Transactional Scheduling for Read-Dominated Workloads[⋆]

Hagit Attiya and Alessia Milani[⋆⋆]

Department of Computer Science, Technion, Haifa 32000, Israel
{hagit|alessia}@cs.technion.ac.il

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, called early-write transactions. It is evaluated by comparison with an optimal clairvoyant scheduler; we prove that BIMODAL achieves 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.

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. When aborted, a transaction is later *restarted* from its beginning.

Most existing transactional memory implementations (e.g. [3, 13]), guarantee consistency by making sure that whenever there is a conflict, i.e. two transactions access a same resource and at least one writes into it, one of the transactions involved is aborted.

[⋆] This research is partially supported by the *Israel Science Foundation* (grant number 953/06).

[⋆⋆] On leave from Sapienza, Universitá di Roma; supported in part by a fellowship from the Lady Davis Foundation and by a grant Progetto FIRB Italia- Israele RBIN047MH9.

We call this approach *conservative*. Taking a non-conservative approach, and ensuring progress while accurately avoiding consistency violation, seems to require complex data structures, e.g., as used in [16].

A major challenge is guaranteeing *progress* through a *transactional scheduler*, by 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.[1] 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 arrival times, duration, and (perhaps 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 [2, 8, 10]. It has been shown that the best competitive ratio achieved by simple transactional schedulers is $\Theta(s)$, where s is the number of resources [2]. These prior studies assumed *write-dominated* workloads, in which transactions need exclusive access to resources for most of their duration.

In transactional memory, however, the workloads are often *read-dominated* [12]: most of their duration, transactions do not need exclusive access to resources. This includes *read-only* transactions that only observe data and do not modify it, as well as *late-write* transactions, e.g., locating an item by searching a list and then inserting or deleting.

We extend the result in [2] by proving that every deterministic scheduler is $\Omega(s)$-competitive on read-dominated workloads, where s is the number of resources. Then, we prove that any non-clairvoyant scheduler which is conservative and thus too "coarse", is $\Omega(m)$ competitive for some workload containing late-write transactions, where m is the number of cores. (These results appear in Section 3.) This means that, for some workloads, these schedulers utilize at most one core, while an optimal, clairvoyant scheduler exploits the maximal parallelism on all m cores. This can be easily shown to be a tight bound, since at each time, a reasonable scheduler makes progress on at least one transaction.

Contemporary transactional schedulers, like CAR-STM [4], Adaptive Transaction Scheduling [20], and Steal-On-Abort [1], are conservative, thus they do not perform well under read-dominated workloads. These transactional schedulers have been proposed 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. Thus, they all end up serializing more than necessary in read-dominated workload, but also in what we call *bimodal* workload, i.e., a workload containing only early-write and read-only transactions. Actually, we show that there is a *bimodal* workload, for which these schedulers are at best $\Omega(m)$-competitive (Section 4).

These counter-examples motivate our BIMODAL scheduler, which has an $O(s)$ competitive ratio on bimodal workloads with equi-length transactions. BIMODAL alternates

[1] It is typically assumed that a transaction running solo, without conflicting accesses, commits with a correct result [13].

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. Due to the known lower bound [2], no algorithm can do better than $O(s)$ for bimodal traffic. BIMODAL also works when the workload is not bimodal, but being conservative it can only be trivially bound to have $O(m)$ competitive makespan when the workload contains late-write transactions.

Contention managers [13,19] were suggested as a mechanism for resolving conflicts and improving the performance of transactional memories. Several papers have recently suggested that having more control on the scheduling of transactions can reduce the amount of work wasted by aborted transactions, e.g., [1,4,14,20]. These schedulers use different mechanisms, in the user space or in the operating system level, but they all end up serializing more than necessary, in read-dominated workloads.

Very recently, Dragojevic et al. [6] have also investigated transactional scheduling. They have taken a complementary approach that tries 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 [4] and ATS [20], 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.

Early work on non-clairvoyant scheduling (starting with [15]) dealt with multiprocessing environments and did not address the issue of concurrency control. Moreover, they mostly assume that a preempted transaction resumes execution from the same point, and not restarted. For a more detailed discussion, see [2,6].

