A method for estimating existence of pairwise STM transaction conflicts

Miroslav Popovic[1,a], Branislav Kordic[2], Marko Popovic[1] and Ilija Basicevic[1]

[1]University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia

Abstract. STM transaction schedulers were introduced to improve system performance. However, designing online transaction scheduling algorithms is challenging because at the same time they should: (i) introduce minimal scheduling overhead, (ii) minimize the resulting makespan, and (iii) minimize contention in the resulting schedule. In our previous work we developed the online transaction scheduler architecture and the four scheduling algorithms, named RR, ETLB, AC, and AAC (listed in increasing order of their quality), for scheduling transactions on the Python STM. Both AC and AAC use Bernstein conditions to check for pairwise data races between transactions, at the cost of time complexity that is proportional to the product of the sizes of transaction's read and write sets, which may be significant. In this paper we propose a method for estimating existence of pairwise transaction conflicts whose time complexity is Θ(1). We validate this method by analysing the resulting transaction schedules for the three benchmark workloads, named RDW, CFW, and WDW. The result of this analysis is positive and encouraging – AAC using the new method produces the same result as when using Bernstein conditions. The limitation of the new method is that it may have false reports, both false negatives and false positives.

1 Introduction

Transactional Memory (TM) was originally introduced as an architectural support for lock-free concurrent data structures used on multicores [1]. By replacing traditional locks with TM *transactions* akin to database transactions, TM introduces higher level of abstraction, which both makes concurrent programming easier and enables composing new software components from existing building blocks [2]. These are the two main advantages of TM over locks.

Some theoretical foundations that enable analysis of TM programs have already been laid, which is important when engineering TM-based systems. For example, we may: (i) calculate parallelism using work-span method [3], (ii) determine conflict types and calculate their probabilities for some cases [4], and (iii) estimate transaction execution time based on log-normal distribution [5].

Right from its inception back in 1993, over the years, and to the present date, the TM was and continues to be the area of intensive research and development worldwide. Over this period, many TM implementations with various semantics and APIs were developed [2]. So nowadays, besides the hardware TMs (such as IBM's Blue Gene/Q, zEnterprise EC12, and Power 8 architectures, and Intel's Haswell architecture), there are various software TMs (STMs), hybrid TMs (HTMs), and distributed TMs (DTMs) [6]. In this paper, we use Python STM (PSTM) [7]. However, the main disadvantage of all these kinds of TMs is that their performance may be significantly degraded when they are exposed to high-contention workloads.

Transaction schedulers (aka contention managers) were introduced in order to mitigate this disadvantage. Transaction scheduling algorithms are widely studied in the literature. Some of the algorithms are theoretically well-established with known impossibility results and time complexity bounds [8, 9, 10], whereas other heuristic algorithms are experimentally well-validated [11, 12]. Still, there is a need for further research on online transaction scheduling algorithms.

Designing online transaction scheduling algorithms is challenging because at the same time they should: (i) introduce minimal scheduling overhead, (ii) minimize the resulting makespan, and (iii) minimize contention in the resulting schedule. In our previous work we developed the online transaction scheduler architecture and the four scheduling algorithms, named Round Robin (RR), Execution Time Load Balancing (ETLB), Avoid Conflicts (AC), and Advanced Avoid Conflicts (AAC), for scheduling transactions on the Python STM [13-14].

Both AC and AAC use Bernstein conditions [15] to check for pairwise data races between transactions, at the cost of time complexity that is proportional to the product of the sizes of transaction's read and write sets, which may be significant. In this paper we propose a method for estimating existence of pairwise transaction conflicts whose time complexity is Θ(1). We validate this method by analysing the resulting transaction schedules for the three benchmark workloads, named RDW, CFW, and WDW. The result of this analysis is

[a] Corresponding author: miroslav.popovic@rt-rk.uns.ac.rs

positive and encouraging – AAC using the new method produces the same result as when using Bernstein conditions. The limitation of the new method is that it may have false reports, both false negatives and false positives.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the scheduler architecture and the four scheduling algorithms (RR, ETLB, AC, and AAC). Section 3 presents the new method for estimating existence of pairwise STM transaction conflicts. Section 4 contains the analysis of the resulting transaction schedules for three workloads (RDW, CFW, and WDW) made for two workers. Section 5 presents conclusions.

