SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 16, No. 1, February 2019, 85-104

UDC: 004.425.032.24:004.43 **DOI: https://doi.org/10.2298/SJEE1901085P**

Online Algorithms for Scheduling Transactions on Python Software Transactional Memory

Marko Popović[1], Branislav Kordić[1], Miroslav Popović[1], Ilija Bašičević[1]

Abstract: Designing online transaction scheduling algorithms is challenging because one needs to reconcile three opposing requirements: (i) they should be fast, (ii) they should minimize makespan and maximize throughput, and (iii) they should produce conflict-free transaction schedules. In this paper we present four online transaction scheduling algorithms, namely, RR, ETLB, AC, and AAC algorithm, we prove their correctness and time bounds, and we conduct a theoretical analysis of the transaction schedules they produce, using three different workloads (RDW, CFW, and WDW). Finally, we compare various features of the four algorithms. The results are as expected, as we go from RR, over ETLB and AC, to AAC algorithms, the quality of the resulting schedules increases at the cost of increase of algorithm's time complexity.

Keywords: Parallel programming, Transactional memory, Transaction scheduling, Algorithms, Time complexity.

1 Introduction

Transaction Memory (TM) is a mechanism based on the concept of database transactions, which replaces convectional locks with transactions on multicores [1]. Even though the theoretical foundation of TM is already mature [2], TM remains an open arena for ongoing research. Leading companies in industry, such as IBM and Intel, already provide synchronization based on TM on their multicores, e.g. IBM Blue Gene/Q, zEnterprise, EC12, Power 8 and Intel Haswell. Besides hardware TMs, researchers have developed many Software TMs (STMs) and Hybrid TMs (HTMs), with different APIs and semantics. Still, the main open issue for all TMs is their performance, which may be significantly reduced in case of high contention among transactions.

Transaction scheduling (a.k.a. contention management) is rather well studied and widespread in the literature related to multicore systems. In addition to a few scheduling algorithms with their proven upper and lower bounds, and

[1]University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia; E-mails: marko.popovic@rt-rk.uns.ac.rs; branislav.kordic@rt-rk.uns.ac.rs; miroslav.popovic@rt-rk.uns.ac.rs; ilija.basicevic@rt-rk.uns.ac.rs

some impossible results given in [3 – 5], there are algorithms which are evaluated entirely by experiments, e.g. [6, 7]. Still, besides these and other contributions, transaction scheduling is a subject of active research worldwide. In this paper we deal with online transaction scheduling algorithms, which are especially hard to design.

Designing a good online scheduling algorithm for STM is a challenge because it needs to fulfill the following opposing requirements: (i) it should be fast, (ii) it should minimize makespan and maximize throughput, and (iii) it should produce conflict-free transaction schedules. As a response to this challenge, in our previous work [8, 9] we developed four algorithms for scheduling transactions on Python STM (PSTM) [10], namely: (i) Round Robin (RR) algorithm, (ii) Execution Time based Load Balancing (ETLB) algorithm, Avoid Conflicts (AC) algorithm, and Advanced Avoid Conflicts (AAC) algorithm.

In this paper, we briefly present these four online transaction scheduling algorithms, prove their correctness and time bounds, and conduct a theoretical analysis of the transaction schedules they produce, using three different workloads (RDW, CFW, and WDW). Finally, we compare these algorithms' four features, namely: the time complexity, the quality of resulting transaction schedules, the average speedup over RR algorithm, and the number of aborts (the last two features are experimentally evaluated in a complete workload execution). All the results are as expected; as we go from RR, over ETLB and AC, to AAC algorithms, the quality of the resulting schedules increases at the cost of increase of the algorithm's time complexity.

We would like to emphasize that the main aim of this paper is to provide a theoretical analysis of scheduling algorithms and transaction schedules. Because of the limited space, we provide a brief overview of the experimental evaluation made in our previous research in Section 4, and an interested reader may find all the details in [9].

2 The Scheduler Architecture and Algorithms

In this section, we present a brief overview of the transaction scheduler architecture and the four scheduling algorithms developed in our previous research: RR algorithm, ETLB algorithm, AC algorithm, and AAC algorithm [8, 9]. Additionally, in this paper we prove their correctness and time bounds.

2.1 The architecture

Let T be an application transaction, $T = (f, V)$, where f is the function to be executed by T and V defines t-variables used by T,$V = (R, W)$, where R is a set of t-variables that T only reads and W is a set of t-variables that T writes (and reads).

The architecture of the system comprises a process scheduler S and n worker processes W_i, $i = 1,\ldots,n$, which are assigned to n available processors (or cores). An application adds new transactions to the input queue Q_{in}. The scheduler S works in phases, where every phase consists of two rounds.

During the first round, *S* repeats the following steps while Q_{in} is not empty or after it scheduled *K* transactions (where *K* is a parameter that may be tuned in accordance to the application specific needs): (1) dequeue a transaction T from Q_{in}, (2) call the configured scheduling algorithm, which in its turn returns an index i of the worker W_i to which T is assigned, and (3) schedule T by queueing it to the queue Q_i which corresponds to the worker W_i.

During the second round, S waits for all the scheduled transactions to get executed by the workers. S does this by waiting for *done* signals at the worker output queues, D_i, where, $i = 1,\ldots,n$.

While S waits, every W_i executes the transactions, one by one, from Q_i. W_i executes each $T_j = (f_j, V)$ by calling the function f_j. If the transaction T_j aborts, W_i enqueues T_j to Q_{in}. Otherwise, it takes the next transaction from Q_i. When the input queue Q_i, becomes empty, W_i enqueues the signal *done* to its output queue D_i. After receiving signals *done* from all the workers, *S* switches back to the first round.

As explained so far, it looks like this architecture rests directly upon hardware. Although this abstraction helps when dealing with scheduling algorithms, in realty our transaction scheduler is implemented as an application level scheduler that rests upon Python runtime, which in turn rests upon the local OS. Because of this, the real execution timing may vary slightly from the ideal timing (without interference from OS and Python runtime). As it happens, these variations may be seen as drifts of transaction release times and durations.

