Noname manuscript No.
(will be inserted by the editor)

Restricted Admission Control in View-Oriented Transactional Memory

Kai-Cheung Leung · Yawen Chen · Zhiyi Huang

the date of receipt and acceptance should be inserted later

Abstract This paper proposes a Restricted Admission Control (RAC) scheme for View-Oriented Transactional Memory. The scheme can control the number of threads concurrently accessing a view in order to reduce the number of aborts of transactions. The RAC scheme has the merits of both the locking mechanism and the transactional memory. A theoretical model is proposed to analyze the performance of the RAC scheme and to provide guidance for dynamic adjustment of the number of concurrent threads accessing the same view. Experimental results demonstrate that theoretical RAC model can mostly provide correct guidance to transactional concurrency control. Our RAC implementation shows that RAC can optimize concurrency control of transactions and performs much better than conventional transactional memory systems such as TinySTM that have no dynamic admission control.

Keywords transactional memory · deadlock · concurrency control · Restricted Admission Control (RAC)

1 Introduction

Parallel programming is becoming mainstream since multicore CPUs have become pervasive. There is a pressing need for parallel programming models to facilitate both performance and convenience. Traditional lock-based programming models can be made efficient but have tedious programmability and are prone to errors such as deadlock. New programming models based on transactional memory are more convenient, but may suffer from low performance [5,18].

Traditionally locking [17,21] is used for concurrency control, where multiple threads/processes[1] have to access a shared data object in an exclusive way. Atomic

Kai-Cheung Leung, Yawen Chen and Zhiyi Huang
Department of Computer Science
University of Otago
E-mail: {kcleung, yawen, hzy}@cs.otago.ac.nz

[1] In the rest of the paper, we use "thread" to mean both process and thread for simplicity since they are identical in terms of concurrency control.

access to a shared object is achieved through a locking mechanism. This lock-based concurrency control is generally regarded as pessimistic approach [26] where conflicts are resolved before they are allowed to happen. Even though locking is an effective mechanism to resolve conflicts, it could result in the deadlock problem if multiple objects are locked in different orders by multiple threads. Moreover, apart from the deadlock problem, fine-grained locks are tedious for programming, while coarse-grained locks often suffer from poor performance due to lack of concurrency.

To avoid the deadlock problem as well as to increase concurrency, Transactional Memory (TM) [14,20] was proposed for shared-memory programming models. In TM, atomic access to shared objects is achieved through transactions. All threads can freely enter a transaction, access the shared objects, and commit the accesses at the end of the transaction. If there are access conflicts among threads, one or more transactions will be aborted and rolled back. TM will undo the effects of the rolled-back transactions and restart them from the beginning. This transaction based concurrency control is labelled as an optimistic approach [3,16] where it is assumed nothing will go wrong and if it does go wrong deal with it later.

In terms of performance, both lock-based and TM-based approaches have their own merits in different situations. When access conflicts are rare (i.e., the contention is low), the TM-based approach has little roll-back overhead and encourages high concurrency since multiple threads can access different parts of the shared data simultaneously. In this situation, however, the lock-based approach has little concurrency due to the sequential access to the shared data, which results in low performance. To increase concurrency and performance, the programmer has to break the shared data into finer parts and use a different lock for each part. This solution using fine-grained locks often complicates the already-complex parallel programs and could incur deadlocks.

On the other hand, when access conflicts are frequent (i.e., the contention is high), the TM-based approach could have staggering roll-back overheads and is not scalable due to a large number of aborts of transactions. In such a situation, it is more effective to use the pessimistic lock-based approach to avoid the excessive operational overheads of transactions.

In order to adaptively improve performance of applications under various contention situations, we proposed a View-Oriented Transactional Memory (VOTM) paradigm [19] that seamlessly integrates the locking mechanism and transactional memory into the same programming model. In VOTM, shared data objects are partitioned into "views" by the programmer according to the memory access pattern of a program. The grain (size) and content of a view are decided by the programmer as part of the programming task, which is as easy as declaring a shared data structure or allocating a block of memory space. Each view can be dynamically created, merged, and destroyed. The most important property for views is that they do not intersect with each other. Before a view is accessed (read or written), it must be acquired; after the access of a view, it must be released. This data-centric model bundles concurrency control and data access together and therefore relieves the programmer from controlling concurrent data access directly with either locks or transactions. When a shared data (i.e. a view) is to be accessed, the programmer just simply uses *acquire_view* to inform the system that the corresponding view is going to be accessed. It is up to the system to decide whether the locking mechanism should be adopted or a transaction should be started for the concurrent access of the shared data.

In VOTM, we adopt a Restricted Admission Control (RAC) scheme that can dynamically adjust the number of threads allowed to access the same view. With the RAC scheme, a view in VOTM is restricted to be accessed by a limited number of threads Q (called admission quota) whose value ranges from 1 to the maximum number of threads (N). If Q is 1, the threads access the set of data objects sequentially as in the lock-based approach. If Q equals N, the RAC scheme behaves like the conventional TM systems where any thread is allowed to start a transaction to access the data objects of the view. However, if Q is greater than 1 but smaller than N, only Q threads are allowed to access the data objects concurrently through transactions. If there are already Q threads accessing the data objects inside uncommitted transactions, other threads are excluded from accessing the set of data objects and have to wait until some existing transactions commit. In addition, RAC can flexibly adjust Q at runtime according to the contention situation, e.g., the number of transactional aborts, to achieve optimal performance, which will be described in details in Section 2.

