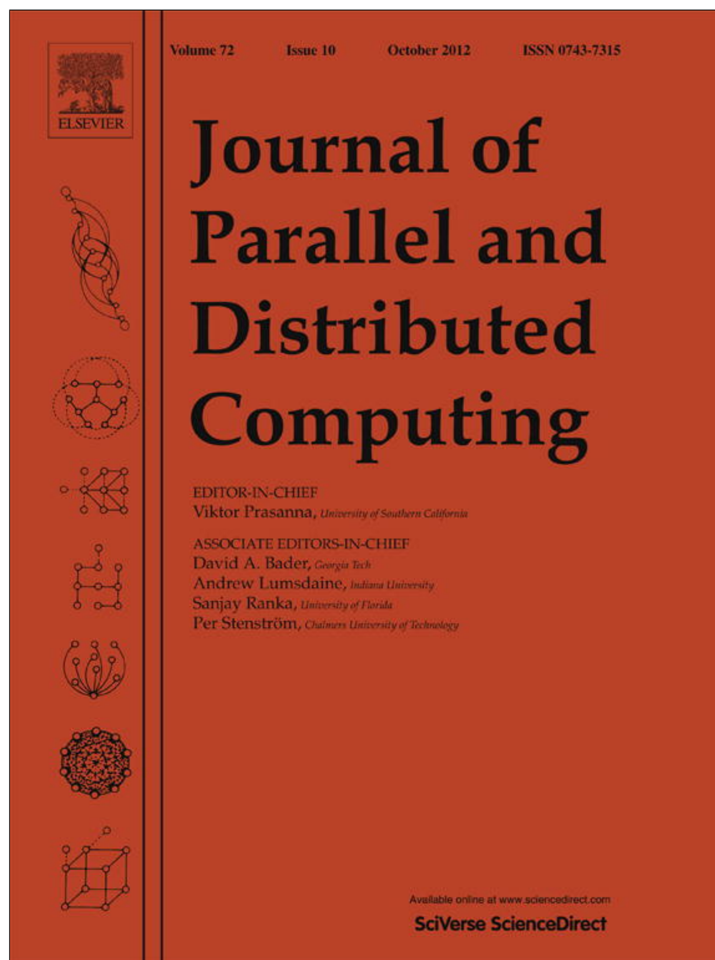
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

J. Parallel Distrib. Comput. 72 (2012) 1386–1396

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Transactional scheduling for read-dominated workloads[☆]

Hagit Attiya [a,*], Alessia Milani [b,1]

[a] Technion, Israel
[b] Labri, University of Bordeaux 1-IPB, France

ARTICLE INFO

Article history:
Received 8 December 2011
Received in revised form
5 May 2012
Accepted 29 May 2012
Available online 15 June 2012

Keywords:
Transactional memory
Scheduling
Competitive analysis
Read-dominated workload

ABSTRACT

The transactional approach to contention management guarantees atomicity by aborting transactions that may violate consistency. A major challenge in this approach is to schedule transactions in a manner that reduces the total time to perform all transactions (the *makespan*), since transactions are often aborted and restarted. The performance of a transactional scheduler can be evaluated by the ratio between its makespan and the makespan of an optimal, clairvoyant scheduler that knows the list of resource accesses that will be performed by each transaction, as well as its release time and duration.

This paper studies transactional scheduling in the context of read-dominated workloads; these common workloads include *read-only* transactions, i.e., those that only observe data, and *late-write* transactions, i.e., those that update only towards the end of the transaction.

We present the Bimodal transactional scheduler, which is especially tailored to accommodate read-only transactions, without punishing transactions that write most of their duration (*early-write* transactions). It is evaluated by comparison with an optimal clairvoyant scheduler; we prove that Bimodal demonstrates the best competitive ratio achievable by a non-clairvoyant schedule for workloads consisting of early-write and read-only transactions.

We also show that late-write transactions significantly deteriorate the competitive ratio of any non-clairvoyant scheduler, assuming it takes a conservative approach to conflicts.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

A promising approach to programming concurrent applications is provided by *transactional synchronization*: 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. If aborted, a transaction is later *restarted* from its beginning.

Several existing transactional memory implementations (e.g., [16,6]) guarantee consistency by making sure that whenever there is a conflict, that is, two transactions access the same resource and at least one writes to it, one of the transactions involved is aborted. We call this approach *conservative*. Taking a non-conservative approach and ensuring progress while accurately avoiding consistency violations, seems to require complex data structures, as used for example in [5,20].

A major challenge is guaranteeing *progress* through a *transactional scheduler*, choosing which transaction to delay or abort and when to restart the aborted transaction, so as to ensure that work eventually gets done, and all transactions commit.[2] This goal can also be stated quantitatively as minimizing the *makespan*—the total time needed to complete a finite set of transactions. Clearly, the makespan depends on the *workload*—the set of transactions and their characteristics, for example, their release times, duration, and most importantly, the resources they read or modify.

The *competitive* approach for evaluating a transactional scheduler A calculates the *ratio* between the makespan provided by A and by an optimal, clairvoyant scheduler, for each workload separately, and then finds the maximal ratio [3,9,11]. It has been shown that the best competitive ratio achieved by simple transactional schedulers is $\Theta(s)$, where s is the number of resources [3]. These prior studies assumed *write-dominated* workloads, in which transactions need exclusive access to resources for most of their duration.

☆ A preliminary version of this paper appeared in OPODIS 2009 [4]. This research is partially supported by the Israel Science Foundation (grants number 953/06 and 1227/10) and by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TRANSFORM.

* Corresponding author.
E-mail addresses: hagit@cs.technion.ac.il (H. Attiya), milani@labri.fr (A. Milani).

1 Part of this work was performed while second author was at the Department of Computer Science, Technion, Israel, supported by the Lady Davis Foundation and grant Progetto FIRB Italia-IsraeleRBIN047MH9, and at the Université Pierre et Marie Curie—Paris 6, LIP6-CNRS UMR 7606, France.

2 It is typically assumed that a transaction running solo, without conflicting accesses, commits with a correct result [16].

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.05.012

Author's personal copy

In this paper, we extend the study of transactional schedulers without a priori knowledge of the workload, by considering *read-dominated* workloads, which are common in transactional memory [14]. In such workloads, transactions do not need exclusive access to resources for most of their duration. This includes *read-only* transactions that only observe data and do not modify it, as well as *late-write* transactions (that perform the first write operation towards the end of their execution), e.g., locating an item by searching a list and then inserting or deleting. In particular, we extend the result in [3] by proving that every deterministic scheduler is $\Omega(s)$-competitive for read-dominated workloads, where s is the number of resources.

Then, we prove that any conservative non-clairvoyant scheduler is $\Omega(m)$-competitive for some workload containing late-write transactions, where m is the number of cores. This means that, for some workloads, conservative schedulers utilize at most one core, while an optimal, clairvoyant scheduler exploits the maximal parallelism on all m cores. This bound can be easily shown to be tight, since at each time, a reasonable transactional scheduler typically makes progress on at least one transaction.

Several transactional schedulers, like CAR-STM [7], Adaptive Transaction Scheduling [29], and Steal-On-Abort [2], try to avoid repeated conflicts and reduce wasted work, without deteriorating throughput. Using somewhat different mechanisms, these schedulers avoid repeated aborts by *serializing* transactions after a conflict happens, and they end up serializing also read-only transactions.

We study *bimodal* workloads containing only early-write transactions (that perform a write operation at the beginning of their execution) and read-only transactions, and show that there are bimodal workloads for which these transactional schedulers are, at best, $\Omega(m)$-competitive.

These counter-examples motivate our Bimodal scheduler, which is $O(s)$-competitive for bimodal workloads with almost equi-duration transactions. Bimodal alternates between *writing epochs* in which it gives priority to writing transactions, and *reading epochs* in which it prioritizes transactions that have issued only reads so far. Bimodal has optimal competitive makespan, since no scheduler can do better than $O(s)$ for bimodal workloads [3]. Bimodal also works when the workload is not bimodal, but it can only be trivially bounded to be $O(m)$-competitive for workloads with late-write transactions.

The rest of the paper is organized as follows. In Section 2, we state the model and present simple preliminary bounds. Section 3 presents the lower bounds, while Section 4 discusses the competitive ratio of recent transactional schedulers. In Section 5, we describe the Bimodal scheduler and analyze its performance. We conclude with a discussion of other related work (Section 6), and a summary (Section 7).