Actually, the main motive to introduce the scheduling phases with two rounds was to minimize the interference between the application level scheduler (together with OS and Python runtime) and the scheduled transactions during the second round, especially on multicores with smaller number of cores.

2.2 RR algorithm

The goal of RR algorithm is to provide load-balancing of the input workload to the available workers. Since there is no information about input transactions, it optimistically assumes that they all have the same duration and

that they are conflict-free. Thus it simply uses the traditional round-robin scheduling to achieve its goal.

The implementation of RR algorithm uses the variables n and i. The algorithm calculates the new value of i by incrementing it with module n, and returns the previous value of i. The scheduler S then uses the returned value i to schedule the next transaction.

Theorem 1. RR algorithm evenly distributes input transactions to the n workers within time $\Theta(1)$ per each transaction.

Proof. RR algorithm correctness follows from the fact that incrementing i with modulo n yields uniform distribution in the interval $[0, n-1]$. Since RR algorithm performs a constant number of steps per transaction (modulo increment), its tight bound on execution time per transaction is $\Theta(1)$.

2.3 ETLB algorithm

ETLB algorithm is a greedy scheduling algorithm, which uses the method for estimating execution times based on log-normal distribution [11], and a simple approach of machine learning. The method works on a sliding window of samples of execution times, which are taken during the execution of the application. Each transaction type has its own sliding window. For the next sample, the method updates the sliding window of samples and related internal data, so that it can estimate the value of the next transaction execution time, when it is necessary.

The ETLB algorithm assumes that the application will perform the initial calibration by measuring the execution time of every transaction type, and filling the sliding window with the initial samples. Doing the initial calibration, the method learns the initial behavior of all the transaction types, so it can immediately provide the execution time estimation, when it is requested by worker W_i.

During a normal execution, the workers measure the execution times of the transactions, and update the appropriate sliding windows with new samples. Thus ETLB represents an adaptive algorithm. As the transaction execution time changes over time (because of the changes in environment), the estimated time follows these changes.

Let L_i be the workload (cumulative execution time), assigned to the worker W_i, where $i = 1,\ldots,n$. The ETLB algorithm uses the current values of L_i and the estimated execution time of the next transaction t_e, in order to schedule the next transaction to the worker with the least load. In that way, the ETLB algorithm tries to minimize the makespan and maximize the throughput.

The implementation of the ETLB algorithm uses the variables n, i, t_e, and L_i, where $i = 1,\ldots,n$. The algorithm finds the index i of L_i with the minimal value, adds t_e to L_i, and returns the value i. The scheduler S then uses the returned value of i and schedules the next transaction.

Theorem 2. ETLB algorithm is an online greedy algorithm that schedules the next input transaction to the least loaded worker within time $\Theta(n)$, where n is the number of workers.

Proof. ETLB algorithm correctness trivially follows from the correctness of the two Python primitives: (i) the primitive min for finding the minimal element in the list L_i, $i = 1,\ldots,n$, and (ii) the primitive index for finding the index i of that element. Since the tight bound on execution time for the composition of primitives index and min is $\Theta(n)$, and the tight bound on transaction execution time estimation is $\Theta(1)$, the overall tight bound for ETLB algorithm is $\Theta(n)$.

2.4 AC algorithm

AC algorithm can be viewed as an extended ETLB algorithm. AC algorithm, like ETLB algorithm uses the method for determining the transaction execution times. Unlike ETLB algorithm, AC algorithm checks for a possible conflict between the next transaction to be scheduled and the already scheduled transactions. Note that here we present and analyze the simplified version of AC algorithm, which does not use the notion of worker load unbalance.

Let F be a scheduled transaction, $F = (t_b, t_e, V)$, where t_b is the F's start time, t_e is the F's end time, and $V = (R, W)$. Let $F_1 = (t_{b1}, t_{e1}, V_1)$ be the already scheduled transaction and $F_2 = (t_{b1}, t_{e2}, V_2)$ be the next transaction from Q_{in}.

In this paper we assume that every transaction F_i uses all of its t-variables during its entire life cycle $[t_{bi}, t_{ei}]$. So, there is no need to maintain the access times for individual t-variables.

Generally, when F_i and F_j execute on the optimistic STM, with the lazy conflict detection, like the PSTM, we say that F_i attacks F_j, if their execution overlaps in time and if F_i finishes before F_j, because F_i will get committed, while F_j will get aborted. Theoretically, F_i and F_j can attack each another, if they finish at the same time, but in reality the PSTM serializes the requests for commit, so one of these transactions will get committed and the other will get aborted.

The transactions F_1 and F_2 are in conflict if: (i) F_1 attacks F_2 or vice versa, and (ii) if the Bernstein conditions [12] are not satisfied (i.e. there is a data race between F_1 and F_2). Precisely, F_1 attacks F_2 if:

$$t_{b2} < t_{e1} \leq t_{e2}.$$

F_2 attacks F_1 if:

$$t_{b1} < t_{e2} \leq t_{e1}.$$

The Bernstein conditions for F_1 and F_2 are:

$$R_1 \cap W_2 = \{\},$$

$$R_2 \cap W_1 = \{\},$$

$$W_1 \cap W_2 = \{\}.$$

AC algorithm uses the following scheduling strategy. Like ETLB algorithm, it schedules the next transaction to the least loaded worker, if the resulting schedule is conflict-free. But, if scheduling the next transaction T to the least loaded worker would lead to a conflict with the already scheduled transactions, AC algorithm immediately backs-off and conservatively schedules T to the most loaded worker (because this slot in the schedule is definitely conflict-free). The advantage of this strategy is that it is relatively fast. The disadvantage is that by taking the conservative back-off, AC algorithm may miss a chance to make a better schedule (with shorter makespan), as we show in Section 3.