This paper has the following contributions:

First, we propose the novel Restricted Admission Control (RAC) scheme that adapts flexibly to runtime contention situations in order to achieve optimal performance for concurrent accesses to shared data objects in transactions.

Second, we propose a theoretical model for RAC to measure the contention levels and decide when Q should be adjusted to achieve optimal performance for TM applications. As far as we know, this is the first time that a theoretical analysis is applied to model admission control of transactions.

Third, we evaluate the RAC model with microbenchmarks and show the model can correctly decide if Q should be adjusted at various contention levels. Our experiment shows that this theoretical model is general enough to help measure the contentions for various TM systems.

The rest of the paper is organized as follows: Section 2 will present the RAC scheme and its theoretical model; Section 3 will evaluate the RAC model with microbenchmarks; Section 4 will discuss related work and Section 5 concludes the paper.

2 Restricted Admission Control

The RAC scheme is implemented for every view in VOTM. Each view consists of memory blocks that may store an entire linked list, tree or graph. Each view has an admission quota Q that restricts the maximum number of threads accessing the view concurrently. Before a view is accessed, the primitive *acquire_view* is used. If Q equals 1, *acquire_view* is equivalent to a lock acquisition. In this case, lock mechanism is used instead of the transaction mechanism to avoid transactional overheads. If Q is greater than 1, *acquire_view* will either start a new transaction or wait according to the following RAC scheme.

Suppose a view has an admission quota Q. We assume the current number of threads concurrently accessing the view is P. When the view is acquired through *acquire_view*, RAC follows the steps below:

- Compare P with Q. If P is smaller than Q, increase P by 1, start a new transaction, and return with success.

– If P equals Q, the calling thread is blocked until P becomes smaller than Q.

When the view is released through *release_view*, RAC executes the following steps:

– Try to commit the transaction. If the commit fails, abort and roll back the transaction, decrease P by 1, and reacquire the view as shown above.
– If the commit succeeds, decrease P by 1, and then return with success.

Furthermore, RAC can dynamically adjust the admission quota Q in the following way according to the contention situation. The admission quota Q of each view is initialized as the maximum number of threads (N). RAC regularly checks the contention situation. If the contention is high, RAC will relieve the contention of the view by halving the admission quota Q. This process can be repeated periodically until Q reaches 1, in which case the concurrency control is switched to the lock-based approach and the transaction mechanism is no longer used to access the view. Conversely, when the contention is low, RAC will increase concurrency by doubling Q. This process will repeat periodically until Q reaches N.

Obviously, to find out when to adjust Q is crucial to the performance of RAC. The following theoretical analysis helps understand when RAC can outperform conventional TM systems and when Q should be adjusted to achieve optimal performance.

2.1 RAC vs. conventional TM

Consider a set of *transactions* $S_T = \{T_1, ..., T_n\}$, which access the same view and are executed by N threads. The *duration* of transaction $T_i (1 \leq i \leq n)$ is denoted by t_i and refers to the time period that T_i is executed from start to commit without conflicts and interruptions. For simplicity of the analysis, we assume that, during the execution of T_i, the expected number of aborts is c_i and the average time spent by an aborted transaction is d_i, where $c_i, d_i \geq 0$. Therefore, the expected execution time for T_i is $c_i d_i + t_i$ in conventional TM that has no admission control of transactions.

Makespan is defined as the total time needed to perform all transactions [2]. Suppose that N threads are continuously executing the transactions, then the best possible *makespan* for S_T in conventional TM, denoted by $makespan_{TM}(S_T)$, can be calculated as

$$makespan_{TM}(S_T) = \frac{\sum_{i=1}^{n} c_i d_i + t_i}{N} \tag{1}$$

In RAC, Q transactions are allowed to be executed at any given time, where $1 \leq Q \leq N$. The expected execution time for T_i is $\frac{Q-1}{N-1} \times c_i d_i + t_i$, which can be proven as follows.

Suppose T_i aborts due to the conflict of shared memory location s accessed by $T_{i'}$ in conventional TM. However, in RAC, if T_i is allowed to access s at a given time, the probability that $T_{i'}$ is also allowed to access s is $\frac{Q-1}{N-1}$, because RAC allows only Q threads accessing s at any given time. So, the probability that T_i has 1 abort due to the conflict with $T_{i'}$ is $\frac{Q-1}{N-1}$. According to the binomial distribution, the probability that T_i has k aborts ($k \in \{0, 1, ..., c_i\}$) is $p(k) = \binom{c_i}{k} (\frac{Q-1}{N-1})^k (\frac{N-Q}{N-1})^{c_i - k}$. Therefore, the expected execution time for T_i in RAC is

$\sum_{k=1}^{c_i}(kd_i + t_i)p(k) = \sum_{k=1}^{c_i} kp(k)d_i + \sum_{k=1}^{c_i} p(k)t_i = \frac{Q-1}{N-1} \times c_i d_i + t_i$. (By the binomial distribution, $\sum_{k=1}^{c_i} kp(k) = \frac{Q-1}{N-1} \times c_i$ and $\sum_{k=1}^{c_i} p(k) = 1$)

Suppose the Q threads are continuously executing the transactions in RAC, then the *makespan* for S_T in RAC, denoted by $makespan_{RAC}(S_T)$, is

$$makespan_{RAC}(S_T) = \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} \tag{2}$$

