2012 41st International Conference on Parallel Processing Workshops

When and how VOTM can improve performance in contention situations

K. Leung, Y. Chen and Z. Huang
Department of Computer Science
University of Otago
Dunedin, New Zealand
Email:{kcleung;yawen;hzy}@cs.otago.ac.nz

***Abstract*—This paper extends the Restricted Admission Control (RAC) theoretical model to cover the multiple-view cases in View-Oriented Transactional Memory (VOTM) to analyze potential performance gain in VOTM when shared data is partitioned into multiple views. Experimental results show that partitioning shared data into separate views, each of which is independently controlled by RAC, can improve performance when one of the views has high contention while others have low contention. In memory-intensive transactions, even when contention is not high enough to justify admission control by RAC, partitioning shared data into different views can improve the performance of TM systems such as NOrec by reducing the contention in accessing the TM metadata.**

***Keywords*-View-Oriented Transactional Memory (VOTM), transactional memory, concurrency control, Restricted Admission Control (RAC), View-Oriented Parallel Programming (VOPP)**

I. Introduction

In transactional memory (TM) models, atomic access to shared data is achieved by transactions. To facilitate concurrency, all threads can freely enter a transaction, access shared objects, and hopefully commit the transaction at the end [1]. If a conflict occurs between concurrent transactions, one or more transactions will be aborted and rolled back. TM will undo the effects of the aborted transactions and restart them. When conflicts of accesses are rare, TM has very little rollback overhead and encourages high concurrency, since multiple threads can freely access different parts of the shared data. However, when conflicts are frequent (i.e., the contention is high), the rollback overhead is staggering, and TM becomes not scalable due to the large number of aborted transactions. In such situations, the lock-based approach becomes more efficient as it avoids the excessive operational overhead of transactions [2].

To address this contention problem, we had proposed the Restricted Admission Control (RAC) model [3]. RAC reduces contention by restricting the number of threads entering a transaction when the contention is high. In extreme cases, RAC can fall back to a lock-based mechanism to ensure progress. When contention is low, RAC allows threads to freely enter a transaction to maximize concurrency. However, a problem in this RAC approach is that, if there are two or more shared objects that are never accessed together in the same transaction, RAC will not perform effectively. Especially when one of the shared objects has high contention but the others have low contention, RAC would unnecessarily restrict access to all shared objects, including those with low contention. In such situations, RAC would hinder the concurrency of transactions that only access shared objects with low contention.

To improve performance in these situations, we proposed the View-Oriented Transactional Memory (VOTM) paradigm in our previous papers [2, 3]. VOTM is a variant of our View-Oriented Parallel Programming (VOPP) model [4–6] in transactional memory. In VOTM, shared memory is partitioned into "views" by the programmer according to the memory access pattern of the application. The size and the content of a view can be determined by the programmer as a natural part of the programming task. A view can be dynamically allocated and destroyed, but cannot overlap with other views. Before a view is accessed (read or written), it must be acquired (by simply using *acquire_view*). After a view is used, it must be released by using *release_view*). This data-centric model bundles concurrency control and data access together. Therefore, it relieves programmers from controlling concurrent data accesses directly with either locks or transactions, but leaves it to the system (such as RAC) to decide how to concurrently access a view.

In VOTM, RAC controls access to each view independently, according to the contention of the view. Shared objects that should be accessed together are put into the same view; whereas objects that are never accessed together in a transaction should be put into *different* views. Therefore, VOTM can ensure progress by restricting admission to views with high contention, while maximizing concurrency by allowing threads to freely access other views with low contention.

This paper has the following contributions. First, we extend the theoretical model of RAC [3] to illustrate that partitioning shared data into multiple views can further improve performance in VOTM.

Second, we extensively evaluate the VOTM model with microbenchmarks and real applications. We have used different transactional memory implementations to investigate in which cases and in what ways VOTM can improve performance.

The rest of the paper is organized as follows. Section 2 will present the RAC scheme and its theoretical model in VOTM. Section 3 will evaluate the VOTM model with experimental results from microbenchmarks and real applications. Section 4 will discuss related work and Section 5 concludes the paper.