The implementation of AC algorithms uses the variables that are already introduced. Here, we introduce the duration of the transaction, t_d, as well as queues of scheduled transactions per worker, QF_i. The algorithm executes the next steps in order to calculate the required index i:

1. Find the indexes i_{min} and i_{max}, which correspond to L_{min} and L_{max}, respectively.
2. Set T_2 to the next transaction from Q_{in}.
3. Determine the estimated execution time for the transaction of type f_2 and assigns it to the value t_{d2}.
4. Set $i = i_{min}$, $F_2 = (L_{min}, L_{min} + t_{d2}, V_2)$
5. Inside the loop, for every already scheduled F_1, check if there is a conflict between F_1 and F_2, and break the loop on the first detected conflict.

6. If a conflict is detected in step 5, set $i = i_{max}$, $F_2 = (L_{max}, L_{min} + t_{d2}, V_2)$
7. Enqueue F_2 at the end of QF_i.
8. Return i.

The scheduler uses the return value of i to schedule the current transaction.

Theorem 3. AC algorithm is an online greedy algorithm with conservative conflict avoidance that schedules the next input transaction T to the least loaded worker, if this does not cause a conflict; otherwise, it schedules T to the most loaded worker. Its time complexity is $O(nm^2)$, where n is the number of workers and m is the number of t-variables, if all the transactions read m t-variables and write (possibly different) m t-variables.

Proof. AC algorithm correctness trivially follows from the correctness of Python primitives: min, max, and index, as well as the primitives for checking attacks and Bernstein conditions among pairs of transactions. Time complexity of all the steps of AC algorithm, except step 5, is $\Theta(n)$. In the worst case, the loop in step 5 has $(n-1)$ passes. We assume QF_i lists are short (thus may be checked in constant time). Time complexity for checking attacks is $\Theta(1)$, and for checking Bernstein's conditions is $\Theta(m^2)$, because finding an intersection between two sets with m elements takes $\Theta(m^2)$ steps (since we represent sets as Python lists, see [13]). So, the upper bound on the execution time for step 5 is $O(nm^2)$. Since $O(nm^2)$ dominates on $\Theta(n)$, the upper bound for the complete algorithm is $O(nm^2)$.

Remark 1. In our future work we may use hash sets instead of lists, in order to reduce set intersection bound to $\Theta(m)$, and the overall bound to $O(nm)$.

2.5 AAC algorithm

In this section, we describe AAC algorithm, which is a natural extension of AC algorithm. The main disadvantage of the conservative scheduling strategy used by AC algorithm is that if there are more than two workers, a worker that has the load between the minimal and maximal will not be employed, even though the new transaction can be scheduled to it, without conflicts with the already scheduled transactions.

The goal of AAC algorithm is to overcome this disadvantage of AC algorithm. If the scheduling of the next transaction T to the least loaded worker would lead to a conflict, AAC algorithm, in contrast to AC algorithm, does not give up immediately, and does not schedule T to the most loaded worker.

Instead it checks the workers, one by one, from the least loaded worker to the most loaded worker. When it finds the worker to which it can schedule the transaction without causing conflicts, AAC algorithm schedules the transaction to it.

The advantage of this scheduling strategy used by AAC algorithm, with respect to AC algorithm, is that in the general case it finds the schedule with a shorter makespan [14], and higher throughput. Actually, AAC algorithm finds the schedule with the shortest makespan possible (this may be trivially shown by contradiction).

So, to implement AAC algorithm, we introduced a dictionary that projects a worker index i to its load L_i, $M(i) = L_i$.

AAC algorithm executes the following steps:

1. Find the indexes i_{min} and i_{max} which correspond to the L_{min} and L_{max}, respectively.
2. Set T_2 to the next transaction from Q_{in}.
3. Determine the estimated execution time for the transaction of type f_2 and assign its value to the variable t_{d2}.
4. Create M.
5. Set $i = i_{min}$, $F_2 = (L_{min}, L_{min} + t_{d2}, V_2)$, *conflictFound* = **False**
6. Iterate through M in the sorted order of worker loads, using the iteration index i. If the index i reaches i_{max}, set the indicator *conflictFound* to **True** and exit the loop. If not, for every already scheduled F_1, check whether there is a conflict between F_1 and F_2. If there is no conflict, set $F_2 = (L_i, L_i + t_{d2}, V_2)$ and exit the loops; otherwise, continue with the execution of the loop.
7. If the *conflictFoound* is True, set $i = i_{max}$, $F_2 = (L_{max}, L_{max} + t_{d2}, V_2)$.
8. Enqueue F_2 at the end of QF_i.
9. Return i.

Theorem 4. AAC algorithm is an online greedy algorithm with exhaustive conflict avoidance that schedules the next input transaction T to the first worker, in order from the least to the most loaded worker, where T does not cause a conflict. Its time complexity is $O(n^2 m^2)$, where n is the number of workers and m is the number of t-variables, if all the transactions read m t-variables and write (possibly different) m t-variables.

Proof. AAC algorithm correctness trivially follows from the correctness of Python primitives: min, max, and index, as well as the primitives for checking attacks and Bernstein conditions among pairs of transactions. Time complexity of all the steps of AAC algorithm, except step 6, is $\Theta(n)$. In the worst case, the two nested loops in step 6 have $(n-1)^2$ passes. We assume QF_i lists are short (thus may be checked in constant time). Time complexity for Python sort is $O(n \cdot log(n))$, for checking attacks is $\Theta(1)$, and for checking Bernstein's conditions is $\Theta(m^2)$ [13]. Since sorting is done before the two nested loops and $O(n^2 m^2)$ dominates on $O(n \cdot log(n))$, the upper bound on execution time for the complete algorithm is $O(n^2 m^2)$.

Remark 2. In our future work we may reduce the bound to $O(n^2 m)$.