Therefore, the difference of $makespan_{RAC}(S_T)$ and $makespan_{TM}(S_T)$, denoted by Δ, can be obtained by Equation (1) and (2) as follows.

$$\begin{aligned} \Delta &= makespan_{RAC}(S_T) - makespan_{TM}(S_T) \\ &= \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} - \frac{\sum_{i=1}^{n} c_i d_i + t_i}{N} \\ &= \frac{1}{N-1}(\frac{1}{Q} - \frac{1}{N})(\sum_{i=1}^{n} c_i d_i - \sum_{i=1}^{n} t_i(N-1)) \end{aligned} \tag{3}$$

Let $\delta = \frac{\sum_{i=1}^{n} c_i d_i}{\sum_{i=1}^{n} t_i(N-1)}$. It can be derived from Equation (3) that

(a) if $\delta > 1$, then $\Delta > 0$ and $makespan_{RAC}(S_T) > makespan_{TM}(S_T)$. That is, RAC outperforms conventional TM and the performance improvement is Δ when $\delta > 1$ (i.e., $\sum_{i=1}^{n} c_i d_i > \sum_{i=1}^{n} t_i(N-1)$). From this condition, it can be seen that RAC works especially well for transactions with high contention (c_i can be considered as the number of conflicts experienced by T_i), which will be verified in our experimental results.

(b) If $\delta \leq 1$, then $\Delta \leq 0$ and $makespan_{RAC}(S_T) \leq makespan_{TM}(S_T)$. That is, when $\delta \leq 1$, we should set Q to N in RAC. When Q equals to N, $\Delta = 0$ and RAC works the same as the conventional TM.

2.2 RAC with Q' threads vs. Q threads

Similar to the deduction of Equation (3), the difference between makespans of RAC using Q' threads ($makespan_{RAC}(S_T, Q')$) and Q threads ($makespan_{RAC}(S_T, Q)$) is

$$\begin{aligned} &makespan_{RAC}(S_T, Q') - makespan_{RAC}(S_T, Q) \\ &= \frac{1}{Q-1}(\frac{1}{Q'} - \frac{1}{Q})(\sum_{i=1}^{n} c_i(Q) \times d_i(Q) - \sum_{i=1}^{n} t_i \times (Q-1)) \end{aligned} \tag{4}$$

where $c_i(Q)$ and $d_i(Q)$ are the expected number of aborts and the average time spent by an abort of T_i when using Q threads in RAC.

Let $\delta(Q) = \frac{\sum_{i=1}^{n} c_i(Q) \times d_i(Q)}{\sum_{i=1}^{n} t_i \times (Q-1)}$. It can be derived from Equation (4) that

(a) if $\delta(Q) > 1$ and $Q' < Q$, then $makespan_{RAC}(S_T, Q) < makespan_{RAC}(S_T, Q')$. That is, if $\delta(Q) > 1$, RAC should decrease Q to reduce the execution time of the concurrent transactions.

(b) if $\delta(Q) < 1$ and $Q' > Q$, then $makespan_{RAC}(S_T, Q) < makespan_{RAC}(S_T, Q')$. To reduce the execution time of the concurrent transactions, RAC should increase Q.

In summary, the following theorem can be derived:

Theorem 1 *If $\delta(Q)$ is larger than 1, Q should be decreased; if $\delta(Q)$ is smaller than 1, Q should be increased.*

In our implementation of RAC, $\sum_{i=1}^{n} c_i(Q) \times d_i(Q)$ is estimated with the total CPU cycles spent in aborted transactions, and $\sum_{i=1}^{n} t_i$ is estimated with the total CPU cycles spent in successful transactions.

Therefore, $\delta(Q)$ is estimated with Equation (5) in RAC:

$$\delta(Q) = \frac{CPUcycles_{aborted\_tx}}{CPUcycles_{successful\_tx} \times (Q-1)} \tag{5}$$

3 Experimental evaluation

This experiment aims at verifying the above Theorem 1 with microbenchmarks. The experiment also shows that the RAC scheme works well in terms of dynamic adjustment of Q. We use the microbenchmark suite Eigenbench [15] in our experiment.

3.1 Eigenbench

Eigenbench models transactions using orthogonal parameters, and allows a better understanding of which parameter contributes to a particular behaviour of the TM system.

For example, contention in Eigenbench is controlled by adjusting the size of the shared *hot_array* (A1) and number of read and write accesses to the *hot_array* in a transaction (R1 and W1 respectively). *hot_array* can be accessed by all transactions. High contention can be modelled by large number of accesses to the *hot_array* and/or small *hot_array* size (thus each element in the *hot_array* is more likely to be accessed by multiple concurrent transactions and have conflicts). The shared *mild_array* is also accessed by transactions, but each thread has its own subarray in the *mild_array* and each transaction can only access elements in the subarray owned by its thread, so access of the array will not cause conflict, but will increase transaction size.

Moreover, long transactions can be modelled with one or more of the following features:

- reading/writing to a large range of locations in shared memory;
- many repeated accesses to the same location(s) in shared memory;
- frequent access to local memory and/or high computation load inside transactions.

In Eigenbench, a transaction is modelled by a sequence of reads/writes to the shared memory with accesses to local memory and computation (represented by a number of NOPs) in between. A microbenchmark can also have computation and accesses to local memory outside transactions.