2. Preliminaries

2.1. Model

We consider a system of m identical *cores* with s shared data items $\{i_1, \ldots, i_s\}$, which should execute a *workload* of transactions $\Gamma = \{T_1, T_2, \ldots, T_n\}$. A transaction is a sequence of operations on the shared data items; we restrict attention to *read* and *write* operations.

A transaction T is *pending* after its first operation, and before T completes either by a *commit* or an *abort* operation. Once a transaction aborts, it is restarted from its very beginning and can possibly access a different set of data items. Generally, a transaction may access different data items if it executes at different times. For example, a transaction inserting an item at the end of a linked list, may access different elements of the list, when traversing the list at different times.

The sequence of operations in a transaction must appear to occur atomically: if any of the operations takes place, they all do, and if they do, they appear to other threads to do so atomically, in the order specified by the transaction. Formally, this is captured by a classical consistency condition like *serializability* [21] or the stronger *opacity* condition [12].

The collection of items accessed by a transaction is the transaction's *data set*; the items modified by the transaction are its *write set*, and the items read by the transaction are its *read set*. A transaction is said to be *read-only* if its write set is empty; otherwise, it is a *writing* transaction.

The data set of a transaction is not known when the transaction starts, except for the first data item that is accessed. At each point, the scheduler must decide what to do, knowing only the data item currently requested and if the access is going to modify the data item or just read it.

Two transactions have a (nontrivial) *conflict* if the write set of one transaction intersects the data set of the other transaction; i.e., either they both write to the same item or one of them writes and the other reads the item. Note that a conflict does not mean that serializability is violated. For example, two overlapping transactions $[read(X), write(Y)]$ and $[write(X), read(Z)]$ can be serialized, despite having a conflict on X.

Each transaction has a release time, when it arrives at the system.

The *duration* of a transaction T_i is a real number $\tau_i > 0$, which is the execution time of T_i when it runs uninterrupted to completion. The time from the first write operation of T_i to completion is $\omega_i = \alpha_i \tau_i$, for some constant $\alpha_i,\ 0 \le \alpha_i \le 1$; ρ_i is the time until the first write operation, that is, $\rho_i = \tau_i - \omega_i$. For a read-only transaction, $\omega_i = \alpha_i = 0$ and $\tau_i = \rho_i$.

If ω_i is negligible relative to the duration of transaction T_i, i.e., α_i is close to 0, we say that T_i is a *late-write* transaction. If α_i is close to 1, we say that T_i is an *early-write* transaction.

We classify workloads according to the transactions they contain. *Read-dominated workloads* include only late-write transactions and read-only transactions; *write-dominated workloads* include only early-write transactions; and *bimodal workloads* include only early-write and read-only transactions.

2.2. Transactional schedulers and competitive measures

Each core is associated with a list of transactions available to be executed (possibly the same for all cores). Transactions are placed in the cores' lists according to an *insertion policy*. Once a core is not executing a transaction, it selects a transaction, according to a *selection policy*, and starts to execute it. An aborted transaction can be moved to a different core's list to prevent it from conflicting again with the transaction that caused it to abort. The selection and insertion policies also determine when an aborted transaction is restarted, in an attempt to avoid repeated conflicts.

We focus on conservative schedulers, which enforce consistency by aborting one of the transactions when a conflict happens, even if this conflict does not correspond to a serializability violation. Note that prominent transactional memory implementations (e.g., [16,6]) are conservative.

Definition 1. A scheduler A is *conservative* if it aborts at least one transaction when a conflict happens.

A scheduler is *work conserving* [3], if it always runs a maximal set of non-conflicting transactions.

Definition 2 (*Makespan*)**.** Given scheduler A and a workload Γ, $\text{makespan}_A(\Gamma)$ is the time A needs to complete all the transactions in Γ.

Author's personal copy

0	1	2		q
$core_1$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	...	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$
$core_2$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$
$core_3$	$R_1, R_2, \ldots, R_q, R_{q+3}, W_{q+3}$	$R_1, R_2, \ldots, R_q, R_{q+3}, W_{q+3}$	...	$R_1, R_2, \ldots, R_q, R_{q+3}, W_{q+3}$
⋮	⋮	⋮		⋮
$core_{q-1}$	$R_1, R_2, \ldots, R_q, R_{2q-1}, W_{2q-1}$	$R_1, R_2, \ldots, R_q, R_{2q-1}, W_{2q-1}$	...	$R_1, R_2, \ldots, R_q, R_{2q-1}, W_{2q-1}$
$core_q$	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
$core_{q+1}$	$R_1, R_2, \ldots, R_q$	$R_1, R_2, \ldots, R_q$	...	$R_1, R_2, \ldots, R_q$
⋮	⋮	⋮		⋮
$core_m$	$R_1, R_2, \ldots, R_q$	$R_1, R_2, \ldots, R_q$	...	$R_1, R_2, \ldots, R_q$

(a) The execution of the optimal scheduler.

0	1	2		q
$core_1$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
$core_2$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
$core_3$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
⋮	⋮	⋮		⋮
$core_{q-1}$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
$core_q$	$R_1, R_2, \ldots, R_q, R_{q+1}, W_{q+1}$	$R_1, R_2, \ldots, R_q, R_{q+2}, W_{q+2}$	...	$R_1, R_2, \ldots, R_q, R_{2q}, W_{2q}$
$core_{q+1}$	$R_1, R_2, \ldots, R_q$	$R_1, R_2, \ldots, R_q$	...	$R_1, R_2, \ldots, R_q$
⋮	⋮	⋮		⋮
$core_m$	$R_1, R_2, \ldots, R_q$	$R_1, R_2, \ldots, R_q$	...	$R_1, R_2, \ldots, R_q$

(b) The prefix of the execution of scheduler *A*.

Fig. 1. Executions used in the proof of Theorem 2.

Definition 3 (*Competitive Ratio*)**.** The *competitive ratio* of a scheduler *A* is

$$\max\left\{\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} : \Gamma \text{ is a workload}\right\}$$

where Opt is the optimal, clairvoyant scheduler knowing all the characteristics of the workload.

We only consider "reasonable" schedulers that utilize at least one core at each time unit for "productive" work. We say that a scheduler is *effective* if in every time unit, some transaction that eventually commits executes a unit of work (if there are any pending transactions).

Theorem 1. *Every effective scheduler A is $O(m)$-competitive.*

Proof. The theorem immediately follows from the fact that for every workload Γ, at each time unit some transaction makes progress, since *A* is effective. Thus, all transactions complete no later than time $\sum_{T_i \in \Gamma} \tau_i$ (as if they are executed serially). The claim follows since the best possible makespan for Γ, when all *m* cores are continuously utilized, is $\frac{1}{m}\sum_{T_i \in \Gamma} \tau_i$. □

3. Lower bounds

We start by proving a lower bound of $\Omega(s)$ on the competitiveness achievable by any scheduler, where *s* is the number of shared data items, for read-dominated workloads. This complements the lower bound for write-dominated workloads [3].

We use R_h, W_h to denote (respectively) a read and a write operation on data item i_h, $h = 1, 2, \ldots, s$.

Theorem 2. *There is a read-dominated workload Γ, such that every deterministic scheduler A is $\Omega(s)$-competitive on Γ, that is, $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} \geq c \cdot s$, for some constant $c > 0$.*

Proof. To prove our result, we first assume the scheduler *A* is work conserving, and then show how to remove this assumption.

Let *m* be the number of cores and *s* be the number of shared data items; we assume *s* is even. The proof uses an execution of $q \cdot m$ equi-duration transactions, where $q = \frac{s}{2}$. The data items $\{i_1, \ldots, i_s\}$ are divided into two disjoint sets, $D_1 = \{i_1, \ldots, i_q\}$ and $D_2 = \{i_{q+1}, \ldots, i_{2q}\}$.

We consider q^2 writing transactions and $q(m-q)$ read-only transactions, all arriving at time 0. All transactions have the same duration, normalized to 1. Writing transactions only write into data items in D_2 and read-only transactions only read data items in D_1. In particular, writing transactions are grouped into *q* sets of *q* transactions each, denoted S_j, $j = 1, 2, \ldots, q$, such that every transaction $T \in S_j$ reads all data items in D_1 and reads and writes to data item $i_{q+j} \in D_2$, i.e., $T = [R_1, R_2, \ldots, R_q, R_{q+j}, W_{q+j}]$.

Observe that a read-only transaction does not conflict with any other transaction, since its data set does not intersect the write set of any writing transaction. Moreover, any two transactions $T \in S_i$ and $T' \in S_j$, $i \neq j$, do not conflict. Transactions in S_i conflict since they read and write the same data item.