2 Overview of scheduling algorithms

In this section, we first introduce necessary definitions, and then continue with a brief overview of the transaction scheduler architecture and the four transaction scheduling algorithms, namely RR, ETLB, AC, and AAC.

A transaction T is a pair (f, V), where f is a function executed by T, and V is a pair (R, W), where R and W are sets of t-variables that T reads and writes, respectively. Let t_e be the transaction execution time.

2.1 Scheduler architecture

The scheduler architecture consists of a single scheduler process S and n worker processes W_i, i = 1, …, n, which are running on n processors (or cores). S has its input queue Q_{in} of transactions, whereas each W_i has its input queue Q_i and its output queue D_i. Application processes enqueue new transactions to Q_{in}.

S operates in two alternating rounds. In the first round, S dequeues transactions from Q_{in}, one by one, until Q_{in} becomes empty. For each dequeued T, S calls the configured scheduling algorithm A that returns the index i, which S uses to enqueue T to Qi. In the second round, S just waits for *done* signals at associated D_i queues.

Each W_i dequeues the next transaction T_j = (f_j, V_j) from its Q_i and executes it by calling f_j. If T_j gets aborted, W_i enqueues T_j back to Q_{in}. Once Q_i becomes empty, W_i enqueues the signal *done* to its D_i.

It should be noted that this scheduler architecture is implemented as a Python module, so it runs on top of the local OS (e.g. Windows/Linux). We assume that target hardware has enough cores, such that each W_i runs on its own core; otherwise the local OS scheduler may compromise transaction schedules made by algorithm A.

2.2 RR algorithm

RR algorithm assumes that there is no information about a given T, i.e. that T's t_e and V are not known, so it just assigns T to the next worker Wi in a circle. More precisely, the algorithm initializes its local counter i to 0, and then every time it is called, it calculates the next i by incrementing it with modulo n, and returns the previous value of i. The scheduler in its turn uses i to schedule the next transaction T.

The main advantage of RR algorithm is its minimal overhead, because its time complexity is $\Theta(1)$. On the other hand, RR algorithm is generally neither able to minimize makespan nor to minimize contention, simply because it takes the black-box approach and treats all the transaction equally.

2.3 ETLB algorithm

ETLB algorithm uses the method based on log-normal distribution [5] to estimate the transaction execution time t_e for a given type of transaction. Additionally, the algorithm maintains the list of current worker's loads L, where L_i is a cumulative execution time currently assigned to W_i, i = 1, …, n. Each time the new T is assigned to W_i, the T's estimated t_e is added to L_i.

The ETLB algorithm uses L and the estimated t_e of the next transaction T from the input queue Q_{in} to greedily schedule T to the least loaded worker process W_i, i.e. i = index(min(L)), where the function min returns L_i and the function index returns i.

The main advantage of ETLB algorithm is that it effectively minimizes the makespan and maximizes the throughput. It does this at the expense of increased time complexity, which is $O(n)$, because the time complexity for the Python functions min and index over the list of size n is $O(n)$ [16]. The main disadvantage of the algorithm is that the resulting schedule may have conflicts, because it assumes that the information about t-variables used by transactions is not available. Maybe surprisingly, this means that even though the initial schedule has minimal makespan, actually it may carry high contention, and because of the conflicts, the final makespan may be even worst that for the RR algorithm.

2.4 AC algorithm

AC algorithm is an extended ETLB algorithm that also uses the information about t-variables used by each T, which is available in the pair V = (R, W). In order to be able to detect and avoid conflicts among the next T from Q_{in} and already scheduled transactions, the algorithm firstly introduces the notion of a scheduled transaction F, which is a transaction T scheduled to start at t_b, where F is a tuple (t_b, t_e, V), t_b is F's start time, and t_e is F's completion time. Secondly, for each W_i, the algorithm maintains the queue QF_i (implemented as a list) of scheduled transactions assigned to W_i.