3 Theoretical Analysis of Transaction Schedules

In this section, we present the analysis of the expected results for all the four algorithms for scheduling transactions, for the three types of the input workloads, that is the transaction packets (RDW, CFW and WDW), in the form of expected transaction schedules, for the given number of workers (three and four workers). In the following three sections we analyze the expected results for the three given input workloads.

3.1 Analysis for RDW workload

Figure 1 shows the expected transaction schedules for the four transaction scheduling algorithms (RR, ETLB, AC and AAC), for the input RDW workload and three workers. The top of Fig. 1 shows the input queue of transactions, which comprises the sequence of RAA (Read All Accounts) transactions, named R transactions, and the sequence of MT (Money Transfer) transactions, named M transactions, where the R transactions are even and the M transactions are odd. The first R transaction is at the head of the queue. The expected schedules for individual algorithms are shown below the input queue.

In this case, RR algorithm works by the module of three, so, it assigns the zeroth R transaction to the worker W_0, then it assigns the first M transaction to the worker W_1, the second R transaction to the worker W_2, and then it starts this cycle over again.

ETLB algorithm assigns the zeroth R transaction to the worker W_0, the first M transaction to the worker W_1, the second transaction R to the worker W_2, then W_1 becomes the least loaded, so the algorithm assigns the third M

transaction and the fourth R transaction to the worker W_1. Then, the workers W_0 and W_2 become the least loaded. Because the worker W_0 has the smaller index, the algorithm assigns the fifth M transaction to it, etc.

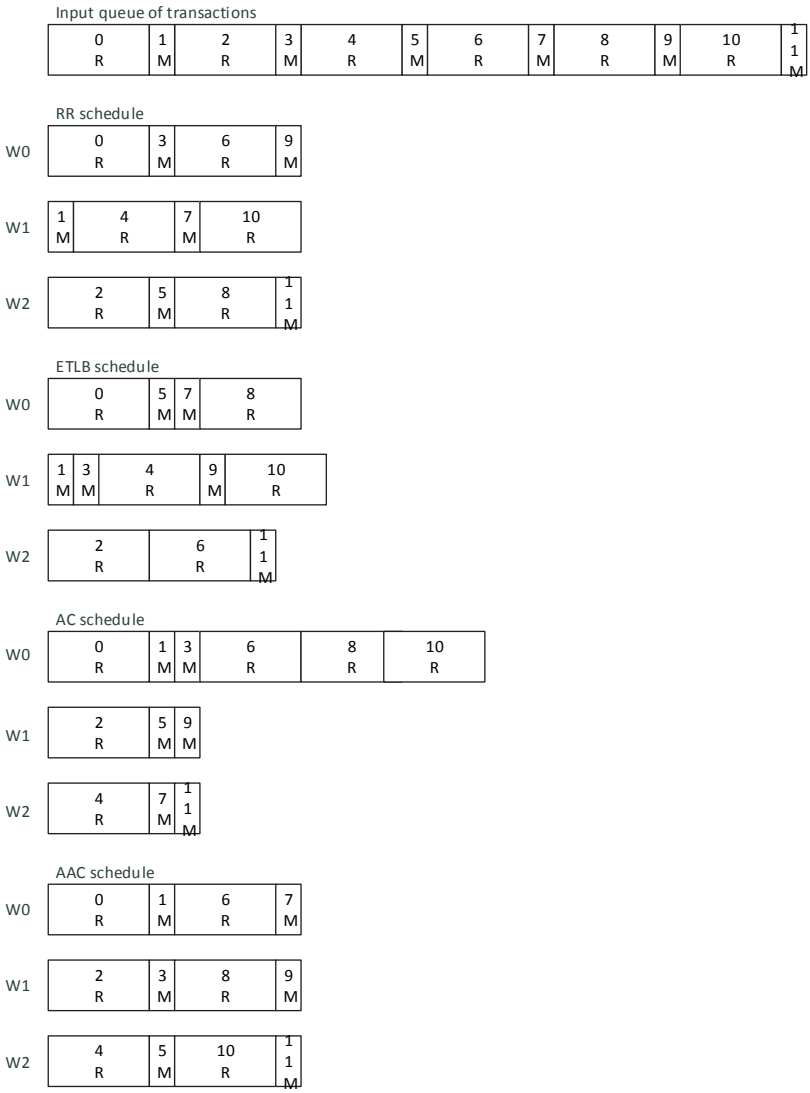

Fig. 1 – *Transaction schedules for RDW workload and three workers.*

AC algorithm assigns the zeroth R transaction to the worker W_0. After that, the least loaded worker with the smallest index is worker W_1. Since the

scheduling of the first M transaction to W_1 would lead to a conflict with the already scheduled R transaction on W_0, the algorithm assigns the first M transaction to the worker with the highest workload – W_0. Further on, the second R transaction goes to the least loaded worker (with the smallest index) W_1, because the second R transaction is not in a conflict with the already scheduled zeroth R transaction (which executes in parallel on W_0), etc.

AAC algorithm assigns the zeroth R transaction to the worker W_0. The algorithm can assign the first M transaction neither to the worker W_1, nor to the worker W_2, because it would make a conflict (with the zeroth R transaction on the worker W_0), so it assigns it to the worker W_0. Then it assigns the second R transaction to the worker W_1 (because it is not in a conflict with the already scheduled zeroth R transaction), the third M transaction to the worker W_1 (because it is not in a conflict with the already scheduled first M transaction). In this step, AAC came to a better schedule than AC algorithm, which, when it discovers that scheduling the third M transaction to the least loaded worker W_2 would lead to a conflict, immediately gives up searching for a suitable worker, and (wrongly) assigns this transaction to the worker W_0. Alternatively, after checking W_2, AAC algorithm checks the next least loaded worker W_1, and finds that there is no conflict.

Summary of the results in Fig. 1: AAC algorithm made the optimal (conflict-free) schedule with the shortest makespan, AC algorithm made a suboptimal (conflict-free) schedule with greater makespan, while RR and ETLB algorithms made the worst schedules containing conflicts. Interestingly, the schedule made by RR algorithm has the same initial makespan as the schedule made by AAC algorithm, but since the latter is conflict-free it is also the final schedule, whereas the former contains conflicts, so some transactions will be aborted and, consequently, the final schedule will be greater.