2 Preliminaries

2.1 Model

We consider a system of m identical *cores* with a finite set of shared data items $\{i_1, \ldots, i_s\}$. The system has to execute a *workload*, which is a finite partially-ordered set of transactions $\Gamma = \{T_1, T_2, \ldots\}$; the partial order among transactions is induced by their arrival times. Each transaction is a sequence of operations on the shared data items; for simplicity, we assume the operations are *read* and *write*. A transaction that only reads data items is called *read-only*; otherwise, it is a *writing* transaction.

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

The sequence of operations in a transaction must be atomic: if any of the operations takes place, they all do, and that if they do, they appear to other threads to do so atomically, as one indivisible operation, in the order specified by the transaction. Formally, this is captured by a classical consistency condition like *serializability* [17] or the stronger *opacity* condition [11].

Two overlapping transactions T_1 and T_2 have a *conflict* if T_1 reads a data item X and T_2 executes a write access to X while T_1 is still pending, or T_1 executed a write access to X and T_2 accesses X while T_1 is still pending. 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. We discuss this issue further in Section 3.

2.2 Transactional Schedulers and Measures

The set of data items accessed by a transaction, i.e., its data set, 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 wishes to modify the data item or just read it.

Each core is associated with a list of transactions (possibly the same for all cores) available to be executed. Transactions are placed in the cores' list according to a strategy, called *insertion policy*. Once a core is not executing a transaction, it selects, according to a *selection policy*, a transaction in the list and starts to execute it. The selection policy determines when an aborted transaction is restarted, in an attempt to avoid repeated conflicts. A scheduler is defined by its insertion and selection policies.

Definition 1 (Makespan). *Given scheduler A and a workload Γ, makespan$_A(\Gamma)$ is the time A needs to complete all the transactions in Γ.*

Definition 2 (Competitive ratio). *The* competitive ratio *of a scheduler A for a workload Γ, is $\frac{\text{makespan}_A(\Gamma)}{\text{makespan}_{Opt}(\Gamma)}$, where* OPT *is the optimal, clairvoyant scheduler that has access to all the characteristics of the workload.*
The competitive ratio *of A is the maximum, over all workloads Γ, of the competitive ratio of A on Γ.*

We concentrate on "reasonable" schedulers, i.e., ones that utilize at least one core at each time unit for "productive" work: a scheduler is *effective* if in every time unit, some transaction invocation that eventually commits executes a unit of work (if there are any pending transactions).

We associate a real number $\tau_i > 0$ with each transaction T_i, which is the execution time of T_i when it runs uninterrupted to completion.

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

Proof. The proof immediately follows from the fact that for any 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 cores are continuously utilized, is $\frac{1}{m} \sum_{T_i \in \Gamma} \tau_i$. □

We say that transaction T_i is *early-write* if the time from its first write access until its completion, denoted ω_i, is at least half of its duration (any other constant can be used, in fact). Formally, $2\omega_i > \tau_i$.

We pick a small constant $\alpha > 0$ and say that a transaction T_i is *late-write* if $\omega_i \leq \alpha\tau_i$, i.e., the transaction needs exclusive access to resources during at most an α-fraction of its duration. For a *read-only* transaction, $\omega_i = 0$.

A workload Γ is *bimodal* if it contains only early-write and read-only transactions; said otherwise, if a transaction writes, then it does so early in its execution.

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 *late-write workloads*, including only late-write transactions. This complements the lower bound proved in [2], for workloads that include only early-write transactions.

We use R_h, W_h to denote (respectively) a read and a write access to data item i_h.

Theorem 2. *There is a late-write workload Γ, such that every deterministic scheduler A is $\Omega(s)$-competitive on Γ.*

Proof. To prove our result we first consider the scheduler A to be *work-conserving*, i.e., it always runs a maximal set of non conflicting transactions [2], and then show how to remove this assumption.