Below is the pseudocode outlining the Eigenbench model:

```
1  shared word hot_array[A1];    /* shared array where conflict occurs,
2                                   accessed in tx */
3
4  shared word mild_array[A2];   /* shared array where each thread accesses
```

```
                                its own subarray, so does not cause
                                conflict, but still needs rollback
                                should tx be aborted */

thread_local word cold_array[A3]   /* private to each thread, can be accessed
                                      either inside or outside tx.
                                      if accessed inside tx and tx aborted
                                      then needs to roll back changed */


each thread:

for loops:
do
        tx_start()
        perform r1 reads and w1 writes to the shared hot_array, and
                r2 reads and r2 writes to the shared mild_array
                in *random order*
                each access touches a random element (word) in
                the shared hot_array, or in the dedicated subarray within
                the shared mild_array

                between two accesses to shared arrays, there will
                also be r3i reads and w3i writes to the thread-local
                cold array, and NOPi instructions
        tx_end()

        /* activities outside transactions:
        perform r3o reads and w3o writes to the thread-local array
        perform NOPo instructions
done
```

In Eigenbench, each thread executes *loops* of iterations, and each iteration consists of a transaction, and then activities outside transactions.

In summary, Eigenbench allows us to model contentions quantitatively with various orthogonal parameters, which is ideal for us to evaluate the RAC model.

In order to verify the theoretical RAC model, we use four Eigenbench microbenchmarks: Highcon, FutileStall, StarvingElder and StarvingWriter. Highcon is configured by ourselves, but FutileStall, StarvingElder and StarvingWriter are taken from [4]. Parameters of each microbenchmark are listed in Table 1.

Table 1 Eigenbench parameters for the microbenchmarks

Application	*N*	loops	A1	A2	A3	R1	W1	R2	W2	R3i	W3i	R3o	W3o	NOPi	NOPo
Highcon	16	400	4k	1m	8k	100	10	20	20	10	20	0	0	20k	10m
FutileStall	16	10k	256	16k	8k	80	20	10	10	0	0	0	0	0	0
StarvingElder	1	100k	1k	1m	8k	128	32	20	20	0	0	100	100	0	0
	15	1m	1k	1m	8k	2	2	20	20	0	0	100	100	0	0
StarvingWriter	1	10k	32	1m	1m	0	30	0	0	500	0	0	0	0	0
	15	100k	32	1m	1m	30	0	0	0	0	0	0	0	0	0

Highcon has high contention long transactions that have computation loads both inside and outside transactions. It mimics realistic TM applications with transactions that are computationally intensive apart from having many high-contentious accesses to the *hot_array*. Each thread executes 400 transactions. In addition, this microbenchmark also has substantial computational workload *outside* transactions.

In FutileStall, transactions read from and then write to highly-contended data. Some transactions are stalled by other transactions that eventually will abort, and thus have futile stalls. In this microbenchmark, each thread executes 10000 transactions.

In StarvingElder, thread 0 executes 100,000 long transactions that read/write many locations in the *hot_array*, while each of threads 1-15 execute 1,000,000 short transactions that can conflict with and abort long transactions. As a result, these long transactions will make little progress.

In StarvingWriter, thread 0 executes 10,000 long writer transactions that write to the small *hot_array*, while each of threads 1-15 executes 100,000 read-only transactions that have frequent access to the *hot_array*, thus conflict with the writer transactions executed by thread 0. These read-only transactions will impede progress of the long writer transactions, since the multiple readers may continuously lock the same location for a long time.

3.2 Verification of Theorem 1

In this experiment, we are going to verify if Theorem 1 can correctly suggest that the admission quota Q should be adjusted.

In the experiment, RAC is implemented over the TinySTM-1.0.0 [10] with both its default encounter-time locking algorithm (TinySTM-ETL) and its alternative commit-time locking algorithm (TinySTM-CTL). These two RAC implementations are denoted as "TinySTM-ETL+RAC" and "TinySTM-CTL+RAC" respectively. We use the two different implementations to show if the RAC model can work with different TM algorithms. To examine the correctness of Theorem 1, Q in RAC is fixed to 1, 2, 4, 8, 12 and 16 respectively, without dynamic adjustment during runtime. In all tests, the number of total threads (N) is fixed to 16. Thus the $Q = 16$ case is equivalent to the conventional TM that has no restriction of admissions. In all microbenchmarks, we use only one view, which is sufficient to verify Theorem 1. All tests are carried out on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running with 800MHz and 16GB DDR2 memory. Linux kernel 2.6.32 and the compiler gcc-4.4 are used during benchmarking. All programs are compiled with the optimization flag -O2 because it is more stable than -O3.

As mentioned previously, $\delta(Q)$ in Theorem 1 is estimated with Equation (5). Perfctr-2.6.42 [22] is used to measure the CPU cycles spent in aborted transactions and successful transactions.

For each Q, the runtime of each microbenchmark and its $\delta(Q)$ are presented in the following tables. To verify if Theorem 1 is correct for a microbenchmark, we compare the runtimes of Q and Q', where $Q' < Q$. If $\delta(Q) > 1$ and the runtime of Q is larger than the runtime of Q', or alternatively if $\delta(Q) < 1$ and the runtime of Q is smaller than the runtime of Q', then Theorem 1 is correct; otherwise it is not quite correct.