Fig. 2 shows the transaction schedules for RR and ETLB algorithms for the input RDW workload and four workers. The analysis is similar to the case shown in Fig. 1, and is therefore skipped.

Fig. 3 shows the transaction schedules for AC and AAC algorithms, for the input RDW workload and four workers.

AC algorithm functions similarly as in the preceding cases with the three workers: it assigns the zeroth R transaction to the worker W_0, but the first transaction cannot be assigned to the workers W_1, W_2 and W_3, because it would lead to a conflict, so the algorithm assigns it to the worker W_0. On the contrary,

the second R transaction has no conflict with the already assigned zeroth transaction, so the algorithm assigns it to the worker W_1. Further on, because scheduling of the third M transaction on the least loaded worker W_2 would lead to a conflict with the already scheduled zeroth R transaction (on worker W_0), AC algorithm immediately assigns the third M transaction to the most loaded worker W_0, and by doing so, it misses the chance to make a better schedule, because the third M transaction does not have a conflict with the first M transaction (on the worker W_0). AAC algorithm uses this opportunity.

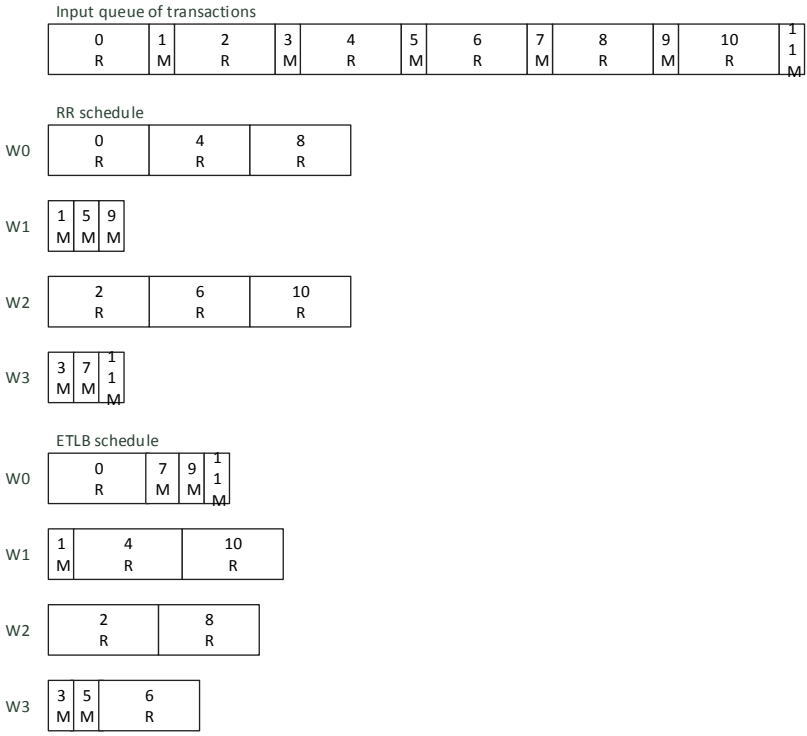

Fig. 2 – *Transaction schedules for RR and ETLB algorithms, for RDW workload and four workers.*

AC algorithm makes a similar miss when scheduling the fifth M transaction, because scheduling this transaction to the least loaded worker W_3 would lead to a conflict with the R transactions already scheduled to W_0, W_1, and W_2. Thus the algorithm assigns this transaction to the most loaded worker W_0, although it could have assigned this transaction to the worker W_1 or to the worker W_2.

AAC algorithm makes the transaction schedule that allows the fastest execution completely without conflicts, which in this particular case has the period of eight transactions, where pairs of transactions are assigned to workers.

Summary for the results in Fig. 2 and Fig. 3: The results are essentially the same as in Fig. 1 – AAC algorithm made the optimal (conflict-free) schedule with the shortest makespan, AC algorithm made a suboptimal (conflict-free) schedule with greater makespan, and RR and ETLB algorithms made the worst schedules containing conflicts.

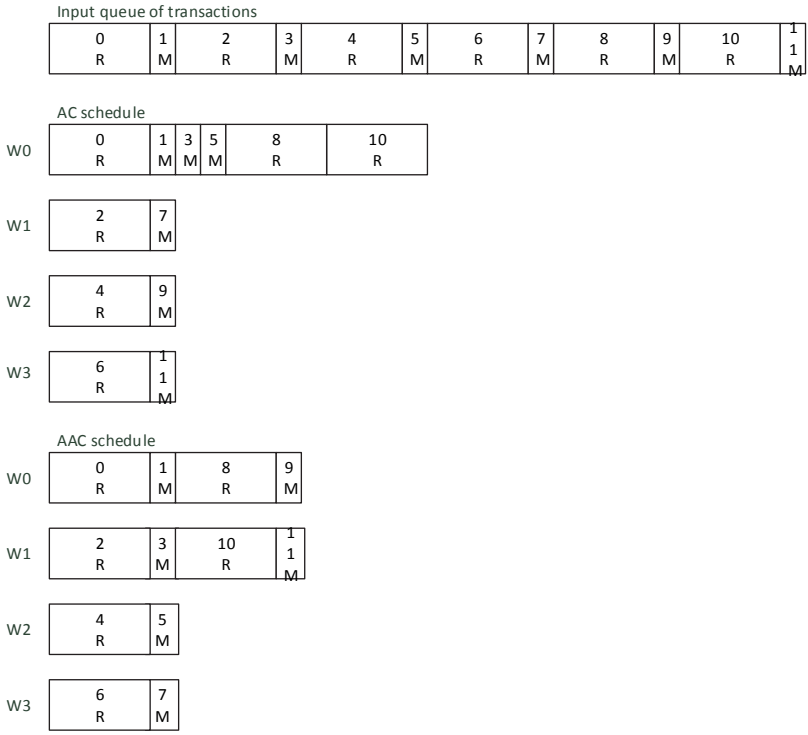

Fig. 3 – *Transaction schedules for AC and AAC algorithms, for RDW workload and four workers*.