Assume that s is even and let $q = \frac{s}{2}$. The proof uses an execution of $q^2 = \frac{s^2}{4}$ equal-length transactions, described in Table 1. Since transactions have all the same duration, we normalize it to 1.

	1	2	$\ldots$	q
1	$[R_1, \ldots, R_q, R_{q+1}, W_{q+1}]$	$[R_1, \ldots, R_q, R_{q+1}, W_{q+1}]$	$\ldots$	$[R_1, \ldots, R_q, R_{q+1}, W_{q+1}]$
2	$[R_1, \ldots, R_q, R_{q+2}, W_{q+2}]$	$[R_1, \ldots, R_q, R_{q+2}, W_{q+2}]$	$\ldots$	$[R_1, \ldots, R_q, R_{q+2}, W_{q+2}]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$[R_1, \ldots, R_q, R_{q+i}, W_{q+i}]$	$[R_1, \ldots, R_q, R_{q+i}, W_{q+i}]$	$\ldots$	$[R_1, \ldots, R_q, R_{q+i}, W_{q+i}]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
q	$[R_1, \ldots, R_q, R_{2q}, W_{2q}]$	$[R_1, \ldots, R_q, R_{2q}, W_{2q}]$	$\ldots$	$[R_1, \ldots, R_q, R_{2q}, W_{2q}]$

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

The data items $\{i_1, \ldots, i_s\}$ are divided into two disjoint sets, $D_1 = \{i_1, \ldots, i_q\}$ and $D_2 = \{i_{q+1}, i_{q+2}, \ldots, i_{2q}\}$. Each transaction reads q data items in D_1 and reads and writes to one data item in D_2. For every $i_j \in D_2$, q transactions read and write to i_j (the ones in row $j - q$ in Table 1).

All transactions are released and available at time $t_0 = 0$. The scheduler A knows only the first data item requested and if it is accessed for read or write. The data item to be read and then written is decided by an adversary during the execution of the algorithm in a way that forces many transactions to abort. Since the first access of all transactions is a read and A is work conserving, A executes all q^2 transactions.

Let time t_1 be the time at which all q^2 transactions have executed their read access to the data item they will then write, but none of them has already attempt to write. It is simple to see that transactions can be scheduled for this to happen. Then, at some point after t_1 all transactions attempt to write but only q of such transactions can commit (the transactions in a single column of Table 1). Otherwise, serializability is violated. All other transactions abort.

When restarted, all of them write to the same data item i_1, i.e., $[R_1,\ldots,R_q,R_{q+1},W_1]$. This implies that after the first q transactions commit (any set in a column), having run in parallel, the remaining $q^2 - q$ transactions end up being executed serially (i.e., even though they are run in parallel only one of them can commit at each time). So, the makespan of the on-line algorithm is $1 + q^2 - q$.

On the other hand, an optimal scheduler OPT executes the workload as follows: at each time τ_i with $i \in \{0, \ldots, q-1\}$, OPT will execute the set of transactions depicted in column $i+1$ in Table 1. Thus, OPT achieves makespan q. Therefore, the competitive ratio of any work-conserving algorithm is $\frac{1+q^2-q}{q} = \Omega(s)$.

As in [2] to remove the initial assumption that the scheduler is work conserving, we modify the requirement of data items in the following way: if a transaction belonging to Γ is executed after time q then it requests to write into i_1 as done in the above proof when a transaction is restarted. Otherwise, it requests the data items as in Table 1. Thus the online scheduler will end up serializing all transactions executed after time q.

On the other hand, the optimal offline scheduler is not affected by the above change in data items requirement since it executes all transactions by time q. The claim follows. □

Next, we prove that when the scheduler is too "coarse" and enforces consistency by aborting one conflicting transaction whenever there is a conflict, even if this conflict does not violate serializability, the makespan it guarantees is even less competitive. We remark that all prior competitive results [2, 8, 10] also assume that the scheduler is conservative. Formally,

Definition 3. *A scheduler A is* conservative *if it aborts at least one transaction in every conflict.*

Note that prominent transactional memory implementations (e.g., [3, 13]) are conservative.