II. VOTM and its implementation

As mentioned above, VOTM is based on the philosophy of VOPP [2] which partitions the shared memory into views.

The RAC scheme is implemented for every view in VOTM. Each view has an admission quota Q that restricts the maximum number of threads accessing the view concurrently. Before a

1530-2016/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPPW.2012.5

view is accessed, the primitive *acquire_view* is used. If Q is equal to 1, *acquire_view* is equivalent to a lock acquisition. In this case, lock mechanism is used instead of the transactional mechanism to avoid the transactional overhead. 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:

1) Compare P with Q. If P is smaller than Q, increase P by 1, start a new transaction, and return with success.
2) If P equals Q, the calling thread is blocked until P becomes smaller than Q, and then goes to Step 1.

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

1) 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.
2) 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 transactional 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.

The rest of this section will first provide a theoretical analysis of RAC and analyze performance gain of multiple views. Then, we will briefly describe our VOTM implementation and the programming interface. The VOTM programming interface can parallelize existing serial code easily with little instrumentation.

A. Theoretical analysis of the RAC model

In Sections II-A1 and II-A2, we first introduce the preliminary of the RAC model that has been obtained in our previous work [3], then we will extend the theoretical analysis on potential performance gain in multiple views in section II-A3.

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 [3]. 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.

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

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 $|\Delta|$ 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 $\Delta \geq 0$ and $makespan_{RAC}(S_T) \geq makespan_{TM}(S_T)$. 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) RAC with Q' threads vs. Q threads: Similar to the deduction of Equation 3, the difference between makespans of RAC using Q' (new) threads ($makespan_{RAC}(S_T, Q')$) and Q (previous) 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 $makespan_{RAC}(S_T, Q) > makespan_{RAC}(S_T, Q')$. Therefore, to reduce the execution time of the concurrent transactions, RAC should increase Q.

In summary, we have the following observation:

Observation 1. *If $\delta(Q)$ is larger than 1, Q should be decreased; if $\delta(Q)$ is smaller than 1, Q should be increased, in order to reduce the makespan of S_T in RAC.*

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) RAC in multiple views vs single view: We analyze the potential gain of performance in multiple-views scenario where RAC can separately control admission to each view according to its contention. It is compared with the scenario where RAC controls access to the entire transactional memory.

Assume the set of transactions $S_T = \{T_1, ..., T_n\}$ can be divided into two non-intersecting subsets $S_T^1 = \{T_1^1, ..., T_n^1\}$ and $S_T^2 = \{T_1^2, ..., T_n^2\}$, where transactions in S_T^1 access data in $Object_1$, and transactions in S_T^2 access data in $Object_2$. So, if $\delta^1 = \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{\sum_{i=1}^{n} t_i^1 (N-1)} > 1$ (high contention), $\delta^2 = \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{\sum_{i=1}^{n} t_i^2 (N-1)} \leq 1$ (low contention), and $Q^1 \leq Q \leq Q^2$, then the makespan of putting $Object_1$ and $Object_2$ into separate views with independent RAC $makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2))$ should be smaller than the makespan of a single view with RAC $makespan_{RAC}(S_T, Q)$:

$$\begin{aligned} &makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\ &\leq makespan_{RAC}(S_T, Q) \end{aligned} \tag{6}$$

The proof of Equation (6) is as follows.

$$\begin{aligned} &makespan_{RAC}(S_T, Q) \\ &= \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} \\ &= \frac{\sum_{i=1}^{n} c_i d_i}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i - \frac{\sum_{i=1}^{n} c_i d_i}{N-1}) \\ &= \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i^1 - \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{N-1}) \\ &+ \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i^2 - \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{N-1}) \\ &= makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q) \end{aligned} \tag{7}$$