Let F_1 = (t_{b1}, t_{e1}, V_1) be an already scheduled transaction from QF_i and F_2 = (t_{b2}, t_{e2}, V_2) be the next transaction from Q_{in} scheduled at t_{b2}. The algorithm uses the method in Algorithm 1 to detect a possible pairwise conflict between F_1 and F_2. F_1 is attacking F_2 if it completes before F_2, because F_1 will get committed and F_2 will get aborted (line 4) and vice versa F_2 is attacking F_1 if it completes before F_1 (line 5). If none of Fs is attacking another F, there is no conflict (line 12). Otherwise, the method checks Bernstein conditions (lines 7-9), and if they are satisfied, meaning there is no

data race, it concludes that there is no conflict (line 11). Otherwise, the conflict is detected (line 10).

Algorithm 1. The method for detecting pairwise conflicts based on Bernstein conditions.

```
01 conflict_exists(F1, F2)
02   (tb1, te1, V1) := F1      // unpack F1 and F2
03   (tb2, te2, V2) := F2
04   if tb2 < te1 ≤ te2 or     // F1 is attacking F2 ?
05      tb1 < te2 ≤ te1        // F2 is attacking F1 ?
06    then if not              // Is there a data race ?
07      R1 ∩ W2 = {} and       // Bernstein cond. 1
08      R2 ∩ W1 = {} and       // Bernstein cond. 2
09      W1 ∩ W2 = {}           // Bernstein cond. 3
10       then return true      // Conflict exists
11    else return false        // Conflict does not exist
12   else return false         // Conflict does not exist
```

The time complexity of Algorithm 1 is proportional to the sum of the products of the sizes of the read and write sets checked in lines 7-9, i.e. $O(|R_1||W_2| + |R_2||W_1| + |W_1||W_2|)$. When all these sets are of equal size m, the time complexity is $O(m^2)$.

The AC algorithm, first determines the indexes of the least and most loaded workers, i_{min} and i_{max}. Then it schedules F_2 at L_{min}, i.e. it sets F_2 to (L_{min}, $L_{min}+t_e$, V_2), and checks whether there is a conflict among F_2 and any F_1 from all the QF_i, $i = 1, \ldots, n$ (it searches each QF_i in the backwards order starting from its end) using the method in Algorithm 1. If there is no conflict, the algorithm returns i_{min}. Otherwise, the algorithm conservatively back-offs by rescheduling F_2 to start at L_{max}, i.e. it returns i_{max}.

If we assume that QF_i lists are short (thus checking them may be done in almost constant time) and that all the read and write sets are of size m, the overall time complexity of the algorithm is $O(n \cdot m^2)$.

The main advantage of AC algorithm over the ETLB algorithm is that it is able to avoid conflicts, at the expense of increased time complexity of $O(n \cdot m^2)$. However, it is only capable of producing suboptimal schedules, because by using the conservative back-off after detecting a conflict at L_{min}, it may miss a chance to find the next conflict-free position in time after L_{min} and before L_{max}, if such a position exists.

2.5 AAC algorithm

AAC algorithm is an extended AC algorithm that greedily searches for the optimal conflict-free position in time for the next transaction T, rather than giving up by making the conservative back-off when scheduling T at L_{min} would create a conflict. In order to do this, the algorithm creates the sorted list of current worker loads, LS = sorted(L), and then checks all the time positions LS_i, $i = 1, \ldots, n$, one by one, using the method in Algorithm 1, until it finds the first (earliest) LS_i where scheduling T would be conflict-free.

Under the assumption introduced in Section 2.4, the overall time complexity of the algorithm is $O(n^2 \cdot m^2)$.

The main advantage of AAC algorithm over AC algorithm is that always produces optimal schedules, which are conflict-free and with the minimal makespan, but at the expense of more increased time complexity of $O(n^2 \cdot m^2)$. The main motive of this paper is exactly to try to decrease the overall complexity to $O(n^2)$ by replacing the method in Algorithm 1 with the new method for estimating existence of pairwise conflicts in constant time $\Theta(1)$, as will be explained in Section 3.