Theorem 3. *There is a late-write workload Γ, with $\alpha < \frac{1}{m}$, such that every deterministic conservative scheduler A has $\Omega(m)$-competitive makespan on Γ.*

Proof. Consider a workload Γ with m late-write transactions, all available at time $t = 0$. Each transaction $T \in \Gamma$ first reads items $\{i_1, i_2, \ldots i_{s-1}\}$, and then modifies item i_s, i.e., $T_i = [R_1, \ldots, R_{s-1}, W_s]$, for every $i \in \{1, \ldots, m\}$. All transactions have the same duration d, and they do not modify their data set when running at different times.

The scheduler A will immediately execute all transactions. At time $d - \epsilon$ all transactions will attempt to write into i_s. Since A is conservative, only one of them commits, while the remaining $m-1$ transactions abort. Aborted transactions will be restarted later, and each transaction will write into i_1 instead of i_s. Thus, all the remaining transactions have to be executed serially in order not to violate serializability. Since A executes all transactions in a serial manner, makespan$_A(\Gamma)$=$\sum_{i=1}^{m} d_i = md$.

On the other hand, the optimal scheduler OPT has complete information on the set of transactions, and in particular, OPT knows that at time $d-\epsilon$, each transaction attempts to write to i_s. Thus, OPT delays the execution of the transactions so that conflicts do not happen: at time $t_0 = 0$, only transaction T_1 is executed; for every $i \in \{2, \ldots, m\}$, T_i starts at time $t + (i-1)\epsilon$, where $\epsilon = \alpha d$. (See Figure 1.)

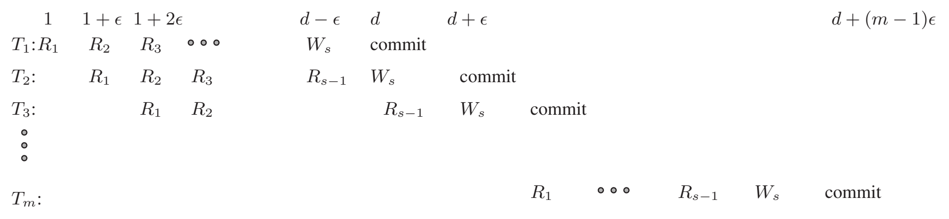

Fig. 1. The execution used in the proof of Theorem 3.

Thus, makespan$_{Opt}(\Gamma)$=$d + (m-1)\epsilon$, and the competitive ratio is $\frac{md}{d+(m-1)d\alpha} > \frac{m}{1+\alpha \cdot m} \geq \frac{m}{2}$. □

In fact, the makespan is not competitive even relative to a clairvoyant *online* scheduler [6], 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 accesses.

As formally proved in [6], knowing at release time, the data items a transaction will access, for transactions which do not change their data sets during the execution, facilitates the transactional scheduler execution and greatly improves performance.

4 Dealing with Read-Only Transactions: Motivating Example

Several recent transactional schedulers [1, 4, 14, 20] attempt to reduce the overhead of transactional memory, by serializing conflicting transactions. Unfortunately, these schedulers are conservative and so, 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 and ATS have a competitive ratio (relative an optimal offline scheduler) that is at least $\Omega(m)$. This is because both CAR-STM and ATS do not ensure the so-called *list scheduler property* [7], i.e., no thread is waiting to execute if the resource it needs are available, and may cause a transaction to wait although 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) [1], 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 yields bimodal workloads in which the makespan of Steal-on-Abort is $\Omega(m)$ competitive, as we show below. (Steal-on-abort [1], as well as the other transactional schedulers [4, 14, 20], are effective, and hence they are $O(m)$-competitive, by Theorem 1.)

The Steal-On-Abort (SoA) scheduler: Application threads submit transactions to a transactional threads pool. 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.

When two running transactions T and T' conflict, the contention manager policy decides which 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 top 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 because a new conflict happen 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*. Here we describe a worst case 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, when T completes. Similar scenarios can be shown for the other strategies.

The SoA scheduler does not specify any policy to manage conflicts. In [1], 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.[2] 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 [9] .