For example, Table 2 shows the results of Highcon (on TinySTM-ETL+RAC) at different values of Q. When Q is 16, $\delta(Q)$ is 1.25, larger than 1, which suggests we should decrease Q according to Theorem 1. Compared with $Q = 12$, we find the runtime at $Q = 12$ is smaller than the runtime at $Q = 16$. This shows Theorem 1 has correctly suggested a smaller Q should relieve the contention situation and improve performance. Likewise, Theorem 1 is correct for $Q = 12$ and $Q = 2$.

The exceptions are $Q = 4$ and $Q = 8$. For $Q = 8$, $\delta(Q)$ is 0.98, slightly smaller than 1, but the experimental result suggests that a smaller Q (4) can improve performance. Similarly, for case $Q = 4$, $\delta(Q)$ is smaller than 1, but a larger Q (8) cannot improve performance. We attribute this inaccuracy to the estimation error

of $\delta(Q)$. To accommodate the error, we should use a critical zone with minimum and maximum thresholds instead of a critical point value (1). If $\delta(Q)$ is in the critical zone, Q should not be adjusted. Our RAC implementation in Section 3.3 will suggest a suitable critical zone for $\delta(Q)$ that makes the RAC model work correctly for all cases of the microbenchmarks.

Table 2 Highcon with ETL

Q	1	2	4	8	12	16
Runtime(s)	52.3	34.3	25.4	54.5	69.6	76.0
$\#abort$	0	1.61k	7.17k	142k	438k	648k
$\#tx$	6.4k	6.4k	6.4k	6.4k	6.4k	6.4k
$CPUcycles_{aborted\_tx}$	0	7.05G	29.7G	283G	591G	774G
$CPUcycles_{successful\_tx}$	41.0G	41.1G	41.1G	41.1G	41.1G	41.1G
$\delta(Q)$	N/A	0.17	0.24	0.98	1.30	1.25

Table 3 Highcon with CTL

Q	1	2	4	8	12	16
Runtime(s)	52.7	30.8	18.2	15.1	15.0	15.0
$\#abort$	0	739	2.08k	3.03k	3.03k	3.03k
$\#tx$	6.4k	6.4k	6.4k	6.4k	6.4k	6.4k
$CPUcycles_{aborted\_tx}$	0	4.64G	12.8G	19.1G	19.2G	19.2G
$CPUcycles_{successful\_tx}$	41.1G	41.1G	41.1G	41.1G	41.1G	41.1G
$\delta(Q)$	N/A	0.11	0.10	0.07	0.04	0.03

In Table 3 (TinySTM-CTL+RAC), the actual optimal value of Q is 16 with the smallest runtime of 15.0s. In the table, $\delta(Q)$ is around 0.1 or smaller for all values of Q. Therefore, RAC should keep increasing Q and use the maximum possible value (16), which is consistent with Theorem 1.

It is worth noting that TinySTM-CTL does not cause as much false aborts and its actual contention is small. Therefore, there is no need for RAC to restrict admission.

Table 4 FutileStall with ETL

Q	1	2	4	8	12	16
Runtime(s)	4.21	7.04	12.4	17.8	21.9	40.3
$\#abort$	0	690k	3.80m	9.95m	16.0G	47.3G
$\#tx$	160k	160k	160k	160k	160k	160k
$CPUcycles_{aborted\_tx}$	0	5.11G	24.8G	58.2G	88.3G	242G
$CPUcycles_{successful\_tx}$	3.10G	4.01G	4.76G	5.45G	5.64G	6.25G
$\delta(Q)$	N/A	1.27	1.73	1.52	1.42	2.58

Table 5 FutileStall with CTL

Q	1	2	4	8	12	16
Runtime(s)	4.86	5.61	5.66	6.01	6.63	7.73
$\#abort$	0	170k	507k	1.17m	1.98m	3.01m
$\#tx$	160k	160k	160k	160k	160k	160k
$CPUcycles_{aborted\_tx}$	0	4.15G	12.1G	28.3G	50.8G	86.3G
$CPUcycles_{successful\_tx}$	3.98G	4.01G	4.23G	4.72G	5.38G	6.29G
$\delta(Q)$	N/A	1.03	0.95	0.86	0.86	0.91

From Table 4 and 5, we can see the actual optimal Q for FutileStall on TinySTM-ETL+RAC and TinySTM-CTL+RAC is 1. When Q is between 2 and 16 in TinySTM-ETL+RAC, $\delta(Q)$ is larger than 1. Thus Q should be kept decreasing to the smallest value (1) which performs the best. Therefore, in these cases, Theorem 1 is correct.

However, for FutileStall on TinySTM-CTL+RAC in Table 5, when Q is between 4 and 16, $\delta(Q)$ is slightly smaller than 1, but decreasing Q can still further reduce runtime. This inaccuracy can be again attributed to the estimation error of $\delta(Q)$.

Table 6 StarvingElder with ETL

Q	1	2	4	8	12	16
Runtime(s)	354.0	180.2	99.3	51.8	47.4	46.6
$\#abort$	0	149k	593k	2.25m	3.02m	3.03m
$\#tx$	15.1m	15.1m	15.1m	15.1m	15.1m	15.1m
$CPUcycles_{aborted\_tx}$	0	2.82G	10.8G	35.5G	45.0G	45.5G
$CPUcycles_{successful\_tx}$	256G	259G	262G	264G	265G	266G
$\delta(Q)$	N/A	0.011	0.013	0.019	0.016	0.011