3 New method for estimating conflicts

In this section, we present the new method for estimating existence of pairwise transaction conflicts whose time complexity is $\Theta(1)$, see Algorithm 2. The main idea of the method is to use the sizes of the read and write sets used by the pair of transactions to estimate the chance of a possible conflict, rather than to use the exact Bernstein conditions.

Intuitively, the smaller the size of these sets is, the smaller is the chance that there is a conflict. Of course, if both W_1 and W_2 are empty, then there is no conflict (line 7). Oppositely, if either $|R_2|$ or $|W_1|$ are greater than the parameter r (line 8), and also symmetrically, if either $|R_1|$ or $|W_2|$ are greater than the parameter r (line 9), then there is a significant chance that there is a conflict. Otherwise, the method assumes that the chance of a conflict is not significant (line 10).

The time complexity of Algorithm 2 is $\Theta(1)$, because the time complexity of the Python function len is $\Theta(1)$ [16].

Algorithm 2. The method for estimating existence of conflicts based on the size of read and write sets.

```
Initially r = 2  // r is a parameter

01 conflict_exists(F1, F2)
02   (tb1, te1, V1) := F1      // unpack F1 and F2
03   (tb2, te2, V2) := F2
04   if tb2 < te1 ≤ te2 or     // F1 is attacking F2 ?
05      tb1 < te2 ≤ te1        // F2 is attacking F1 ?
06    then                     // Is there a data race ?
07      if |W1| = 0 and |W2| = 0 then return false  // No
08      if |R2| > r or |W1| > r     then return true  // Yes
09      if |R1| > r or |W2| > r     then return true  // Yes
10    else return false        // Conflict does not exist
11   else return false         // Conflict does not exist
```

The main advantage of this method is that it is fast – its time complexity is $\Theta(1)$. Of course, its main limitation is that it may have false reports. More precisely, in case when the conflict exists, but the sizes of read and write sets are smaller than r, the method will erroneously report false, i.e. a false negative will occur. Alternatively, in case when the conflict does not exist, but the sizes of read or write sets are greater than r, the method will erroneously report true, i.e. a false positive will occur.

Obviously, the result of Algorithm 2 depends on the characteristic of a given workload and on the value of the parameter r. The parameter r was set to the value 2 in this paper, because it perfectly matches the three workloads (RDW, CFW, and WDW) that were used for

the experimental validation of the scheduling algorithms in [13, 14], as will be shown in Section 4.

4 Analysis of transaction schedules

In this section we validate the new method by showing that transaction schedules made by AAC algorithm using the new method (Algorithm 2) are the same as transaction schedules made by AAC algorithm using the method based on Bernstein conditions (Algorithm 1). We do this for the three workloads, namely Read Dominated Workload (RDW), Conflict-Free Workload (CFW), and Write Dominated Workload (WDW), and for the scheduler architecture with two workers.

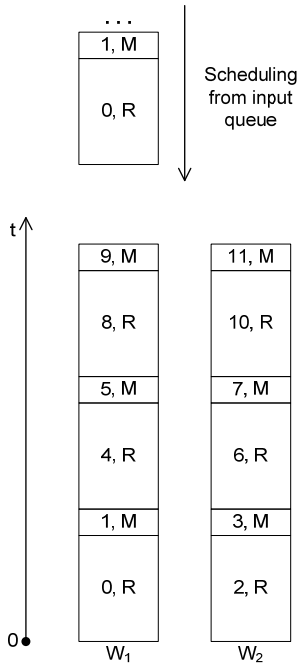

Figure 1. The transaction schedule for the workload RDW.

The above mentioned workloads are composed of the three kinds of transactions that are used in the PSTM-based program named Bank [13, 14]: (i) Read all accounts (R) which reads all t-variables representing accounts, (ii) Money transfer (M) which subtracts a given amount from the first t-variable (account) and adds it to the second, and (iii) Write all accounts (W) which initializes all the t-variables (accounts).