Moreover, while several contention management policies have been proposed in the literature [10, 19], none of them, except *Greedy* [10], has nontrivial provable properties.

Thus, we consider the SoA scheduler with a contention management policy based on timestamps, like Greedy [10] or Timestamp [19]. These policies do not require costly data structures, like the $Polka$ policy. Our choice also provides a fair comparison with *CAR-STM*, which embeds a contention manager based on timestamps.

[2] 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 [19].

Theorem 4. *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-length transactions, two writing transactions and $2m - 3$ read-only transactions, depicted in Table 2. 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, $m-1$ read-only transactions [R_2,R_1,R_3] become available. Let S_1 denote this set of read-only transactions.

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

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

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. We consider U_2 to discover the conflict and to abort all the read-only transaction in S_2. Actually, if U_2 arrives immediately before the read-only transactions, it has a bigger 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.

In the following we repeat the above scenario, until all transactions commit. In particular, for every $i \in \{3, \dots m\}$, we have that immediately before time t_i, there are $m-i+1$ read-only 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 $i - 1$. All the remaining threads have no transaction in their work queues. 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 built, the way each thread steals the transactions from the work queues of other threads is governed by a uniformly random distribution as requested by the Steal on Abort work-steal strategy.

Thus, makespan$_{SoA}(\Gamma)$=$m + 2$. 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 $\frac{m+2}{4}$. □

In the following section, we present a conservative scheduler, called BIMODAL, which is $O(s)$-competitive for bimodal workloads. BIMODAL embeds a simple contention management policy utilizing timestamps.

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

Table 2. Steal-On-Abort with steal-tail strategy: illustration for Theorem 4. 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 dequeue 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 dequeue, the bottom line is empty.

5 The BIMODAL Scheduler

The BIMODAL scheduler architecture is similar to CAR-STM [4]: each core is associated with a work dequeue (double-ended queue), where a *transactional dispatcher* enqueues arriving transactions. BIMODAL also maintains a fifo queue, called RO-queue, shared by all cores to enqueue transactions which abort before executing their first writing operation and that are predicted to be read-only transactions.

Transactions are executed as they are available unless the system is overloaded. BIMODAL requires visible reads in order for a conflict to be detected as soon as possible.

Once two transactions 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, BIMODAL moves it to the work dequeue of the conflicting transaction; otherwise, it is enqueued in the RO-queue.

Specifically, the contention manager, embedded in BIMODAL, decides which transaction to abort in a conflict, according to two levels of priority:

1. In a conflict between two writing transactions, the contention manager aborts the newer transaction. Towards this goal, a transaction is assigned a timestamp when it starts, which it retains even when it aborts, as in the greedy contention manager [10].
2. To handle a conflict between a writing transaction and a read-only transaction, 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*.

Below, we detail the algorithm and we provide its competitive analysis.

5.1 Detailed Description of the BIMODAL Scheduler

Transactions are assigned in round-robin to the work dequeues of the cores (inserted at their tail), starting from cores whose work dequeue is empty; initially, all work dequeues are empty.

At each time, the system is in a given epoch associated with a pair $(mode, ID)$, where $mode \in \{Reading, Writing\}$ is the type of epoch and ID is a monotonically increasing integer that uniquely identifies the epoch. A shared variable ξ stores the pair corresponding to the current epoch and it is initially set to $(Writing, 0)$.

When in a writing epoch i, the system moves to a reading epoch $i+1$, i.e., $\xi = (Reading, i+1)$, if there are m transactions in the RO-queue or every work dequeue is empty. Analogously, if during a reading epoch $i+1$, m transactions have been dequeued from the RO-queue or this queue is empty, the system enters writing epoch $i+2$, and so on. A process in the system, called *ξ-manager*, is responsible to managing epoch evolution and updating the shared variable ξ. The ξ-manager checks if the above conditions are verified and sets the variable ξ in a single atomic operation (e.g., using a Read-Modify-Write primitive).