An optimal clairvoyant scheduler Opt executes the workload as shown in Fig. 1(a): at each time $t \in \{0, 1 \ldots, q-1\}$, Opt executes $m - q$ read-only transactions and *q* writing transactions, one from each set S_j, for $j = 1, 2, \ldots, q$. At each time unit, Opt successfully executes *m* transactions. By time *q*, Opt executes all $q \cdot m$ transactions, and its makespan is *q*.

The non-clairvoyant scheduler *A* knows only the first data item accessed by the transaction, and whether it is accessed for read or write. Being work-conserving, *A* picks a maximal set of non-conflicting transactions, which includes *m* transactions, since the first access of all transactions is to read R_1. Since all transactions have duration 1, the scheduler *A* selects a new set of *m* transactions at times $i = 0, 1, \ldots, q-1$. The execution can then be fixed so that the set of transactions picked by *A* at time *i* is the set S_{i+1} as well as

Author's personal copy

	0	α	2α				$1-\alpha$	1	$1+\alpha$	$1+2\alpha$			$1+(m-1)\alpha$
core_1	R_1	R_2	R_3	...	...	R_{s-1}	W_s	commit					
core_2		R_1	R_2	R_3	...	...	R_{s-1}	W_s	commit				
core_3			R_1	R_2	R_3	...	...	R_{s-1}	W_s	commit			
$\vdots$													
core_m						R_1	R_2	R_3	...	...	R_{s-1}	W_s	commit

Fig. 2. The execution of optimal scheduler used in the proof of Theorem 3.

$m-q$ read-only transactions (depicted in Fig. 1(b)), by choosing the later accesses of the transactions to follow this pattern. For every $j = 1, \dots, q-1$, let t_j be the time at which all late-write transactions in S_j have executed their read access to data item i_{q+j}, but none of them has already attempted to write it. It is simple to see that transactions can be scheduled for this to happen. Then, at some point after t_j, all these transactions attempt to write but only one of them can commit; otherwise, serializability is violated. All other late-write transactions abort. An aborted transaction can abort at some point after time t_j and before time j, while the other transactions commit at time j. Nevertheless, to simplify the presentation, we assume that all transactions in S_{j+1} are selected to start their execution at time j.

Aborted transactions are restarted at time q or later. At this time, only $q(m-q+1)$ transactions have committed. In particular, q^2-q late-write transactions still have to be executed. When restarted, all the aborted transactions write to the same data item i_1, i.e., restarted transactions are of the form $[R_1, \dots, R_q, R_{q+1}, W_1]$. This implies that all transactions end up being executed serially; even though they run in parallel, only one of them can commit at each time.

Thus, $\text{makespan}_A(\Gamma) \geq q^2$ and the competitive ratio is $\frac{q^2}{q} = q$, which is in $\Omega(s)$ since $q = \frac{s}{2}$.

To remove the assumption that the scheduler is work conserving, we modify the data set requirements or the transactions in the following way: if a transaction executes a write operation after time q, then it requests to write into i_1 as done in the above proof when a transaction is restarted.

Finally, we consider the case where the scheduler can distinguish between a new transaction that has never been executed and a previously aborted transaction. We assume that an aborted transaction is not executed before the conflicting transaction completes. This is usually guaranteed by the state-of-the art schedulers to avoid repeated conflicts. Next, we show that our result still holds, even if the scheduler re-executes aborted transactions before selecting a new transaction.

In particular, up to time q at most $m-q+3$ transactions commit at each time unit. We use an execution similar to the previous one. The main difference is that before selecting new writing transactions, the scheduler selects the transactions that aborted in the previous time unit (if any): At time 0, the scheduler selects the set S_0; at time 1, the scheduler selects the $q-1$ aborted transactions in S_0, $m-q$ read-only transactions in S_1 and one writing transaction in S_1; at time 2, the scheduler selects all the transactions previously aborted, $m-q$ read-only transactions in S_2 and two new writing transactions in S_1. This continues until time $\frac{q}{2}$, where we start to select writing transactions in the set S_2. All the transactions that execute after time q write into data item i_1, so, all q^2-3q remaining transactions are executed serially at time q. □

Next, we prove that when the scheduler is conservative (Definition 1), the makespan it guarantees is even less competitive. To understand the reasoning, we start by proving the result under the additional assumption that the scheduler is work conserving.

Theorem 3. *There is a late-write workload Γ such that every deterministic work-conserving conservative scheduler A has $\Omega(m)$-competitive makespan on Γ, that is, $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} \geq c \cdot m$, for some constant $c > 0$.*

Proof. Consider a workload Γ with m late-write transactions, all available at time 0. Each transaction $T_i \in \Gamma$ first reads data items $\{i_1, i_2, \dots, i_{s-1}\}$, and then writes to data item i_s, i.e., $T_i = [R_1, R_2, \dots, R_{s-1}, W_s]$, $i = 1, \dots, m$. All transactions have the same duration, normalized to 1. Being late-write transactions, the time from the first write operation to completion is negligible; we assume $\omega_i = \alpha < \frac{1}{m}$.

The optimal scheduler Opt has complete information on the set of transactions, and in particular, Opt knows that at time $1-\alpha$, each transaction will attempt to write to i_s. Thus, as depicted in Fig. 2, Opt delays the execution of the transactions so that no conflict happens: for every $i \in \{1, \dots, m\}$, T_i starts at time $(i-1)\alpha$. At time $1+(m-1)\alpha$, all transactions complete. Thus, $\text{makespan}_{\text{Opt}}(\Gamma) = 1+(m-1)\alpha$, where $\alpha < \frac{1}{m}$.

Since the scheduler A is work-conserving, it selects all the transactions to be executed at time 0. At time $1-\alpha$, all running transactions will attempt to write to i_s. Only one of these transactions commits, and the remaining $m-1$ transactions abort. When restarted, aborted transactions will write to data item i_1, instead of i_s. Thus, they have to be executed serially in order to not violate serializability. Thus, $\text{makespan}_A(\Gamma) = \sum_{i=1}^{m} 1-\alpha = m-\alpha$.

The competitive ratio is $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} = \frac{m-\alpha}{1+\alpha\cdot(m-1)} > \frac{m-\alpha}{1+\alpha\cdot m} > \frac{m-\alpha}{2} > \frac{m^2-1}{2m}$. □

The next theorem removes the assumption that the conservative scheduler is work-conserving; instead, it uses transactions whose duration is not fixed. The proof uses almost equi-duration transactions, in which the duration of transactions differ by at most a multiplicative factor of two.

Theorem 4. *There is a late-write workload Γ, with almost equi-duration transactions, such that every deterministic conservative scheduler A has $\Omega(m)$-competitive makespan on Γ, that is, $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} \geq c \cdot m$, for some constant $c > 0$.*

Proof. Consider a workload Γ with m late-write transactions, all available at time 0. Each transaction $T_i \in \Gamma$ first reads data items $\{i_1, i_2, \dots, i_{s-1}\}$, and then writes to data item i_s, i.e., $T_i = [R_1, R_2, \dots, R_{s-1}, W_s]$, $i = 1, \dots, m$. The duration of the transactions is determined by the adversary during the execution in a way that forces many transactions to abort.

Let Δ_i be the time interval $[i\frac{\tau}{2}, (i+1)\frac{\tau}{2})$, $i \geq 0$, for some $\tau > 0$. Let S_i be all the transactions that are scheduled for the first time in Δ_i. No matter at what time the scheduler A schedules the transactions in S_i to start, we choose their durations so that at time $(i+2)\frac{\tau}{2} - \alpha \cdot \tau$, with $\alpha < \frac{1}{m}$, they all execute their write operation to data item i_s. Only one of these transactions commits, while the other transactions abort. In particular, the committed transaction commits at time $(i+2)\frac{\tau}{2}$. When restarted, an aborted transaction writes to data item i_1, instead of i_s and its duration is τ. Aborted transactions have to be executed serially, in order to not violate serializability.

For each $i \geq 0$, at most two transactions commit in Δ_i: one restarted transaction and one transaction scheduled for the first time in Δ_{i-1}. Thus, $\text{makespan}_A(\Gamma) \geq \frac{m}{2} \cdot \frac{\tau}{2} = \frac{1}{4} \cdot m \cdot \tau$.

