*, 3(2): 178-198 (1978).

[9] William N. Scherer III and Michael Scott, Contention Management in Dynamic Software Transactional Memory. PODC Workshop on Concurrency and Synchronization in Java Programs, 2004: 70-79.

[10] William N. Scherer III and Michael Scott, Advanced Contention Management for Dynamic Software Transactional Memory, PODC 2005: 240-248.

[11] Abraham Silberschatz and Peter Galvin, *Operating Systems Concepts*, 5th edition, John Wiley & sons, 1999.

[12] Gottfried Vossen and Gerhard Weikum, *Transactional Information Systems*, Morgan Kaufmann, 2001.