A transaction T that starts in the i-th epoch, is associated with epoch i up to the time it either commits or aborts. An aborted transaction may be associated to a new epoch when restarted. Moreover, 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 *overlap*. To manage conflicts between transactions associated with different epochs, we give higher priority to the transaction in the earlier epoch. Specifically, if a core executes a transaction T belonging to the current epoch i while some core is still executing a transaction $T^{'}$ in epoch $i-1$, and T and $T^{'}$ have a conflict, T is aborted and immediately restarted.

Writing epochs. The algorithm starts in a *writing epoch*. During a writing epoch, each core selects a transaction from its work dequeue (if it is not empty) and executes it. During this epoch:

1. A read-only transaction that conflicts with a writing transaction is aborted and enqueued in the RO-queue. We may have a *false positive*, i.e., a writing transaction T, wrongly considered to be a read-only transaction and enqueued in the RO-queue, because it has a conflict before invoking its first writing access.
2. If there is a conflict between two writing transactions T_1 and T_2, and T_2 has lower priority than T_1, then T_2 is inserted at the head of the work dequeue of T_1. (As in the *permanent serializing* contention manager of CAR-STM.)

Reading epochs. A *reading epoch* starts when the RO-queue contains m transactions, or the work dequeues of all cores are empty. The latter option ensures that no transaction in the RO-queue is indefinitely, waiting to be executed.

During a reading epoch, each core takes a transaction from the RO-queue and executes it. The reading epoch ends when m transactions have been dequeued from the RO-queue or this latter is empty. Conflicts may occur during a reading epoch, due to false positives or because epochs overlap. If there is a conflict between a read-only transaction and a false positive, the writing transaction is aborted. If the conflict is between two writing transactions (two false positives), then one aborts, and the other transaction simply continues its execution; as in a writing epoch, the decisions are based on their priority. Once aborted, a false positive is enqueued in the head of the work dequeue of the core where it executed.

5.2 Analysis of the BIMODAL scheduler

We first bound (from below) the makespan that can be achieved by an optimal conservative scheduler.

Theorem 5. *For every workload Γ, the makespan of Γ under an optimal, conservative offline scheduler* OPT *satisfies makespan$_{Opt}(\Gamma) \geq \max\{\frac{\sum \omega_i}{s}, \frac{\sum \tau_i}{m}\}$.*

Proof. There are m cores, and hence, the optimal scheduler cannot execute more than m transactions in each time unit; therefore, makespan$_{Opt}(\Gamma) \geq \frac{\sum \tau_i}{m}$.

For each transaction T_i in Γ with $\omega_i \neq 0$, let X_{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 $X_{f_i} = X_{f_j}$, 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 makespan$_{Opt}(\Gamma) \geq \frac{\sum \omega_i}{s}$. □

We analyze BIMODAL assuming all transactions have the same duration.

A key observation is that if a false positive is enqueued in the RO-queue and executed during a *reading epoch* because it is falsely considered to be a read-only transaction, either it completes successfully without encountering conflicts or it is aborted and treated as a writing transaction once restarted.

Theorem 6. BIMODAL *is $O(s)$-competitive for bimodal workloads, in which for every writing transaction T_i, $2\omega_i \geq \tau_i$.*

Proof. Consider the scheduling of a bimodal workload Γ under BIMODAL. Let t_k be the starting time of the last reading epoch after all the work deques of cores are empty, and such that some transactions arrive after t_k.

At time t_k, no transactions are available in the work queues of any core, and hence, no matter what the optimal scheduler OPT does, its makespan is at least t_k.

Let Γ_k be the set of transactions that arrive after time t_k, and let $n_k = |\Gamma_k|$. Since at time t_k, OPT does not schedule any transaction, it will schedule new transactions to execute as they arrive. On the other hand, BIMODAL may delay the execution of new available transactions because the cores are executing the transactions in the RO-queue (if any). Since RO-queue has less than m transactions, this will take at most τ time units, where τ is the duration of a transaction (the same for all transactions).