On the other hand, the optimal clairvoyant scheduler Opt has complete knowledge about the workload, and in particular, knows

Author's personal copy

when the transactions execute their write operation. Let t_i be the time at which transaction T_i has to be started in order to execute its first write operation at time $(1-\alpha)\tau,\ i = 1, 2, \ldots, m$. The optimal scheduler starts transaction T_i at time $t_i + (i-1)\alpha \cdot \tau,\ i = 1, 2, \ldots, m$. (Fig. 2 shows an example with $\tau = 1$, and $t_i = 0$, for every $i = 1, 2, \ldots, m$.) Then $\text{makespan}_{\text{Opt}}(\Gamma) = \tau + (m-1)\alpha \cdot \tau$.

The competitive ratio is $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{\text{Opt}}(\Gamma)} \geq \frac{\frac{m\cdot\tau}{4}}{\tau+(m-1)\alpha\cdot\tau} = \frac{m}{4(1+(m-1)\alpha)} > \frac{m}{8}$. □

In fact, the makespan is not competitive even relative to a clairvoyant *online* scheduler [8], which does not know the workload in advance, but has complete information on a transaction once it arrives, in particular, the set of resources it is going to access. This agrees with the result [8] that knowing – at release time – the data items a transaction is going to access, improves the performance achieved by a transactional scheduler.

4. Dealing with bimodal workloads: a motivating example

Several transactional schedulers [7,29,2,19] attempt to reduce the overhead of transactional memory by serializing conflicting transactions. Unfortunately, these schedulers are conservative and hence they are $\Omega(m)$-competitive. Moreover, they do not distinguish between read and write accesses and do not provide special treatment to read-only transactions, causing them not to work well also with bimodal workloads.

There are bimodal workloads of m transactions (m is the number of cores) for which both CAR-STM [7] and ATS [29] have a competitive ratio (relative to the optimal offline scheduler) that is at least $\Omega(m)$. This is because both CAR-STM and ATS do not ensure the *list scheduler* property, i.e., they make a transaction wait even if the resources it needs are available. In fact, to reduce the wasted work due to repeated conflicts, these schedulers may serialize also read-only transactions.

Steal-on-Abort (SoA) [2], in contrast, allows free cores to take transactions from the queue of another busy core; thus, it ensures the list scheduler property, trying to execute as many transactions concurrently as possible. However, in an overloaded system, with more than m transactions, SoA may create a situation in which a starved writing transaction can starve read-only transactions. This leads to bimodal workloads in which the makespan of Steal-on-Abort is $\Omega(m)$ competitive, as we show below. (Steal-on-abort [2] and other transactional schedulers [19,7,29], are effective, and hence they are $O(m)$-competitive, by Theorem 1.)

The Steal-On-Abort (SoA) scheduler. Application threads submit transactions to the pool of a corresponding transactional thread. Each transactional thread has a work queue where available transactions wait to be executed. When new transactions are available, they are distributed among the transactional threads' queues in round-robin manner.

When two running transactions T and T' conflict, the contention manager policy decides which of them to commit. The aborted transaction, say T', is then "stolen" by the transactional thread executing T and is enqueued in a designated *steal queue*. Once the conflicting transaction commits, the stolen transaction is taken from the steal queue and inserted to the work queue. There are two possible insertion policies: T' is enqueued either in the head or in the tail of the queue. Transactions in a queue are executed serially, unless they are moved to other queues. This can happen either due to a new conflict or because some transactional thread becomes idle and steals transactions from the work queue of another transactional thread (chosen uniformly at random) or from the steal queue if all work queues are empty.

SoA suggests four strategies for moving aborted transactions: *steal-tail*, *steal-head*, *steal-keep* and *steal-block*. We describe a bad scenario for the steal-tail strategy, which inserts the transactions aborted because of a conflict with a transaction T, at the tail of the work queue of the transactional thread that executed T (after T completes). Similar scenarios can be shown for the other strategies.

The SoA scheduler does not specify any policy to manage conflicts. In [2], the SoA scheduler is evaluated empirically with three contention management policies: the simple *Aggressive* and *Timestamp* contention managers, and the more sophisticated *Polka* contention manager.[3] Yet none of these policies outperform the others, and the optimal one depends on the workload. This result is corroborated by an empirical study that has shown that no contention manager is universally optimal, and performs best in all reasonable circumstances [10]. Moreover, while several contention management policies have been proposed in the literature [24,11], none of them, except *Greedy* [11], has nontrivial provable properties.

Thus, we consider the SoA scheduler with a contention management policy based on timestamps, like Greedy [11] or Timestamp [24]. These policies do not require costly data structures, like *Polka*. Our choice also provides a fair comparison with *CAR-STM*, which embeds a contention manager based on timestamps.

Theorem 5. *Steal-on-Abort with steal-tail has $\Omega(m)$-competitive makespan for some bimodal workload.*

Proof. We consider a workload Γ with $n = 2m-1$ unit-duration transactions, two writing transactions and $2m-3$ read-only transactions (see Fig. 3). At time $t_1 = 0$, a writing transaction $U_1 = [R_1, W_1]$ is available and at time $t_1 + \epsilon$, when the writing transaction is executing its first access, and a set S_1 of $m-1$ read-only transactions $[R_2, R_1, R_3]$ become available.

All the transactions are immediately executed. But in their second access, all the read-only transactions have conflict with the writing transaction U_1. All the read-only transactions are aborted, because U_1 have a greater priority than them, and they are inserted to the work queue of the transactional thread where U_1 is executing.

At time t_2, immediately before U_1 completes, $m-1$ other transactions become available: a writing transaction $U_2 = [R_3, W_4, W_3]$ and a set S_2 of $m-2$ read-only transactions $[R_1, R_4]$. Each of these transactions is placed in one of the idle transactional threads, as depicted in Fig. 3.

Immediately after time t_2, U_2, all the transactions in S_2 and one read-only transaction in S_1 are running. In their second access, all the read-only transactions in S_2 conflict with the writing transaction U_2. U_2 discovers the conflict and aborts all the read-only transactions in S_2, by having U_2 arriving immediately before the read-only transactions, and obtaining higher priority.

The aborted read-only transactions are then moved to the queue of the worker thread which is currently executing U_2. Then, U_2 conflicts with the third access of the read-only transaction in S_1. Thus, U_2 is aborted and it is moved to the tail of the corresponding work queue. We assume the time between cascading aborts is negligible.

We then repeat the above scenario, until all transactions commit. In particular, for every $i \in \{3, \ldots, m\}$, we have that immediately before time t_i, there are $m-i+1$ read-only

[3] In the *Aggressive* contention manager, a conflicting transaction always aborts the competing transaction. In the *Timestamp* contention manager, each transaction is associated with the system time when it starts and the newer transaction is aborted, in case of a conflict. The *Polka* contention manager increases the priority of a transaction whenever the transaction successfully acquires a data item; when two transactions are in conflict, the attacking transaction makes a number of attempts equal to the difference among priorities of the transactions before aborting the competing transaction, with a exponential backoff between attempts [24].

Author's personal copy

time	thread 1	thread 2	...	thread $i-1$	thread i	...	thread $m-1$	thread m
t_1	$[R_1,W_1]$		...			...		
$t_1+\epsilon$	$[R_1,W_1]$ $(m-1)$ $[R_2,R_1,R_3]$	$[R_2,R_1,R_3]$	...	$[R_2,R_1,R_3]$	$[R_2,R_1,R_3]$	...	$[R_2,R_1,R_3]$	$[R_2,R_1,R_3]$
t_2	$[R_2,R_1,R_3]$ $(m-2)$ $[R_2,R_1,R_3]$; $[R_3,W_4,W_3]$	$[R_3,W_4,W_3]$ $(m-2)$ $[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
t_{i-1}	$[R_2,R_1,R_3]$ $(m-i+1)$ $[R_1,R_4]$; $[R_3,W_4,W_3]$	$[R_3,W_4,W_3]$	...	$[R_1,R_4]$ $(m-2)$ $[R_1,R_4]$	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$
t_i	$[R_2,R_1,R_3]$ $(m-i)$ $[R_2,R_1,R_3]$; $[R_3,W_4,W_3]$	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_3,W_4,W_3]$ $(m-2)$ $[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
t_{m-1}	$[R_2,R_1,R_3]$ $[R_3,W_4,W_3]$	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$	...	$[R_3,W_4,W_3]$ $(m-2)$ $[R_1,R_4]$	$[R_1,R_4]$
t_m	–	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_3,W_4,W_3]$
t_{m+1}	–	$[R_1,R_4]$	...	$[R_1,R_4]$	$[R_1,R_4]$	...	$[R_1,R_4]$	–