Suppose view 1 has high contention, i.e., $\delta^1 = \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{\sum_{i=1}^{n} t_i^1 (N-1)} > 1$, and $Q^1 \leq Q$. Then,

$$makespan_{RAC}(S_T^1, Q^1) \leq makespan_{RAC}(S_T^1, Q) \tag{8}$$

Suppose view 2 has low contention, i.e., $\delta^2 = \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{\sum_{i=1}^{n} t_i^2 (N-1)} \leq 1$, and $Q \leq Q^2$. Then,

$$makespan_{RAC}(S_T^2, Q^2) \leq makespan_{RAC}(S_T^2, Q) \tag{9}$$

Therefore, we have

$$\begin{aligned} &makespan_{RAC}(S_T^1, Q^1) + makespan_{RAC}(S_T^2, Q^2) \\ &\leq makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q) \end{aligned} \tag{10}$$

Since the makespan of the multiple-view RAC is:

$$\begin{aligned} &makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\ &= makespan_{RAC}(S_T^1, Q^1) + makespan_{RAC}(S_T^2, Q^2) \end{aligned} \tag{11}$$

and the makespan of the single view RAC is:

$$\begin{aligned} &makespan_{RAC}(S_T, Q) \\ &= makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q) \end{aligned} \tag{12}$$

From Equation (10), we have:

$$\begin{aligned} &makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\ &\leq makespan_{RAC}(S_T, Q) \end{aligned} \tag{13}$$

Now we have this observation:

Observation 2. *If there are two shared objects, which are not required to be accessed together in the same transaction, and the first object has high contention ($\delta(Q)$ is larger than 1) but the second object has low contention ($\delta(Q)$ is smaller than 1), then the two objects should be put into separate views to reduce the makespan of RAC.*

In the experimental section, we will examine the multiple-view RAC model with the Eigenbench microbenchmark suite, and examine the performance of RAC over different TM implementations and different applications.

B. Implementation

The implementation of VOTM is based on RSTM-7.0 [7], a C++-based modular software transactional memory system where TM algorithms such as the encounter-time locking algorithm *OrecEagerRedo* and the commit-time locking algorithm *NOrec* [8] are implemented as plug-ins and can be chosen easily by reconfiguration. In VOTM, each view is essentially an independent TM system, and the access of each view is separately controlled by its own RAC. The implementation of the RAC algorithm had been described in our previous paper [3].

C. The VOTM programming interface

Figure 1 shows a C example to explain how to create a view for a linked list in VOTM. In the example, *create_view()* creates a view *vid* for the linked list, and *malloc_block()* allocates a memory block from the view for the list head.

```
1  typedef struct Node_rec Node;
2
3  struct Node_rec {
4    Node *next;
5    Elem val;
6  };
7
8  typedef struct List_rec {
9    Node *head;
10 } List;
11
12 List *ll_init(vid_type vid) {
13   List *result;
14   create_view(vid, size, 0);
15   result = malloc_block(vid, sizeof(result[0]));
16   acquire_view(vid);
17   result->head = NULL;
18   release_view(vid);
19   return result;
20 }
```

Figure 1. Code snippet of list initialization in VOTM [2]

Here, programmers can either let RAC to dynamically control access to the view by specifying a value smaller than 1 to the third argument of $create\_view()$. Alternatively, if the contention of the view is known to the programmer, the admission quota Q of the view can be statically set via the third argument.

A VOTM code snippet for list insertion is shown in Figure 2. Here the parameter *node* of the function points to a node that is a memory block belonging to the view of the linked list. Compared with the sequential version of the code snippet, the only extra code is the view primitives, *acquire_view()* and *release_view()*, that demarcate view access.

A summary of the VOTM API [2] is shown in Table I.