By Theorem 5,

$$\text{Makespan}_{Opt}(\Gamma_k) \geq \frac{1}{2}(\frac{\sum_{i=1}^{n_k} \omega_i}{s} + \frac{\sum_{i=1}^{n_k} \tau_i}{m}) \ ,$$

and therefore,

$$\text{Makespan}_{Opt}(\Gamma) \geq t_k + \frac{1}{2}(\frac{\sum_{i=1}^{n_k} \omega_i}{s} + \frac{\sum_{i=1}^{n_k} \tau_i}{m}) \ .$$

On the other hand, we have that

$$\text{Makespan}_{Bimodal}(\Gamma) \leq t_k + \tau + \sum_{i=1}^{n_k} 4\omega_i + \frac{1}{m}\sum_{i=1}^{n_k} \tau_i \ .$$

The penultimate term holds because $2\omega_i \geq \tau_i$, for every writing transaction $T_i \in \Gamma_k$, and taking into account the impact of false positives during reading epochs. In fact, a writing transaction T may conflict only once during a reading epoch, because when restarted T will be treated as a writing transaction. This is just as if T is executed during a writing epoch with its duration doubled, to account for the time spent for the execution of the read-only transaction that aborted T (if there is one). The last term holds since all transactions have the same duration.

Therefore, the competitive ratio is

$$\frac{\text{Makespan}_{Bimodal}(\Gamma)}{\text{Makespan}_{Opt}(\Gamma)} \leq \frac{t_k + \tau + \sum_{i=1}^{n_k} 4\omega_i + \frac{1}{m}\sum_{i=1}^{n_k} \tau_i}{t_k + \frac{1}{2}(\frac{\sum_{i=1}^{n_k} \omega_i}{s} + \frac{\sum_{i=1}^{n_k} \tau_i}{m})} \ ,$$

which can be shown to be in $O(s)$.

Note that if t_k does not exist, we can take t_k to be the time immediately before the first transaction in Γ is available, and repeat the reasoning with $t_k = 0$ and $\Gamma_k = \Gamma$. □

6 Discussion

We have studied the competitive ratio achieved by non-clairvoyant transactional schedulers on read-dominated workloads. The BIMODAL transactional scheduler, presented in this paper, 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.

This last result assumes that the scheduler is conservative, namely, it aborts at least one transaction involved in a conflict. This is the approach advocated in [13] 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 is the cost of improving parallelism. Keidar and Perelman [18] prove that contention managers that abort a transaction only when it is necessary to ensure correctness have local computation that is NP-complete; however, it is not clear whether being less accurate in ensuring consistency can be done more efficiently.

Our study should be completed by considering other performance measures, e.g., the average response time of transactions.

The contention manager embedded in SwissTM [5] is also bimodal, distinguishing between *short* and *long* transactions, and it would be interesting to see whether our analysis techniques can be applied to it.

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., [1,4,14,20].

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

References

1. M. Ansari, M. Lujn, C. Kotselidis, K. Jarvis, C.C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC 2009*, pages 4–18.
2. Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC 2006*, pages 308–315.
3. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC 06*, pages 194–208.
4. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC 2008*, pages 125–134.
5. Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI*, pages 155–165, 2009.
6. Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *PODC 2009*, pages 7–16.
7. M.R. Garey and R.L. Graham. Bound for multiprocessor scheduling with resource constraints. *SIAM Journal Computing*, 4:187–200, 1975.

8. Rachid Guerraoui, Maurice Herlihy, Michał Kapałka, and Bastian Pochon. Robust contention management in software transactional memory. In *OOPSLA'05 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05)*, October 2005.
9. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC 2005*, pages 303–323.
10. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC 2005*, pages 258–264.
11. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP 2008*, pages 175–184.
12. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys 2007*, pages 315–324.
13. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC 2003*, pages 92–101.
14. Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, and Gilles Muller. Transaction activation: Scheduling support for transactional memory. Technical Report 6807, INRIA, jan 2009.
15. Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47, 1994.
16. Jeff Napper and Lorenzo Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin, 2005.
17. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
18. Dmitri Perelman and Idit Keidar. On avoiding spare aborts in transactional memory. In *SPAA 2009*, pages 59–68.
19. William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC 2005*, pages 240–248.
20. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA 2008*, pages 169–178.