Fig. 3. Steal-On-Abort with steal-tail strategy: illustration for Theorem 5. Each table entry (i, j) shows at the top the transaction executed by thread j at time t_i, and at the bottom, the status of the main queue of thread j immediately before time t_{i+1}. $(k)[R_i, W_j]$; $[R_h, R_l]$ denotes a work queue with k transactions $[R_i, W_j]$ and one read-only transaction $[R_h, R_l]$, in this order, from head to tail. If there is no transaction in such a queue, the bottom line is empty.

transactions $[R_2, R_1, R_3]$ and the writing transaction U_2 in the work queue of thread 1 and $m-2$ read-only transactions $[R_1, R_4]$ in the work queue of thread $i-1$. The work queues of the remaining threads are empty. Then, at time t_i, the worker thread i takes the writing transaction from the work queue of thread 1 and the other free worker threads take a read-only transaction $[R_1, R_4]$ from the work queue of thread $i-1$. Thus, at each time t_i, $i \in \{3, \dots, m\}$, the writing transaction U_2, one read-only transaction $[R_2, R_1, R_3]$ and $m-2$ read-only transactions $[R_1, R_4]$ are executed, but only the read-only transaction in S_1 commits.

Finally, at time t_m, U_2 commits and hence, all read-only transactions in S_2 commit at time t_{m+1}.

Note that in the scenario we construct, the way each thread steals the transactions from the work queues of other threads is governed by a uniform random distribution as required by the Steal-on-Abort policy.

Thus, $\text{makespan}_{\text{SoA}}(\Gamma) = m + 2$. On the other hand, the makespan of an optimal offline algorithm is less than 4, because all read-only transactions can be executed in 2 time units, and hence, the competitive ratio is at least $\frac{m+2}{4}$. $\square$

In the next section, we present a conservative scheduler, using a simple timestamp-based contention management policy, which is $O(s)$-competitive for bimodal workloads.

5. The Bimodal scheduler

The architecture of Bimodal is similar to CAR-STM [7]: each core has a work double-ended queue, where a *transactional dispatcher* enqueues arriving transactions; the work queue of core i, $1 \le i \le m$, is denoted q_i. Transactions can be enqueued either at the tail or at the head of the double-ended queue, but they are dequeued only from the head of the queue.

Additionally, there is a fifo queue, called *ROqueue*, shared by all cores, to hold transactions that abort before executing a write operation; such transactions are assumed to be read-only transactions.

Bimodal requires read operations to be visible, in order to detect conflicts as soon as possible.

When two transactions have a conflict, one of them is aborted and Bimodal prohibits them from executing concurrently again and possibly repeating the conflict. In particular, if the aborted transaction is a writing transaction, it is moved to the work queue of the conflicting transaction; otherwise, it is enqueued in the *ROqueue*.

Bimodal alternates between periods in which it privileges the execution of writing transactions, called *writing epochs*, and periods in which it privileges the execution of read-only transactions, called *reading epochs*. Regardless of the epoch, when there is a conflict between two writing transactions, the contention manager aborts the newer transaction. To support this decision, as in [11], a transaction is assigned a timestamp when it starts, which is retained even if the transaction aborts. The transaction with a larger timestamp is newer.

5.1. Detailed description of the Bimodal scheduler

Transactions are assigned to the work queues of the cores (inserted at their tail), starting from cores whose work queue is empty; initially, all work queues are empty. The dispatcher works in round-robin manner if the queues of the cores have the same number of transactions to process; otherwise, the queue with fewest transactions is filled first.

At each point during the execution, the system is in a specific *epoch*, uniquely identified by a monotonically increasing integer identifier, *ID*. Odd and even identifiers correspond to reading and writing epochs, respectively.

The algorithm uses the following state variables at each transaction T_i:

- *ts* $\in \mathbb{N}$, initialized when the transaction starts.
- *queue* $\in \{q_1, \dots, q_m, ROqueue\}$, initialized by the dispatcher. Indicates the queue where T_i is enqueued, either when it starts or after being aborted.

Author's personal copy

– *status* $\in$ {active, aborted, committed}.
– *epoch* $= ID$ where $ID \in \mathbb{N}$. Indicates the epoch in which the transaction starts.
– *readOnly* $\in$ {true, false}, initially, *readOnly* $=$ true.

The algorithm also uses the following shared variables:

– $\xi \in \mathbb{N}$, holds the ID of the current epoch; it is initially 0, indicating a writing epoch. ξ is modified with compare&swap (CAS) operations.[4]
– *ROqueue*, a fifo queue, initially empty.
– *ROqueue.count* $\in \{0, 1, \ldots, m\}$, initially, *ROqueue.count* $= 0$.

The algorithm uses the following functions:

– *size*(q) returns the number of transactions in a queue q.
– *dequeue*(q) returns and deletes the transaction at the head of a queue q.
– *insertTail*(q, T) inserts the transaction T at the tail of a double-ended queue q.
– *insertHead*(q, T) inserts the transaction T at the head of a double-ended queue q.
– *isReading*(e) takes as input an epoch id (an integer) e and returns true if e is a reading epoch (odd number), and false, otherwise.

Algorithm 1 describes how a core picks the next transaction to execute. During a writing epoch, each core selects a transaction from its work queue (if it is not empty). When in a writing epoch i, the system moves to a reading epoch $i + 1$ if there are m transactions in *ROqueue* or if all work queues are empty and there is a transaction in *ROqueue* (lines 17 and 18). Together with the assumption that the workload is finite, the last condition ensures that no transaction in *ROqueue* is indefinitely waiting to execute. The function allqueueEMPTY() returns true if all queues are empty, and it does so after reading all work queues twice (i.e., a *double collect* of the state of the work queues [1]). If allqueueEMPTY() returns true, then there is a point during the double collect in which all work queues are empty. Note, however, that the function may return false, although there is a point in which all work queues are empty.

Let R be the set of read-only transactions in *ROqueue* when the system moves to the $(i + 1)$th reading epoch (lines 19 and 21). Once all the transactions in R are dequeued or m read-only transactions have being processed, whichever happens first, the system enters writing epoch $i + 2$. The shared counter *ROqueue.count* monitors when $|R|$ transactions were dequeued from *ROqueue*; it is modified with CAS operations. *ROqueue.count* is set to $|R|$ when the system enters a reading epoch (lines 19 and 21 of Algorithm 1); whenever a transaction is dequeued from *ROqueue*, *ROqueue.count* is decremented (line 6 of Algorithm 1). When *ROqueue.count* is 0, the system enters a writing epoch (lines 11 and 14 of Algorithm 1).

After a core completes to execute its current transaction, it verifies the epoch number in order to decide from which queue to take the next transaction to execute. A core moves the system to a new epoch if the expected conditions hold (line 11 for a writing epoch or line 17 for a reading epoch). In particular, the core that dequeues the last transaction of R updates the shared variable ξ to enter a new writing epoch (line 14).

A transaction T that starts in the i-th epoch, is associated with epoch i until T either commits or aborts. An aborted transaction is associated with a new epoch when restarted. When a core selects the new transaction to execute, it associates the current epoch with the transaction (lines 8, 13 and 26).

Algorithm 1 Selection of a new transaction to execute by core i

```
1:  Upon core_i idle:
2:    epoch ← ξ
3:    if isReading(epoch) then
4:      count ← ROqueue.count
5:      if count > 1 then
6:        if CAS(ROqueue.count, count, count − 1) then
7:          currT ← Deque(ROqueue)
8:          currT.epoch ← epoch
9:        else go to line 2
10:     else
11:       if CAS(ROqueue.count, 1, 0) then
12:         currT ← Deque(ROqueue)
13:         currT.epoch ← epoch
14:         CAS(ξ, epoch, epoch + 1)
15:       else go to line 2
16:   else
17:     if (size(ROqueue) ≥ m) OR (allqueueEMPTY()) then
18:       if (size(ROqueue) ≠ 0) then
19:         if CAS(ξ, epoch, epoch + 1) then
20:           c ← min (m, ROqueue.size())
21:           CAS(ROqueue.count, 0, c)
22:       go to line 2
23:     else
24:       if size(queue_i) ≠ 0 then
25:         currT ← Deque(queue_i)
26:         currT.epoch ← epoch
27:       else
28:         go to line 2
```

It may happen that while a transaction T, associated with epoch i, is running, the system transitions to an epoch $j > i$. When this happens, we say that epochs i and j *overlap*. To manage conflicts between transactions associated with different epochs, we give higher priority to the transaction associated with the earlier epoch, as explained below.