3.2 Analysis for CFW workload

Figure 4 shows the expected transaction schedules for all of the algorithms, for the input CFW workload, and for the two cases: (1) with three and (2) with four workers. The top of Fig. 4 shows the input queue of transactions, which comprises the sequence of *M* conflict-free transactions. The expected schedules for individual algorithms are shown below the input queue.

Considering that there is no conflict between transactions, there is no concurrency between the workers, and therefore the transactions may be executed in parallel – three transactions on the three workers, and four transactions on four workers.

Summary of the results in Fig. 4: All the algorithms made the same transaction schedules.

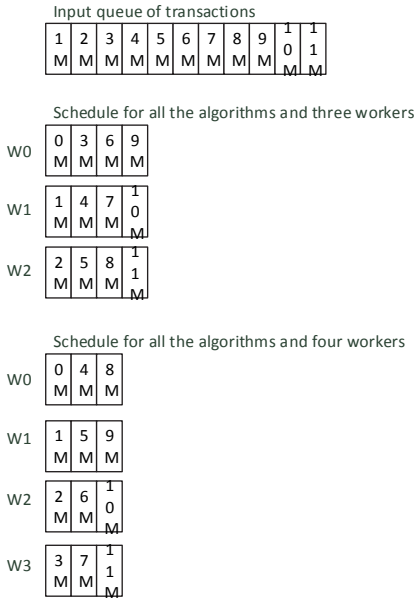

Fig. 4 – *Transaction schedules for all the algorithms, for CFW workload and three or four workers*.

3.3 Analysis for WDW workload

Figure 5 shows the transaction schedules for all of the algorithms, for the input workload WDW and three workers. The top of Fig. 5 shows the input queue of transactions, which comprises the sequence of WAA (Write All Accounts) transactions, named W transactions, and the sequence of MT (Money Transfer) transactions, named M transactions, where the W transactions are even and the M transactions are odd. The first W transaction is at the head of the queue. The expected schedules for individual algorithms are shown below the input queue.

Since RR and ETLB algorithms make no difference between the types of transactions (they treat W and R transactions as equal), they make the same transaction schedule for WDW and RDW inputs. The output schedules in Fig. 5

are the same as the schedules in Fig. 1, except that the R transactions in Fig. 1 are replaced with the W transactions in Fig. 5.

On the other hand, AC and AAC algorithms serialize all of the transactions to the worker W_0, whereas, the worker W_1 and W_2 stay idle. The reason is that, after the algorithm assigns the zeroth transaction W to the worker W_0, no other transaction can be assigned neither to the worker W_1, nor to the worker W_2, because this would lead to a conflict with the already scheduled zeroth W transaction.

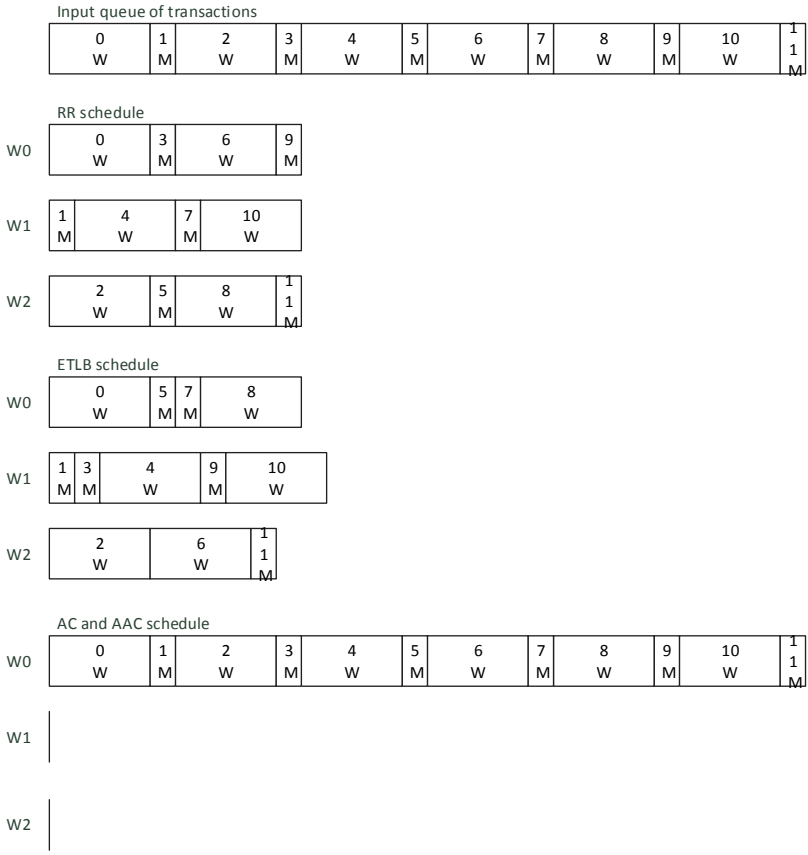

Fig. 5 – *Transaction schedules for all the algorithms, for WDW workload and three workers.*

Summary of the results in Fig. 5: Since all the transactions are in conflict with each other, both AC and AAC algorithms made the optimal transaction schedules by serializing them (so each transaction is executed just once). Although the transaction schedules made by RR and ETLB algorithms look

better at first sight (because it looks like they have shorter makespans), actually they are worse than the schedules made by AC and AAC algorithms, since they contain conflicts. Thus some of the transactions will be executed more than once, and the final makespans will be greater (than the makespans for AC and AAC algorithms).

The results for the input workload WDW and four workers are essentially the same as the results on Fig. 5 – both AC and AAC algorithms made optimal transaction (conflict-free) schedules, whereas RR and ETLB algorithms made worse schedules (containing conflicts), and are thus not shown here.