Table 7 StarvingElder with CTL

Q	1	2	4	8	12	16
Runtime(s)	348.0	178.5	99.4	56.0	51.3	51.2
$\#abort$	0	82.2k	359k	1.04m	1.04m	1.04m
$\#tx$	15.1m	15.1m	15.1m	15.1m	15.1m	15.1m
$CPUcycles_{aborted\_tx}$	0	3.28G	14.9G	37.6G	37.7G	37.7G
$CPUcycles_{successful\_tx}$	252G	256G	259G	264G	264G	264G
$\delta(Q)$	N/A	0.013	0.019	0.021	0.013	0.010

Table 8 StarvingWriter with ETL

Q	1	2	4	8	12	16
Runtime(s)	29.0	24.8	22.3	21.8	21.8	21.8
$\#abort$	0	3.76m	9.68m	10.6m	10.6m	10.6m
$\#tx$	15.1m	15.1m	15.1m	15.1m	15.1m	15.1m
$CPUcycles_{aborted\_tx}$	0	2.77G	6.73G	7.38G	7.39G	7.39G
$CPUcycles_{successful\_tx}$	20.5G	20.7G	20.8G	20.8G	20.8G	20.8G
$\delta(Q)$	N/A	0.13	0.11	0.051	0.032	0.023

Table 9 StarvingWriter with CTL

Q	1	2	4	8	12	16
Runtime(s)	29.0	19.9	19.2	19.2	19.2	19.2
$\#abort$	0	2.82k	3.27k	3.41k	3.43k	3.44k
$\#tx$	15.1m	15.1m	15.1m	15.1m	15.1m	15.1m
$CPUcycles_{aborted\_tx}$	0	11.1m	15.6m	20.5m	21.6m	22.1m
$CPUcycles_{successful\_tx}$	20.5G	20.6G	20.6G	20.6G	20.6G	20.6G
$\delta(Q)$	N/A	0.0005	0.0002	0.0001	0.0001	0.0001

Long transactions in StarvingElder and writers in StarvingWriter can have poor progress. However, as shown in Table 6, 7, 8 and 9, the overall contention in both cases on TinySTM-ETL+RAC and TinySTM-CTL+RAC is not sufficiently high to justify admission control, as other short transactions still make good progress. Since $\delta(Q)$ is much smaller than 1 with all values of Q and the runtime at $Q = 16$ is the smallest in all cases, Theorem 1 correctly predicts that it is inappropriate to restrict admission of transactions.

In conclusion, Theorem 1 based on the theoretical RAC model can correctly predicts whether Q should be adjusted in most cases for both TinySTM-ETL and TinySTM-CTL. However, for a few cases like Highcon on TinySTM-ETL and FutileStall on TinySTM-CTL, when $\delta(Q)$ is slightly smaller than 1, decreasing Q can still improve performance. This discrepency from Theorem 1 can be attributed to the estimation error of $\delta(Q)$. To reduce (or avoid) the estimation error, we use a critical zone introduced in the following implementation of RAC.

3.3 Performance of the RAC scheme

We use Theorem 1 to guide the dynamic adjustment of Q in RAC. However, due to estimation error, the previous results indicate that even when $\delta(Q)$ is slightly smaller than 1, contention can still be high, and performance can still be improved by decreasing Q. For the same reason, conversely $\delta(Q)$ would need to be much smaller than 1 to indicate low contention to justify increasing Q. As suggested previously, to accommodate the estimation error of $\delta(Q)$, we need to use a critical zone instead of a critical point for $\delta(Q)$.

In our implementation of RAC, we use a critical zone with two values MAX and MIN. Q will be decreased when $\delta(Q) > MAX$; and Q will be increased when $\delta(Q) < MIN$. MAX and MIN are set to 0.8 and 0.05 respectively based on our experimental results.

Additionally, in our implementation, to eliminate cache flushing overheads on incrementing the shared counter P (number of threads holding the view) in the RAC metadata, when it is clear that admission control to the view is unnecessary because of the following low contention condition, the RAC mechanism will be disabled.

- 20000 transactions are executed since Q is set to N, and
- $\delta(Q) < MIN$

After the RAC mechanism of the view is disabled, access to the view will no longer be restricted until the contention situation of the view changes.

Table 10 and 11 show the performance of the RAC scheme for the previous microbenchmarks. We compare the following three implementations: RAC with the above dynamic adjustment of Q (denoted as RAC in Table 10 and 11), no admission control ($Q = 16$, denoted as Q16), and RAC with static Q whose value is set to the optimal value of Q (denoted as OPT). We compare RAC with OPT to verify whether the RAC scheme can find the optimal Q in practice. Also we compare RAC with Q16 to show the performance improvement of RAC over conventional TM. Note that, in the tables, "$Q(RAC)$" is the Q picked up by the RAC with dynamic adjustment, and "$Q(OPT)$" represents the Q with which RAC works the best.

In all microbenchmarks, we use only one view, though multiple views would make RAC perform even better, as demonstrated in [19].