Conflict management, shown in Algorithm 2, is different in writing and reading epochs.

In a writing epoch,

1. A read-only transaction that conflicts with a writing transaction is aborted and enqueued in *ROqueue* (lines 14, 21, and 30). Each transaction has a Boolean variable *readOnly*, initially true. When the transaction executes its first write operation (i.e., requests to write to a data item), this variable is set to false, and it remains false until it commits. This means that if the transaction is aborted and the value of its *readOnly* variable is false, this value is carried over when the transaction restarts.
2. If there is a conflict between two writing transactions T_1 and T_2, and the timestamp of T_2 is bigger than the timestamp of T_1, then T_2 is inserted at the head of the work queue of T_1 (lines 3, 4, 5 and 31). This is similar to the *permanent serializing* contention manager of CAR-STM.

We may have *false positives*, i.e., a writing transaction T that is enqueued in *ROqueue* because it is mistakenly considered to be a read-only transaction (before its first write operation). This transaction T may have a conflict, due to later write operations.

During a reading epoch, conflicts occur either due to false positives, or because epochs overlap, i.e., transactions with different epoch IDs are executed concurrently. Although the epoch ID of the transaction remains fixed from the beginning of its

[4] A CAS(o, u, v) operation on the object o, checks whether the value of o is u and, if so, sets o to v and returns true. Otherwise, the value of o does not change and false is returned.

Author's personal copy

Algorithm 2 Transaction T_i executed by core i detects a conflict with transaction T_j executed by core j

```
1:  Upon ⟨conflict(T_j)⟩:
2:    queue_j ← T_j.queue
3:    if (¬T_j.readOnly) AND (¬T_i.readOnly) then
4:      if T_j.ts > T_i.ts then
5:        T_j.queue ← queue_i
6:        CAS(T_j.status,active,aborted)
7:    else
8:      if T_j.epoch = T_i.epoch then
9:        if T_i.readOnly then
10:         if isReading(epoch) then
11:           T_j.queue ← queue_i
12:           CAS(T_j.status,active,aborted)
13:         else
14:           T_i.queue ← ROqueue
15:           CAS(T_i.status,active,aborted)
16:       else
17:         if isReading(epoch) then
18:           T_i.queue ← queue_j
19:           CAS(T_i.status,active,aborted)
20:         else
21:           T_j.queue ← ROqueue;
22:           CAS(T_j.status,active,aborted);
23:     else
24:       if T_i.epoch > T_j.epoch then
25:         CAS(T_i.status,active,aborted)
26:       else
27:         T_j.queue ← queue_j
28:         CAS(T_j.status,active,aborted)
29: Upon ⟨status = aborted⟩:
30:   if (T_i.queue = ROqueue) then insertTail(ROqueue, T_i)
31:   else insertHead(T_i.queue, T_i)
```

execution until it either aborts or commits, the system may transition to a new epoch (i.e., the variable ξ increases).

1. If two transactions have the same epoch ID, one of them a read-only transaction and the other a writing transaction, then the writing transaction is aborted (lines 12 and 19). If the conflict is between two writing transactions, then one aborts and the other transaction simply continues its execution; as in a writing epoch, the decision is based on their timestamps (lines 3, 4 and 5). If aborted, a writing transaction is enqueued in the head of the work queue of the core where the conflicting transaction executes (lines 5 and 18).
2. When two transactions T and T' with different epoch IDs have a conflict, the transaction with the bigger epoch ID is aborted and it is enqueued at the head of the core's work queue where it was executed before (lines 24–27).

5.2. Analysis of the Bimodal scheduler

Recall that ρ_i is the execution time of a transaction T_i until its first write access, and that $\rho_i = \tau_i$, the duration of T_i, if T_i is a read-only transaction. We first bound (from below) the makespan that can be achieved by an optimal conservative scheduler.

Theorem 6. *For every workload Γ, the makespan of Γ under an optimal, conservative offline scheduler* Opt *satisfies* $\text{makespan}_{\text{Opt}}(\Gamma) \geq \max\{\frac{\sum \omega_i}{s}, \frac{\sum \tau_i}{m}\}$.

Proof. There are m cores, and hence, the optimal scheduler executes at most m transactions in each time unit; therefore, $\text{makespan}_{\text{Opt}}(\Gamma) \geq \frac{\sum \tau_i}{m}$.

For each transaction T_i in Γ with $\omega_i \neq 0$, let F_i be the first item T_i modifies.

Any two transactions T_i and T_j whose first write access is to the same item, i.e., that have $F_i = F_j$, have to execute the part after their write serially. Thus, at most s transactions with $\omega_i \neq 0$ proceed at each time, implying that $\text{makespan}_{\text{Opt}}(\Gamma) \geq \frac{\sum \omega_i}{s}$. □

We analyze Bimodal under the assumption that all transactions in a given workload $\Gamma = \{T_1, \ldots, T_n\}$ have a duration $\tau_1, \ldots, \tau_n$, such that for every $i \neq j$, $\tau_i \leq C \cdot \tau_j$ for some positive constant C. In particular, the analysis holds when the durations of transactions differ by a multiplicative factor of two, as considered in the proof of Theorem 4.

The time needed to run all read-only transactions in Γ concurrently is less than or equal to $\frac{\sum_{i=1}^{n} \rho_i}{m}$.

Moreover, if a writing transaction is enqueued in *ROqueue* and executed during a reading epoch, since it is falsely considered to be a read-only transaction, then either it completes successfully without encountering conflicts or it is aborted and treated as a writing transaction once restarted.

Theorem 7. Bimodal *is $O(s)$-competitive on bimodal workloads.*

Proof. Consider the scheduling of a bimodal workload Γ under Bimodal. Let τ be the maximal duration of a transaction in Γ; i.e., $\tau = \max\{\tau_i \mid T_i \in \Gamma\}$.

Let $\bar{t}$ be the last time at which the work queues of all cores are empty, and such that some transaction arrives after $\bar{t}$. Otherwise, $\bar{t} = 0$ is the time immediately before the first transaction in Γ is available.

At time $\bar{t}$, no transactions are available in the work queue of any core, and hence, no matter what the optimal scheduler Opt does, its makespan is at least $\bar{t}$.

Let $\Gamma_{\bar{t}}$ be the set of transactions that are available after time $\bar{t}$, and let $n_{\bar{t}} = |\Gamma_{\bar{t}}|$. (If $\bar{t} = 0$, then $\Gamma_{\bar{t}} = \Gamma$.) Since at time $\bar{t}$, Opt does not schedule any transaction, it schedules new transactions to execute as they arrive. By Theorem 6,

$$\text{Makespan}_{\text{Opt}}(\Gamma_{\bar{t}}) \geq \frac{1}{2}\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m}\right),$$

and therefore,

$$\text{Makespan}_{\text{Opt}}(\Gamma) \geq \bar{t} + \frac{1}{2}\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m}\right). \quad (1)$$

On the other hand, Bimodal may delay the execution of new available transactions because the cores are executing the transactions in *ROqueue* (if any). We show that, at any point, there are at most $2m$ transactions in *ROqueue*, and hence, executing all pending read-only transactions in *ROqueue* requires at most 2τ time units.

The proof is by induction on the number of transactions selected for execution by the cores. Initially, *ROqueue* is empty. Each time a core selects a new transaction to execute, it checks if the *ROqueue* contains m transactions, and if so, moves the system to a reading epoch and takes a transaction to execute from *ROqueue*. When a reading epoch starts, at most $m-1$ transactions are executing, and at most all but one of them may abort and be inserted into *ROqueue*, which will then contain at most $2m-3$ transactions. The reading epoch ends only after m transactions from *ROqueue* are handled or *ROqueue* is empty, and the claim follows.

Author's personal copy

Hence, the time for Bimodal to complete the new set of transactions $\Gamma_{\bar{t}}$ is:

$$\text{Makespan}_{\text{Bimodal}}(\Gamma_{\bar{t}}) \leq 2\tau + \sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i \right).$$

This holds since $2\omega_i \geq \tau_i$ ($\omega_i = \alpha\tau_i$ with α close to 1), for every writing transaction $T_i \in \Gamma_{\bar{t}}$, and to account for writing transactions executed during reading epochs. In fact, a writing transaction T may conflict only once during a reading epoch, because when restarted T is no longer treated as a read-only transaction. This is just as if T is executed during a writing epoch, with its duration doubled, to account for its execution during a reading epoch (if there is one). Therefore, the time required by Bimodal to execute $\Gamma_{\bar{t}}$ is less than or equal to the time needed to execute all writing transactions serially, taking their duration to be $4\omega_i$, and all read-only transactions in parallel, when there are no more writing transactions to execute.