4 Brief Overview of Experimental Evaluation

Here we provide a brief overview of the experimental evaluation from [9]. Since PSTM is a new STM for Python, we could not directly use standard benchmarks, such as STAMP and STMBench7, because they are written for different languages (C++ and Java) and for STMs with different APIs. Therefore we used the PSTM-based application Bank and the three workloads, which we introduced in Section 3 (RDW, CFW, and WDW). In the experiments, RDW is a mix of 100 R and 100 M transactions, CFW is a packet of 100 M transactions, and WDW is a mix of 100 W and 100 M transactions. The parameter *K* (see Section 2.1) is set to 200.

In the theoretical analysis in Section 3 and the experimental evaluation presented in this section, we were interested in the worst case scenario that happens when transactions arrive to Q_{in} immediately one after the other, which may be seen as a constant distribution with 0 inter-arrival times. In the experiments, this is achieved by storing the complete workload in Q_{in} at the beginning of the workload execution.

We conducted the experiments of Intel Core i7-3770@3.40GHz machine with 16 GB of operating memory, running OS Linux. Since OS uses at least one core, we could use up to three cores for the worker processes. In [9] we made the experiments with two workers (not shown here) and with three workers.

We used the *relative speedup* (*S*) to compare the performances of two scheduling algorithms, A_1 and A_2. Let both A_1 and A_2 process the same workload *L*, and let t_{e1} and t_{e2} be the corresponding mean execution times of *L* using A_1 and A_2, respectively. The relative speedup of an algorithm A_1 over an algorithm A_2, for a given workload *L*, is defined as the ratio $S = t_{e1} / t_{e2}$.

Since RR algorithm is the simplest of all the four scheduling algorithms proposed in this paper, we used it as the baseline for performance analysis. So, we calculated the relative speedups of ETLB, AC, and AAC algorithms over RR algorithm.

The experiments with three workers are organized as follows. We made 3 groups of experiments for 3 different workloads. Further on, within each group of experiments we made 4 sub-groups of experiments for 4 scheduling algorithms. Finally, we executed the given workload 12 times in each sub-group of experiments. So, we made $3 \times 4 \times 12 = 144$ experiments all together.

Although we made every possible precaution (disconnecting the target machine from the net, closing unnecessary processes, etc.), we could not eliminate the interference from OS on the schedules made by the application level scheduler. Because of this imperfectness of measurements, even after eliminating the obvious outliers, some of the experimental results exhibit minor deviations from the expected theoretical results from Section 3. Therefore, we did not provide detail statistics, and these results should be regarded as initial preliminary results. In our future work we plan to conduct more detailed experiments on some many-core machine, where we expect that it would be possible to isolate the worker processes from OS and its processes, because they will execute on different cores.

Table 1 shows the experimental results. The rows of Table 1 correspond to the type of workload. The elements of the column "T; S" contains the average execution time in seconds (1st line) and the corresponding relative speedup (2nd line). The column "A" contains the average of the total number of aborts. The additional data form [9] is that the average execution time for R and W transactions is 45 ms and for M transactions is 0.65 ms.

Overall discussion of the experimental results in Table 1 is given in the next section (Section 5) within the comparison of all the algorithms' features.

Table 1
Results of Experimental Evaluation for 3 workers from [9].

Alg.	RR algorithm		ETLB algorithm		AC algorithm		AAC algorithm	
Load	T; S	A	T; S	A	T; S	A	T; S	A
RDW	2.49 -	1.33	2.96 0.84	70	2.23 1.11	9.33	1.91 1.30	9.33
CFW	0.126 -	0	0.127 0.99	0	0.127 0.99	0	0.127 0.99	0
WDW	6.67 -	88.67	6.44 1.03	97.33	4.57 1.45	1	4.42 1.50	0.67

5 Comparison of the Presented Algorithms' Features

Here we define four features for each online scheduling algorithm: (i) time complexity, (ii) resulting schedule quality, (iii) speed-up over RR algorithm, and (iv) number of aborts. The last two features are defined based on the

complete workload execution, which terminates when all the transactions within a given workload are successfully committed. The *speed-up over RR algorithm* is defined as the complete workload makespan for a given algorithm divided by the complete workload makespan for RR algorithm, whereas the *number of aborts* is the number of aborts for complete workload execution.

We analyzed the first two features (complexity and schedules) in this paper. Recently, we experimentally validated all four algorithms and measured the values for the last two features (speed-up and number of aborts), for RDW, CFW, and WDW workloads, on three workers, in our previous work [9].

Table 2 shows all the features (in rows) for all the algorithms (in columns). We now compare all the presented algorithms for each algorithm's feature. These features are related to the three requirements stated in the abstract of the paper, namely speed of the algorithm is characterized by its time complexity, the total makespan is indirectly characterized by its average speedup over the baseline RR algorithm, and the conflict freeness is characterized by the quality of theoretical initial schedules.

Table 2
Comparison of Online Scheduling Algorithm's Features.

Feature \ Algorithm			RR	ETLB	AC	AAC
Time complexity			$\Theta(1)$	$\Theta(n)$	$O(nm^2)$	$O(n^2m^2)$
Quality of theoretical initial schedules		RDW	Confs	Confs	SuOpt	Opt
		CFW	Opt	Opt	Opt	Opt
		WDW	Confs	Confs	Opt	Opt
Average Speed-up and Aborts (Measured in arch. with 3 workers)	RDW	S	-	0.84	1.11	1.3
		A	70	70	9.33	9.33
	CFW	S	-	0.99	0.99	0.99
		A	0	0	0	0
	WDW	S	-	1.03	1.45	1.50
		A	97.33	97.33	1	0.67
Legend: Confs – with conflicts; SuOpt – suboptimal; Opt - optimal						

As we go from RR, over ETLB and AC, to AAC algorithm, their time complexity increases from $\Theta(1)$, over $\Theta(n)$ and $O(nm^2)$, to $O(n^2m^2)$, respectively. This is as expected, because as they become more involved, their time complexity increases. However, if the number of workers n and the number of used t-variables m are smaller, the corresponding scheduling overhead, even for AAC, may be tolerable. In the experimental validation [9] this was exactly the case, since $n \leq 3$ and $m \leq 2$ (because we used clever

encoding for R and W sets – for all t-variables, we use a special value '*', thus for example the set R for RAA transaction has a single element '*').