III. Performance evaluation

This experiment aims at verifying Observation 2 for applications in which shared data can be divided into multiple views. The applications include microbenchmarks from a modified version of the Eigenbench Suite [9] and the memory-intensive real TM application Intruder from the STAMP-0.9.10 benchmark suite [10].

```
1  void ll_insert(List *list, Node *node, vid_type vid) {
2    Node *curr;
3    Node *next;
4
5    acquire_view(vid);
6
7    if (list->head->val >= node->val) {
8      /* insert node at head */
9      node->next = list->head;
10     list->head = node;
11   } else {
12     /* find the right place */
13     curr=list->head;
14     while (NULL != (next = curr->next) &&
15            next->val < node->val) {
16       curr = curr->next;
17     }
18     /* now insert */
19     node->next = next;
20     curr->next = node;
21   }
22   release_view(vid);
23 }
```

Figure 2. Code snippet of list insertion in VOTM [2]

For each application, the following versions are implemented:

single-view
: the VOTM implementation with the entire shared data placed into a single view;

multi-view
: the VOTM implementation with shared objects placed into two separate views;

multi-TM
: similar to multi-view, except the access to each view is completely free without using the RAC mechanism;

TM
: the plain RSTM implementation.

In our experiments, if the multi-view version performs better than the single-view version, Observation 2 is verified. In order to verify that using multiple views can reduce contention on global TM metadata, we have implemented the multi-TM version. If the multi-TM version performs better than the TM version, we can conclude that using multiple views can help improve TM performance even if the RAC mechanism is not used.

Furthermore, to examine when and how VOTM with multiple views can improve performance in different TM systems, we have implemented two VOTM versions:

VOTM-OrecEagerRedo
: is based on the the encounter-time locking TM algorithm "OrecEagerRedo" (similar to TinySTM [11]), which is implemented in RSTM-7.0 [7].

VOTM-NOrec
: is based on the commit-time locking TM algorithm "NOrec" [8] which is also from RSTM.

A. Eigenbench

Eigenbench [9] can generate transactions using orthogonal parameters, and allows a better understanding of the behavior of a TM system by adjusting the parameters.

Table I
SUMMARY OF VOTM API [2]

void create_view(int vid, size_t size, int q)	Creates a view with ID *vid* and size *size*. *q* is the maximum number of processes admitted to this view. If *q* is less than 1, admission quota of this view will be dynamically managed by RAC.
void *malloc_block(int vid, size_t size)	Allocates a memory block with the specified *size* for the view *vid*. Returns the base address of the allocated block.
void free_block(int vid, void *ptr)	Frees the memory block pointed by *ptr* from the view *vid*.
void destroy_view(int vid)	Destroys the view *vid*.
void brk_view(int vid, size_t size)	Expands the memory space of the view *vid* by *size*.
void acquire_view(int vid)	Acquires read-write access to the view *vid*.
void acquire_Rview(int vid)	Acquires read-only access to the view *vid*.
void release_view(int vid)	Releases access to the view *vid*.

For example, contention in Eigenbench is controlled by adjusting the size of *hot_array* and the number of read and write accesses to the *hot_array*. High contention can be caused by large number of read-write accesses to a relatively small length of *hot_array*. The shared *mild_array* is also accessed by transactions, but each thread has its own subarray and therefore access to *mild_array* will not cause conflicts, but will increase transaction size and rollback overheads.

Long transactions can be generated by adjusting one or more of the following features:

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

In Eigenbench, a transaction is modelled by a sequence of reads/writes to the shared memory, interleaved with accesses to local memory and computation (represented by NOPs). There are also accesses to local memory and computations outside transactions in Eigenbench.

In our experiments, we have modified the Eigenbench program to have two views, each view has its own hot_array, mild_array and parameters concerning the number of read/write accesses and the number of NOPs in each transaction that access the view.