The time for Bimodal to complete the entire workload Γ is:

$$\text{Makespan}_{\text{Bimodal}}(\Gamma) \leq \bar{t} + 2\tau + \sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i \right). \tag{2}$$

Since the maximal duration of a transaction τ is larger than the minimal duration by a constant multiplicative factor C, we have that for every transaction $T_i \in \Gamma_{\bar{t}}$, $\tau \leq C \cdot \tau_i$, where τ_i is the duration of T_i. If T_i is a writing transaction, then $\tau \leq C \cdot 2\omega_i$. Thus, we can overestimate

$$\tau \leq C \cdot \sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i \right).$$

Substituting in (2):

$$\text{Makespan}_{\text{Bimodal}}(\Gamma) \leq \bar{t} + (2C+1) \sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i \right).$$

Combining with (1), this yields:

$$\frac{\text{Makespan}_{\text{Bimodal}}(\Gamma)}{\text{Makespan}_{\text{Opt}}(\Gamma)} \leq \frac{\bar{t} + (2C+1)\sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i\right)}{\bar{t} + \frac{1}{2}\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)}$$

$$= \frac{\bar{t}}{\bar{t} + \frac{1}{2}\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)} + \frac{(2C+1)\sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i\right)}{\bar{t} + \frac{1}{2}\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)}$$

$$\leq 1 + \frac{2(2C+1)\sum_{i=1}^{n_{\bar{t}}} \left(\frac{\rho_i}{m} + 4\omega_i\right)}{\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)}$$

$$= 1 + 2(2C+1) \frac{\sum_{i=1}^{n_{\bar{t}}} \frac{\rho_i}{m}}{\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)} + 8(2C+1) \frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{\left(\frac{\sum_{i=1}^{n_{\bar{t}}} \omega_i}{s} + \frac{\sum_{i=1}^{n_{\bar{t}}} \tau_i}{m} \right)}$$

$$< 1 + (4C+2) + (16C+2) \frac{m \cdot s \sum_{i=1}^{n_{\bar{t}}} \omega_i}{m \sum_{i=1}^{n_{\bar{t}}} \omega_i + s \sum_{i=1}^{n_{\bar{t}}} \tau_i} \quad \text{since } \rho_i \leq \tau_i$$

$$< 1 + (4C+2) + (16C+2) \frac{m \cdot s \sum_{i=1}^{n_{\bar{t}}} \omega_i}{m \sum_{i=1}^{n_{\bar{t}}} \omega_i}$$

$$= 1 + (4C+2) + (16C+2)s$$

which is in $O(s)$, since C is a constant.

If there is no writing transaction in $\Gamma_{\bar{t}}$, then after time $\bar{t} + 2\tau$, Bimodal behaves exactly like the optimal scheduler by executing all read-only transactions concurrently. Hence, Bimodal takes at most a constant time 2τ more than Opt to complete the workload. □

6. Related work

Contention managers [16,24] were suggested as a mechanism for resolving conflicts and improving the performance of transactional memories. Contention managers have control over the period an aborted transaction has to wait before it can restart. Being a local decision, an aborted transactions is not guaranteed to restart only after the conflicting transaction has completed; once restarted, the transaction may experience the same conflict and abort again. A transaction that continuously aborts is wasting CPU resources, without doing any work, although other transactions could be executed successfully during this time period.

Guerraoui et al. [11] provided the first theoretical analysis of contention managers, showing that the Greedy contention manager is $O(s^2)$ competitive relative to an optimal offline scheduler, where s is the number of shared data items. The Greedy contention manager ensures the *pending commit* property, i.e., at any time, some running transaction executes uninterrupted until it commits.

Attiya et al. [3] recast this question in the framework of non-clairvoyant scheduling, originally proposed to investigate task scheduling in multi-processing environments. The original work (e.g., [23]) did not consider effects of concurrency control, and mostly assume that a preempted transaction resumes execution from the same point, and is not restarted. Attiya et al. show that every work conserving contention manager ensuring ensures the pending-commit property, is $O(s)$ competitive. They also show that this bound is tight for any deterministic contention manager, and under certain assumptions about the transactions, also for randomized contention managers.

Later research [29,7,2] advocate taking more control over the scheduling of transactions, in order to avoid repeated conflicts and reduce the amount of work wasted by aborted transactions. The main idea is to serialize the execution of conflicting transactions as soon as conflicts are discovered. This avoids repeated conflicts, but requires special care not to limit the possible parallelism by serializing transactions whose data sets do not intersect but that conflicted with the same transaction.

In CAR-STM [7], each core manages a queue of all transactions that conflict with the transaction it is executing; these are executed serially, after the conflicting transaction completes. Steal-on-Abort [2] uses a similar approach, but once a processor has no more transactions to execute in its queue, it can steal transactions from the queues of other processors and execute them. This avoids under-utilization of cores, at the cost of repeated conflicts. Yoo and Lee [29] propose to control the level of contention by monitoring the ratio of committed transactions. This information

Author's personal copy

on the contention intensity exposes the inherent parallelism of a given workload and allows the scheduler to adapt the execution according to the workload: if contention is high, the number of transactions that execute concurrently is limited by enquing all aborted transactions in a main queue, to be executed serially. This is equivalent to putting a single lock on all shared data when the contention is high. Despite using different mechanisms, these transactional schedulers serialize more than is necessary in read-dominated workloads, and in particular, they serialize read-only transactions.

The architecture of Bimodal is similar to CAR-STM [7] except for distinguishing between writing and read-only transactions, which avoids serializing the execution of read-only transactions. Our approach may cause transactions to suffer repeated conflicts only when the system transitions from a reading to a writing epoch or vice versa.

More recent transactional schedulers [15,19] have aborted transactions wait only for the completion of the winning transaction, and not for each other. This is done either at the user level, by waiting on a spinlock held by the winning transaction, or at the kernel level, by blocking the thread on a condition variable of the winning transaction. When the winning transaction completes, it releases the spinlock or wakes up all transactions sleeping on its condition variable, respectively. In the user-level approach, an aborted transaction keeps the CPU busy without doing any work, while the kernel-level approach relies on expensive system calls.

Dragojevic et al. [8] take a complementary approach, trying to predict the accesses of transactions based on past behavior, together with a heuristic mechanism for serializing transactions that may conflict. They also present counter-examples to CAR-STM [7] and ATS [29], although they do not explicitly detail which accesses are used to generate the conflicts that cause transactions to abort; in particular, they do not distinguish between access types, and the portion of the transaction that requires exclusive access.

Schneider and Wattenhofer [25] present a deterministic contention manager, which ensures that every transaction completes in bounded time, without global information (unlike Greedy [11]). They also show that if the data sets of transactions may vary over time such that the number of conflicting transactions increases by a factor of k, then the competitive ratio of any deterministic conservative scheduler is $\Omega(k)$ for $k < \sqrt{m}$. Our proof of Theorem 3 (the $\Omega(m)$ lower bound on the competitive ratio of deterministic conservative schedulers for late-write workloads) uses transactions that have a slightly different data set once restarted; however, the number of conflicting transactions does not increase.

Effective schedulers, considered in this paper, execute at each time a unit of work of some transaction that eventually commits. This is a generalization of the *pending commit* property [11]. The pending commit property implies that the same transaction makes progress at each time, until it commits. Therefore, a schedule that at time t lets a transaction T make progress and at time $t+1$ lets a different transaction T' make progress, even though T did not commit, but eventually ensures that both T and T' commit, is effective but does not guarantee the pending commit property. Bimodal guarantees the pending commit property and therefore, it is effective.

Effectiveness is a property orthogonal to *strong progressiveness* [13], a property aiming to capture the progress of lock-based transactional memories, since effectiveness allows non-conflicting transactions to abort, as long as some transaction executes a unit of work towards its commit. On the other hand, strong progressiveness does not require any progress for transactions that conflict on more than one data item, e.g., if transaction T_1 conflicts on a data item x with T_2 and both T_1 and T_2 do not conflict with any other transaction on a different data item, then either T_1 or T_2 should commit. But if, for example, in addition to the conflict on x, T_2 also conflicts on a data item y with T_3, then both T_1 and T_2 can abort.