However, this increase in time complexity pays well in the quality of resulting theoretical initial schedules. Except for the workload CFW, where all the algorithms produce the optimal schedules (Opt), RR and ETLB produce the worst initial schedules (for both RDW and WDW), which contain transaction conflicts (Confs). AC algorithm produces the optimal schedule for the workload WDW, and a suboptimal (SuOpt) initial schedule for the workload RDW. The term suboptimal here means that the initial schedule is conflict-free but its makespan is longer than the optimal makespan. Finally, only AAC algorithm produces the optimal initial schedules for all the workloads, so the quality of its results is the best.

The average speed-up S and the number of aborts A (which were experimentally measured in the architecture with three workers [9]) are in accordance with the theoretical transaction schedules. For the workload CFW, S is 0.99 on average for all other algorithms, and A is 0, for all the algorithms. ETLB and RR algorithms are the worst: (i) they have the same A on both RDW and WDW workloads, and (ii) S for ETLB is worse than for RR algorithm (0.84 for RDW and 1.03 for WDW).

AC and AAC algorithms have a comparable S for WDW, 1.45 and 1.50, respectively. Finally, AAC algorithm has grater S than AC algorithm for RDW ($1.3 > 1.11$), so AAC produced the best schedule. Note that although both AC and AAC algorithms produced conflict-free schedules for both RDW and WDW workloads, some aborts occurred during the complete workload execution (A is not equal to 0), because we conducted the experiments on the quad-core PC, and only one core was not enough for all the system processes, so local OS Linux compromised the initial conflict-free schedules made by AC and AAC algorithms (we mentioned this possibility in Section 2.1).

6 Conclusion

In this paper we presented four online transaction scheduling algorithms, namely, RR, ETLB, AC, and AAC algorithm, proved their correctness and time bounds, and conducted a theoretical analysis of the transaction schedules they produce, using three different workloads (RDW, CFW, and WDW). Finally, we compared various features of the four algorithms. The theoretical results are as expected: as we go from RR, over ETLB and AC, to AAC algorithms, the quality of resulting schedules increases at the cost of increase of algorithm's time complexity. The experimental results for the average speedup and the number of aborts, in the complete workload execution, which were measured in the architecture with three workers, are in accordance with the theoretical results. For our future work we plan a more detailed experimental evaluation on

a many-core machine, research on tuning the parameter *K*, and further research on scheduling algorithms.

7 Acknowledgments

This work is partially supported by the Ministry of Education, Science, and Technology Development of Republic of Serbia under Grant III-44009-2.

8 References

[1] M. Herlihy, J. E. B. Moss: Transactional Memory: Architectural Support for Lock-Free Data Structures, Proceeding of the 20th Annual International Symposium on Computer Architecture, San Diego, California, USA, May 1993, pp. 289 – 300.

[2] T. Harris, J. Larus, R. Rajwar: Transactional Memory, 2nd Edition, Morgan and Claypool, Madison, Wisconsin, USA, 2010.

[3] R. Guerraoui, M. Herlihy, B. Pochon: Toward a Theory of Transactional Contention Managers, Proceeding of the 24th annual ACM Symposium on Principles of Distributed Computing, Las Vegas, USA, July 2005, pp. 258 − 264.

[4] H. Attiya, L. Epstein, H. Shachnai, T. Tamir: Transactional Contention Management as a Non-Clairvoyant Scheduling Problem, Algorithmica, Vol. 57, No. 1, May 2010, pp. 44 − 61.

[5] H. Attiya, A. Milani: Transactional Scheduling for Read-Dominated Workloads, Journal of Parallel and Distributed Computing, Vol. 72, No. 10, October 2012, pp. 1386 − 1396.

[6] W. N. Scherer, M. L. Scott: Advanced Contention Management for Dynamic Software Transactional Memory, Proceeding of the 24th Annual ACM Symposium on Principles of Distributed Computing, Las Vegas, USA, July 2005, pp. 240 − 248.

[7] R. M. Yoo, H.-H. S. Lee: Adaptive Transaction Scheduling for Transactional Memory Systems, Proceeding of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008, pp. 169 − 178.

[8] M. Popovic, B. Kordic, I. Basicevic: Transaction Scheduling for Software Transactional Memory, 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), Chengdu, China, April 2017, pp. 191 − 195.

[9] M. Popovic, B. Kordic, M. Popovic, I. Basicevic: Advanced Algorithm for Scheduling TM Transactions with Conflict Avoidance, 25th Telecommunications Forum (TELFOR 2017), Belgrade, Serbia, November 2017, pp. 844 − 847.

[10] M. Popovic, B. Kordic: PSTM: Python Software Transactional Memory, 22nd Telecommunications Forum (TELFOR 2014), Belgrade, Serbia, November 2014, pp. 1106 − 1109.

[11] M. Popovic, B. Kordic, I. Basicevic: Estimating Transaction Execution Times for a Software Transactional Memory, 6th International Conference on Information Science and Technology, Dalian, China, May 2016, pp. 137 − 141.

[12] A. J. Bernstein: Analysis of Programs for Parallel Processing, IEEE Transactions on Electronic Computers, Vol. EC-15, No. 5, October 1966, pp. 757 – 763.

[13] Time Complexity, Available at: https://wiki.python.org/moin/TimeComplexity.

[14] M. Popovic, B. Kordic, I. Basicevic: Work, Span, and Parallelism of Transactional Memory Programs, 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, Brno, Czech Republic, August 2015, pp. 59 − 66.