Table 10 Performance of Adaptive RAC in TinySTM-ETL

Microbenchmark	*time*(s) (RAC)	Q (RAC)	*#aborts* (RAC)	*time*(s) (Q16)	*#aborts* (Q16)	*time*(s) (OPT)	Q (OPT)	*#aborts* (OPT)
Highcon	25.6	4	9.96k	76.0	648k	25.4	4	7.17k
FutileStall	3.23	1	7.98	40.3	47.3m	4.21	1	0
StarvingElder	47.0	16	3.05m	46.8	3.02m	46.8	16	3.02m
StarvingWriter	22.1	16	33.2m	21.8	10.6m	21.8	16	10.6m

Table 11 Performance of Adaptive RAC in TinySTM-CTL

Microbenchmark	*time*(s) (RAC)	Q (RAC)	*#aborts* (RAC)	*time*(s) (Q16)	*#aborts* (Q16)	*time*(s) (OPT)	Q (OPT)	*#aborts* (OPT)
Highcon	15.0	16	2.98k	15.0	3.01k	15.0	16	3.01k
FutileStall	3.33	1	46.5k	7.73	3.01m	4.86	1	0
StarvingElder	51.5	16	1.05m	51.2	1.05m	51.2	16	1.05m
StarvingWriter	19.2	16	3.83k	19.2	3.80k	19.2	16	3.80k

From Table 10 and 11, it can be seen that in all microbenchmarks on both TinySTM-ETL and TinySTM-CTL, RAC has correctly settled to the optimal Q values. For example, in Highcon on TinySTM-ETL, RAC correctly settled to $Q = 4$ and has a 200% improvement over conventional TM (Q16). The RAC scheme settles to the correct Q value very quickly, as the runtime of RAC (25.6) is similar to the runtime of OPT (25.4). Also the number of aborts in RAC (9.96k) is only slightly more than OPT (7.17k).

In FutileStall, RAC has correctly settled to $Q = 1$ to in both TinySTM-ETL and TinySTM-CTL, and the runtimes in both cases (3.23s and 3.33s respectively) are shorter than OPT (4.21s and 4.86s respectively). This improvement is attributed to the turn-off of the transactional mechanism in RAC when Q reaches 1.

In StarvingElder, StarvingWriter and the TinySTM-CTL version of Highcon, the contention is not high enough to justify restricting admission and RAC has correctly stayed at $Q = 16$. The runtime of RAC is similar to OPT in all cases.

From the above results, it can be seen that RAC based on Theorem 1 can quickly choose the optimal Q in different TM algorithms, and can therefore improve performance of TM applications with high contention. Our previous work [19] showed RAC was also able to improve performance of real TM applications.

4 Related work

Approaches to concurrency control in TM can be classified into three types: in-transaction conflict resolution, transactional scheduling, and adaptive locks.

4.1 In-transactional conflict resolution

In-transaction conflict resolution aims to resolve conflicts effectively to reduced wasted work of aborted transactions. All in-transaction conflict resolution algorithms, including both encounter-time locking (DSTM [13, 25], SXM [11] and McRT-STM [24]) and commit-time locking (TL-2 [7] and NOrec [6]) algorithms, resolve conflicts *within* a transaction only *after* these conflicts have been detected, but threads are still freely admitted into transactions. Therefore, the aborts cannot be stemmed in high contention and work is still wasted by transactions that eventually aborts, as shown in our experimental results.

4.2 Transactional scheduling

Transactional scheduling can control the admission of transactions when contention is high. It can prevent conflicts before they occur and therefore reducing wasted work on aborted transactions. For example, transaction scheduling algorithms such as [28] use a thread-local contention score. When a thread experiences high contention, it queues the starting transaction to a central scheduler, which will execute queued transactions serially. [8] adopts a similar approach, except when a thread experiences high contention it uses a heuristic approach that predicts read and write sets of the starting transaction using read and write sets of previous transactions of the threads. If any address in the predicted read and write sets is being written by any other currently executing transactions, then the

starting transaction will be queued to be executed serially. Otherwise, the transaction executes immediately. This algorithm relies on heuristic prediction of what will be read/written in the starting transactions. The admission control algorithm in [1] also adopts a similar approach. This family of transactional scheduling algorithms works orthogonally with the in-transaction conflict resolution algorithms mentioned above.

However, the above transactional scheduling algorithms use empirical thresholds to decide contention level. There is no theoretical model to guide the selection of those threshold values, as we discussed in this paper. As far as we know, the theoretical RAC model proposed in this paper is the first model that provides theoretical guidance to transactional concurrency control.

Furthermore, as discussed in [19], all transaction scheduling algorithms described above treat the entire TM with the same scheduling decision. Therefore, access to a low-contention shared object can be unreasonably restricted due to other high-contention shared objects in TM. Also the statistics collected for the entire TM are not as accurate as those collected per view basis, which will enlarge the estimation error of $\delta(Q)$ in the RAC model. However, RAC in VOTM treats each view individually and the estimation of $\delta(Q)$ is thus more accurate.

4.3 Adaptive locks

The speculative lock elision (SLE)-based model [23] was proposed to avoid unnecessary exclusive accesses in lock-based programs. An elidable lock can be acquired "speculatively" (using TM) or "non-speculatively" (using mutex). At any time, an elidable lock can be acquired speculatively by multiple threads, but only one thread can hold an elidable lock non-speculatively at any time.

The adaptive lock model in [27] has a similar approach, except a thread trying to acquire the lock in mutex mode must wait until all existing threads holding the lock in transaction mode to finish.

Like VOTM, both SLE and adaptive lock models have separate access control on each elidable lock, to ensure restrictions placed by the system on locks with high contention will not unnecessarily affect concurrency of accessing other elidable locks with low contention. However these models either allows all threads to hold the elidable lock in speculative mode, or exclusive access to one thread during non-speculative (mutex) mode, yet as shown in Sections 3 and 2, there are some cases where the optimal admission quota of a lock/view is actually between 1 and N. Therefore, the RAC scheme can achieve a superior performance by finding out the optimal admission quota to achieve maximal concurrency rather than only choosing between the two extremes – exclusive access to one thread or admitting all threads.