The modified Eigenbench program will execute a number of iterations, which is the total number of transactions specified for each view. Each iteration accesses one of the two views randomly, followed by the activities outside transactions. The pseudocode outlining the modified Eigenbench application is shown in Figure 3, and parameters used in Eigenbench are shown in Table II.

```
1
2   struct View_data {
3       /* shared array where conflict occurs,
4          accessed in tx */
5       shared word hot_array[A1];
6
7       /* shared array where each thread accesses
8          its own subarray, so does not cause
9          conflict, but still needs rollback
10         should tx be aborted */
11      shared word mild_array[A2];
12
13      /* private to each thread, can be accessed
14         either inside or outside tx.
15         if accessed inside tx and tx aborted
16         then needs to roll back changed */
17      thread_local word cold_array[A3];
18  };
19
20  View_data views[2];
21
22  each thread:
23
24  for loops:
25  do
26      acquire view 1 or 2 randomly
27      in acquired view:
28      perform
29         r1 reads and w1 writes to the shared hot_array, and
30         r2 reads and r2 writes to the shared mild_array
31         in *random order*
32         each access touches a random element (word) in
33         the shared hot_array, or in the
34         dedicated subarray within the shared mild_array
35
36         between two accesses to shared arrays, there will
37         also be r3i reads and w3i writes to the thread-local
38         cold array, and NOPi instructions
39      release view
40
41      /* activities outside transactions:
42      perform r3o reads and w3o writes to the
43          thread-local array
44      perform NOPo instructions
45  done
```

Figure 3. Pseudocode of the modified Eigenbench application

In the "multi-view" version, each thread executes 100000 transactions that access view 1 (the high contention view) and 100000 transactions that access view 2 (the low contention view), with the accesses interleaved randomly. View 1 is set to be accessed by long transactions with high contention, with each transaction accessing many elements in a small *hot_array*; whereas view 2 is accessed by long transactions with low contention.

In the "single-view" version, each thread executes 200000 transactions. In each iteration, after the view is acquired, the transaction can access either object 1 (with high contention) or object 2 (with low contention). Accesses to object 1 and 2 have the same patterns as the accesses to view 1 and 2 in the "multi-view" version.

B. Intruder

Intruder from the STAMP benchmark suite is a memory-intensive TM application. In this application, a dictionary is used to store partial results, and jobs are handled by a

Table II
EIGENBENCH PARAMETERS FOR THE MULTI-VIEW MICROBENCHMARK

View	1	2
N	16	
loops	100k	100k
A1	256	16k
A2	16k	16k
A3	8k	8k
R1	80	10
W1	20	10
R2	10	10
W2	10	10
R3i	0	5
W3i	0	1
NOPi	0	20
R3o	0	
W3o	0	
NOPo	0	

centralized task queue. Since the task queue and the dictionary are never accessed together in the same transaction, they are allocated in separate views in the "multi-view" version. Default parameters "-a10 -l128 -n262144 -s1" are used.

C. Experimental Results

All tests are carried out on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running with 2.5GHz 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 -O3.

As mentioned previously, $\delta(Q)$ in Observations 1 and 2 is estimated with Equation 5. *rdtsc()* is used to measure the CPU cycles spent in aborted transactions and successful transactions.

The rest of this section will show the results on VOTM-OrecEagerRedo and VOTM-NOrec.

1) VOTM-OrecEagerRedo: In this part of the experiment, the "single-view" version of Eigenbench, which has a high contention object and a low contention object placed in the same view, and the real TM application Intruder, are run with the admission quota Q fixed to 1, 2, 4, 8 and 16 respectively, without dynamic adjustment during runtime. In all tests, the total number of threads (N) is fixed to 16, and thus the $Q = 16$ case is equivalent to the conventional TM that has no restriction of admissions.

Table III
SINGLE-VIEW EIGENBENCH WITH VOTM-ORECEAGERREDO

Q	1	2	4	8	16
Runtime(s)	63.8	65.7	241.2	2698	livelock
$\#abort$	0	7.01m	178m	5.26G	livelock
$\#tx$	3.2m	3.2m	3.2m	3.2m	3.2m
$CPUcycles_{aborted\_tx}$	0	101G	2.08T	49.8T	livelock
$CPUcycles_{successful\_tx}$	145G	205G	216G	231G	livelock
$\delta(Q)$	N/A	0.49	3.21	30.7	livelock

In the single-view version of Eigenbench, Table III confirms that from $Q = 4$ to $Q = 16$ (livelock), the contention is high and $\delta(Q) > 1$, which suggests that we should decrease Q according to Observation 1, and the runtime is shorter by lowering Q at these ranges. Therefore, Observation 1 is correct for $Q = 4$, $Q = 8$ and $Q = 16$. At $Q = 2$, $delta(Q)$ is much smaller than 1, which suggests that Q should not be further reduced according to Observation 1. However, the runtime is slightly decreased when Q is further reduced to 1. This could be attributed to the fact that at $Q = 1$, RAC falls back to the lock-based mode and TM mechanism is not used to access the shared memory. Since the TM overhead is removed, the performance is further improved. However, Observation 1 has not taken this special optimization into account.