The workloads are packets of transactions whose indexes start from zero, which is here considered as an even number. In particular, RDW is the packet of R and M transactions, such that even (0, 2, …) transactions are R transactions and odd (1, 3, …) transactions are M transactions. CFW is a packet of M transactions. WDW has the same structure as RDW with W transactions instead of R transactions. In all the workloads, all the M transactions are conflict-free among themselves.

We now analyse transaction schedules for each workload. In the following three figures, transactions are oriented vertically. Pairs in parenthesis indicate transaction's index and type – when we refer to a transaction we state both its index and type, e.g. zeroth R transaction, etc. Q_{in} is shown at the top, and input transactions are falling from it downwards, like in the evergreen game Tetris. The resulting schedule is shown at the bottom, in the form of two piles of transactions (one pile for each worker; the worker labels are shown below). The time axes points upwards, so the transactions at the bottom execute first.

Fig. 1 shows the resulting transaction schedule for the workload RDW. The schedule is periodic with the period of four transactions. Here is how this schedule is constructed. AAC algorithm using either method (either the method based on Bernstein condition or the new method) starts by scheduling the zeroth R transaction to W_1. Then both methods report the conflict between the zeroth R and the first M transaction. The new method does this because $|R_1|$ is greater than r (line 9 in Algorithm 2). Thus the first M transaction is scheduled to W_1, too.

Then the second R transaction is scheduled to W_2, because both methods report that resulting schedule is conflict-free. The new method does this because both write sets are empty (line 7 in Algorithm 2). Finally, the third M transaction is scheduled also to W_2 because again both methods report that the result is conflict free. The new method does this because none of the sets are greater than r, so the method falls through the lines 7 to 9 and returns false (line 10). This completes the schedule period.

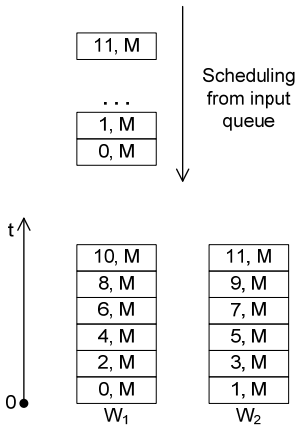

Figure 2. The transaction schedule for the workload CFW.

Fig. 2 shows the resulting transaction schedule for the workload CFW. The schedule is periodic with the period of two transactions. Here is how this schedule is constructed. AAC algorithm using either method starts by scheduling the zeroth M transaction to W_1. Then both methods report that there is no conflict between the zeroth M and the first M transaction. The new method does this because none of the sets are greater than r, so the method returns false (line 10). Thus the first M transaction is scheduled to W_2, and this completes the schedule period.

Fig. 3 shows the resulting transaction schedule for the workload WDW. The schedule is not really periodic but its construction follows the same patter for the group of four transactions. Here is how this schedule is constructed. AAC algorithm using either method starts by scheduling the zeroth W transaction to W_1. Then both

methods report the conflict between the zeroth W and the first M transaction. The new method does this because $|W_1|$ is greater than r (line 8 in Algorithm 2). Thus the first M transaction is scheduled to W_1, too.

Then the second W transaction is scheduled again to W_1, because both methods report that conflict would exist between the zeroth W and the second W transaction if the latter would be scheduled to W_2. The new method does this because $|W_1|$ is greater than r (line 8 in Algorithm 2). Finally, the third M transaction is scheduled also to W_1 because again both methods report that conflict would exist between the zeroth W and the third M transaction if the latter would be scheduled to W_2. The new method does this because $|W_1|$ is greater than r (line 8 in Algorithm 2). This completes the schedule pattern. Effectively, the workload WDW is serialized to W_1.