Keidar and Perelman [22] consider *strict online permissiveness*, i.e., a transactional memory can abort some transaction only if not aborting any transaction would violate correctness, and they prove that it cannot be ensured efficiently.

Initial analysis of transactional scheduling considered write-dominated workloads [11,3,25]. Following the initial publication of our paper [4], Sharma et al. [26,28,27] considered the competitive ratio of schedulers that decide based on a priori knowledge about the workload (duration and data set of transactions). They propose the Clairvoyant contention manager, which is $O(\sqrt{s})$-competitive for *balanced workloads*, in which the data items written by a transaction are a constant fraction of its data set [26]. This holds for a one-shot situation, where each thread has only one transaction to execute, and equi-duration transactions. They prove that Clairvoyant is optimal by showing it is impossible to approximate in polynomial time any transactional scheduler with competitive ratio smaller than $O(\sqrt{s}^{1-\epsilon})$, for any constant $\epsilon > 0$; this holds even for balanced workloads with unit length transactions. They also propose a randomized contention manager that does not rely on a priori knowledge, which is $O(\sqrt{s \log n})$-competitive with high probability, where n is the number of transactions. In followup work [28], they consider workloads in which every thread executes up to M transactions serially, by dividing the execution into *windows*.

Liu and Spear [18] consider *toxic transactions*, which conflict with most of the other transactions, and suggest they should be executed serially, in a manner similar to [29]. They propose the Hourglass contention manager, in which a transaction that is consecutively aborted more than a certain threshold, tries to gain an exclusive token. When the transaction gains the token, other transactions are not allowed to start their execution.

Kim and Ravindran [17] use the epoch approach of Bimodal for transactional scheduling in networked distributed systems; they divide the execution into intervals where read-only and writing transactions are prioritized, and show the benefits of this approach compared with a traditional contention manager.

7. Discussion

This paper studies the competitive ratio achieved by non-clairvoyant transactional schedulers on read-dominated workloads. We have presented the Bimodal transactional scheduler, which allows to achieve maximum parallelism on read-only transactions, without harming early-write transactions. On the other hand, we proved that the long reading periods of late-write transactions cannot be overlapped to exploit parallelism, and must be serialized if the writes at the end of the transactions are in conflict. The latter result assumes that the scheduler is conservative, namely, it aborts at least one transaction involved in a conflict. This is the approach advocated in [16] as it reduces the cost of tracking conflicts and dependencies. It is interesting to investigate whether less conservative schedulers can reduce the makespan and what the cost is for improving throughput in this manner.

While we have considered only read and write accesses, our results can be extended to include also other access types. The lower bounds clearly hold, as they can still be proved using only reads and writes; Bimodal can be adapted to treat modifying and non-modifying operations separately.

Our study can be complemented by considering other performance measures, e.g., average response time of transactions.

Finally, while we have theoretically analyzed the behavior of Bimodal, it is important to see how it compares, through simulation, with prior transactional schedulers, e.g., [7,15,29,2,19].

Author's personal copy

Acknowledgments

We would like to thank Adi Suissa for many useful discussions, Richard M. Yoo for discussing ATS, and the referees for helpful comments.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, J. ACM 40 (1993) 873–890.
[2] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, I. Watson, Steal-on-abort: improving transactional memory performance through dynamic transaction reordering, in: HiPEAC, 2009, pp. 4–18.
[3] H. Attiya, L. Epstein, H. Shachnai, T. Tamir, Transactional contention management as a non-clairvoyant scheduling problem, Algorithmica 57 (1) (2010) 44–61.
[4] H. Attiya, A. Milani, Transactional scheduling for read-dominated workloads, in: OPODIS, 2009, pp. 3–17.
[5] U. Aydonat, T.S. Abdelrahman, Relaxed concurrency control in software transactional memory, IEEE Trans. Parallel Distrib. Syst. 23 (7) (2012) 1312–1325.
[6] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: DISC, 2006, pp. 194–208.
[7] S. Dolev, D. Hendler, A. Suissa, CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory, in: PODC, 2008, pp. 125–134.
[8] A. Dragojevic, R. Guerraoui, A.V. Singh, V. Singh, Preventing versus curing: avoiding conflicts in transactional memories, in: PODC, 2009, pp. 7–16.
[9] R. Guerraoui, M. Herlihy, M. Kapałka, B. Pochon, Robust contention management in software transactional memory, in: OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL, 2005.
[10] R. Guerraoui, M. Herlihy, B. Pochon, Polymorphic contention management, in: DISC, 2005, pp. 303–323.
[11] R. Guerraoui, M. Herlihy, B. Pochon, Toward a theory of transactional contention managers, in: PODC, 2005, pp. 258–264.
[12] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: PPoPP, 2008, pp. 175–184.
[13] R. Guerraoui, M. Kapalka, The semantics of progress in lock-based transactional memory, in: POPL, 2009, pp. 404–415.
[14] R. Guerraoui, M. Kapalka, J. Vitek, STMbench7: a benchmark for software transactional memory, in: EuroSys, 2007, pp. 315–324.
[15] T. Heber, D. Hendler, A. Suissa, On the impact of serializing contention management on STM performance, in: OPODIS, 2009, pp. 225–239.
[16] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: PODC, 2003, pp. 92–101.
[17] J. Kim, B. Ravindran, On transactional scheduling in distributed transactional memory systems, in: SSS, 2010, pp. 347–361.
[18] Y. Liu, M. Spear, Toxic transactions, in: 6th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT, 2011.
[19] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J.L. Lawall, G. Muller, Scheduling support for transactional memory contention management, in: PPOPP, 2010, pp. 79–90.
[20] J. Napper, L. Alvisi, Lock-free serializable transactions, Tech. Rep. TR-05-04, The University of Texas at Austin, 2005.
[21] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653.
[22] D. Perelman, I. Keidar, On avoiding spare aborts in transactional memory, in: SPAA, 2009, pp. 59–68.
[23] S.P. Rajeev Motwani, E. Torng, Non-clairvoyant scheduling, Theoret. Comput. Sci. 130 (1) (1994) 17–47.
[24] W.N. Scherer III, M.L. Scott, Advanced contention management for dynamic software transactional memory, in: PODC, 2005, pp. 240–248.
[25] J. Schneider, R. Wattenhofer, Bounds on contention management algorithms, Theoret. Comput. Sci. 412 (32) (2011) 4151–4160.
[26] G. Sharma, C. Busch, A competitive analysis for balanced transactional memory workloads, in: OPODIS, 2010, pp. 348–363.
[27] G. Sharma, C. Busch, Window-based greedy contention management for transactional memory: theory and practice, Distributed Computing 25 (3) (2012) 225–248.
[28] G. Sharma, B. Estrade, C. Busch, Window-based greedy contention management for transactional memory, in: DISC, 2010, pp. 64–78.
[29] R.M. Yoo, H.-H.S. Lee, Adaptive transaction scheduling for transactional memory systems, in: SPAA, 2008, pp. 169–178.

Hagit Attiya received the B.Sc. degree in Mathematics and Computer Science from the Hebrew University of Jerusalem, in 1981 and the M.Sc. and Ph.D. degrees in Computer Science from the Hebrew University of Jerusalem, in 1983 and 1987, respectively. She is presently a professor at the Department of Computer Science at the Technion, Israel Institute of Technology. Before joining the Technion, she was a post-doctoral research associate at the Laboratory for Computer Science at M.I.T. Her general research interests are in the area of distributed and parallel computation. She is the editor-in-chief of the journal Distributed Computing, and a fellow of the ACM.

Alessia Milani received the M.S. degree in Computer Engineering from Sapienza, University of Rome in 2003 and the Ph.D. degree in Computer Science jointly from Sapienza University of Rome and Université de Rennes I in 2007. She was postdoct at Universidad Rey Juan Carlos, Madrid, at Technion, Haifa and at LIP6, Paris in 2008, 2009 and 2010, respectively.

She has been an associate professor at ENSEIRB–MATMECA, Institut Polytechnique de Bordeaux since September 2010. Her research interests are in distributed and parallel computing. In particular, she is interested in the theoretical foundations of emerging computing systems: dynamic networks and multi-core machines. She is particularly interested in transactional memory, a new paradigm for concurrent programming.

Currently, she is on the management committee of the COST-action Transactional Memories: Foundations, Algorithms, Tools, and Applications (Euro-TM), and on the program committee of the 7th ACM SIGPLAN Workshop on Transactional Computing (Transact 2012).