5 Conclusions and future work

As shown in the microbenchmark results, our theoretical RAC model can mostly provide correct guidance to transactional concurrency control. The RAC scheme based on the model can improve performance of TM regardless of which underlying TM algorithm is used. In any TM algorithms, there will be situations where the contention becomes very high (the number of aborts becomes much larger than

the number of transactions). In these situations, RAC will dynamically adjust the number of threads admitted to a view to control contention, thereby reducing works wasted by aborted transactions and improving progress. Experimental results show that RAC has superior performance to conventional TM because of the ability of RAC controlling admission and switching between TM and locking, whereas conventional TM has a performance issue when the contention is high and lock-based approach only works well in fine-grained locking but poorly in coarse-grained locking [19]. Therefore, the RAC scheme enables VOTM to have better performance than the conventional TM and better convenience (and sometimes better performance) than lock-based programming.

One issue with RAC is blocking of threads by RAC when Q is smaller than N. This blocking seems to violate the lock-free or obstruction-free feature of TM systems [12]. Even though this feature is arguably necessary [9], RAC can quickly resolve this kind of blocking when the contention becomes low and thus Q is increased up to N, as long as Q does not become 1. If necessary, RAC can completely avoid blocking by using transactions even when Q equals 1, though it will lose some performance gain. In this way, if the system discovers that blocking is too long, the blocking can be easily lifted by increasing Q. Actually, in normal situations, the blocking in RAC is not worse than the live-locking in TM when transactions abort each other without progress under high contention.

As a future work, we will further investigate how to reduce the estimation error of $\delta(Q)$ in the RAC model. We will also investigate the performance of the RAC scheme for multiple views, which we believe is another strength of VOTM.

References

1. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Watson, I.: Adaptive concurrency control for transactional memory. Tech. rep., University of Manchester (2007)
2. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. Algorithmica **57**, 44–61 (2010). URL http://dx.doi.org/10.1007/s00453-008-9195-x. 10.1007/s00453-008-9195-x
3. Bernstein, P., Goodman, N.: Concurrency control in distributed database systems. ACM Computer Survey **13**(2), 185–221 (1981)
4. Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: Proceedings of the 34th annual international symposium on Computer architecture, pp. 81–91. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1250662.1250674. URL http://doi.acm.org/10.1145/1250662.1250674
5. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue **6**, 46–58 (2008). DOI http://doi.acm.org/10.1145/1454456.1454466
6. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 67–78. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1693453.1693464
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing (2006)
8. Dragojević, A., Guerraoui, R., Singh, A.V., Singh, V.: Preventing versus curing: avoiding conflicts in transactional memories. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 7–16. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1582716.1582725
9. Ennals, R.: Software transactional memory should not be obstruction-free. Tech. rep., Intel Corporation (2006)

10. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1345206.1345241
11. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of the 19th International Symposium on Distributed Computing, pp. 26–29. LNCS, Springer (2005)
12. Guerraoui, R., Kapalka, M.: On obstruction-free transactions. In: 20th ACM Symposium on Parallelism in Algorithms and Architectures (2008)
13. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd annual symposium on Principles of Distributed Computing, pp. 92–101. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/872035.872048
14. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Computer Architecture News **21**, 289–300 (1993). DOI http://doi.acm.org/10.1145/173682.165164
15. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A simple exploration tool for orthogonal tm characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10), pp. 1–11. IEEE Computer Society, Washington, DC, USA (2010). DOI http://dx.doi.org/10.1109/IISWC.2010.5648812. URL http://dx.doi.org/10.1109/IISWC.2010.5648812
16. Kung, H., Robinson, J.: On the optimistic methods for concurrency control. ACM Transactions on Database Systems **6**(2), 213–226 (1981)
17. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974). DOI http://doi.acm.org/10.1145/361082.361093
18. Larus, J.R., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture. Morgan and Claypool (2007)
19. Leung, K., Huang, Z.: View-oriented transactional memory. In: The Fourth International Workshop on Parallel Programming Models and Systems Software for High-end Computing, in Proceedings of the 40th International Conference on Parallel Processing (2011)
20. Lomet, D.B.: Process structuring, synchronization, and recovery using atomic actions. In: ACM Conference on Language Design for Reliable Software, pp. 128–137 (1977)
21. Peterson, G.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (1981)
22. Pettersson, M.: The Perfctr Linux Performance Monitoring Counters Driver. Uppsala University (2004)
23. Roy, A., Hand, S., Harris, T.: A runtime system for software lock elision. In: Proceedings of the 4th ACM European Conference on Computer Systems, pp. 261–274. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1519065.1519094. URL http://doi.acm.org/10.1145/1519065.1519094
24. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1122971.1123001
25. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: M.K. Aguilera, J. Aspnes (eds.) Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005). DOI http://doi.acm.org/10.1145/1073814.1073861
26. Tanenbaum, A., Steen, M.: Distributed Systems: Principles and Paradigms, Chapter 5. Prentice Hall (2002)
27. Usui, T., Behrends, R., Evans, J., Smaragdakis, Y.: Adaptive locks: Combining transactions and locks for efficient concurrency. In: Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques. IEEE Computer Society, Washington, DC, USA (2009)
28. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 169–178. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1378533.1378564. URL http://doi.acm.org/10.1145/1378533.1378564