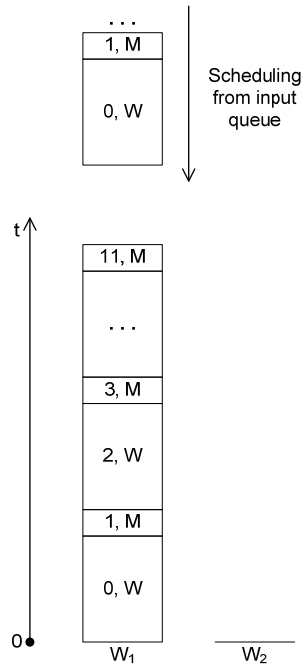

Figure 3. The transaction schedule for the workload WDW.

So, for all the workloads the new method based of the sizes of read and write sets provided the same reports as the method based on Bernstein condition. However, one should keep in mind that the new method could make mistakes on some other workloads.

5 Conclusions

In this paper, we propose a method for estimating existence of pairwise transaction conflicts whose time complexity is $\Theta(1)$. We validate this method by analysing the resulting transaction schedules for the three benchmark workloads, named RDW, CFW, and WDW.

The result of this analysis is positive and encouraging – AAC using the new method produces the same result as when it uses Bernstein conditions. The limitation of the new method is that it may have false reports, both false negatives and false positives.

In our future work, we plan to conduct experimental evaluation of the proposed method on a series of synthetic workloads and with more worker processes running on a manycore architecture, as well as to research other possible methods for estimating existence of pairwise transaction conflicts.

Acknowledgment

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

References

1. M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lockfree data structures, *Proc. of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300 (1993)
2. T. Harris, J.R. Larus, R. Rajwar, *Transactional Memory, 2nd edition* (Morgan and Claypool, 2010)
3. M. Popovic, B. Kordic, I. Basicevic, Work, Span, and Parallelism of Transactional Memory Programs, *Proceedings of the 4th IEEE Eastern European Regional Conference on the Engineering of Computer Based Systems*, pp. 59-66 (2015)
4. M. Popovic, I. Basicevic, Conflict Types and Probabilities in Stochastic TM-Programs, *Proceedings of the 4th IEEE International Conference on Information Science and Technology*, pp. 19-22 (2014)
5. M. Popovic, B. Kordic, I. Basicevic, Estimating Transaction Execution Times for a Software Transactional Memory, *Proceedings of the 6th IEEE International Conference on Information Science and Technology*, pp. 137-141 (2016)
6. P. Romano, R. Palmieri, F. Quaglia, L. Rodrigues, JCSS, **80**, 1, pp. 257-276 (2014)
7. M. Popovic, B. Kordic, PSTM: Python Software Transactional Memory, *Proceedings of the 22nd IEEE Telecommunications Forum*, pp. 1106-1109 (2014)
8. R. Guerraoui, M. Herlihy, B. Pochon, Toward a theory of transactional contention managers, *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pp. 258-264 (2005)
9. H. Attiya, L. Epstein, H. Shachnai, Tami Tamir, Algorithmica, **57**, 1, pp. 44-61 (2010)
10. H. Attiya, A. Milani, J Parallel Distrib Comput., **72**, 10, pp. 1386-1396 (2012)
11. W.N. Scherer, M.L. Scott, Advanced contention management for dynamic software transactional memory, *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pp. 240-248 (2005)
12. R.M. Yoo, H-H.S. Lee, Adaptive transaction scheduling for transactional memory systems”,

Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 169-178 (2008)

13. M. Popovic, B. Kordic, I. Basicevic, Transaction Scheduling for Software Transactional Memory, *Proceedings of the 2nd IEEE International Conference on Cloud Computing and Big Data Analysis*, pp. 191-195 (2017)
14. M. Popovic, B. Kordic, M. Popovic, I. Basicevic, Advanced algorithm for scheduling TM transactions with conflict avoidance, *Proceedings of the 25nd IEEE Telecommunications Forum*, pp. 844-847 (in Serbian, 2017)
15. A. J. Bernstein, IEEE Transactions on Electronic Computers, **EC-15**, 5, pp. 757–763 (1966)
16. Time Complexity page at python.org, https://wiki.python.org/moin/TimeComplexity, visited on August 28th, 2018 (2018)