Table IV
SINGLE-VIEW INTRUDER WITH VOTM-ORECEAGERREDO

Q	1	2	4	8	16
Runtime(s)	113	91.3	47.6	25.3	17.4
$\#abort$	0	3.10k	7.31m	10.5m	14.4m
$\#tx$	23.4m	23.4m	23.4m	23.4m	23.4m
$CPUcycles_{aborted\_tx}$	0	6.53G	19.9G	49.3G	100G
$CPUcycles_{successful\_tx}$	124G	287G	291G	299G	311G
$\delta(Q)$	N/A	0.02	0.02	0.02	0.02

In the single-view version of Intruder, At every Q, $\delta(Q)$ is much smaller than 1, which suggests that Q should be increased according to Observation 1. Since $Q = 16$ has the shortest runtime, Observation 1 is true in the single-view version of Intruder.

Now we compare the single-view and the multi-view versions of Eigenbench.

In order to test how RAC behaves in view 1 (with high contention), we manually set Q_2 of view 2 (with low contention) to 16 which is the optimal value suggested by Observation 1. However, Q_1 is manually set to different values to test whether Observation 1 is correct for individual views.

In the following tables, the individual statistics of each view, including the number of aborts, transactions (tx), CPU cycles spent in aborted and successful transactions, and $\delta(Q)$ are shown. The subscript indicates the identity of the view (for example, Q_1 indicates Q for view 1). Though Q_2 of view 2 is always set to 16, when Q_1 is set to different values, the statistics of view 2 are changed accordingly in the tables.

Table V
MULTI-VIEW EIGENBENCH WITH VOTM-ORECEAGERREDO

Q_1	1	2	4	8	16
Runtime(s)	24.1	75.0	306	3276	livelock
$\#abort_1$	0	18.3m	246m	6.57G	livelock
$\#tx_1$	1.6m	1.6m	1.6m	1.6m	livelock
$CPUcycles_{aborted\_tx_1}$	0	268G	2.83T	61.3T	livelock
$CPUcycles_{successful\_tx_1}$	52.7G	93.5G	104G	118G	livelock
$\delta(Q_1)$	N/A	2.87	9.06	74.2	livelock
$\#abort_2$	25.2k	6.94k	1.58k	178	livelock
$\#tx_2$	1.6m	1.6m	1.6m	1.6m	livelock
$CPUcycles_{aborted\_tx_2}$	1.16G	320m	74.7m	8.48m	livelock
$CPUcycles_{successful\_tx_2}$	116G	116G	116G	118G	livelock
$\delta(Q_2)$	N/A	0.003	0.0002	0	livelock

For the multi-view Eigenbench, Table V shows that $\delta(Q_1)$ is larger than 1 for all Q_1, which suggests that Q_1 should be decreased to 1, which is the actual optimal value of Q_1. Therefore, in this case, Observation 1 is correct. In addition, the optimal runtime of the multi-view version (24.1s) is shorter

Table VI
PERFORMANCE OF ADAPTIVE RAC IN VOTM-ORECEAGERREDO

	single-view			multi-view				multi-TM		TM	
Application	time(s)	Q	#abort	time(s)	Q1	Q2	#abort	time(s)	#abort	time(s)	#abort
Eigenbench	65.1	2	7.52m	24.8	1	16	1.07m	livelock		livelock	
Intruder	17.7	16	18.2m	17.4	16	16	49.5m	17.2	14.2m	17.3	15.0m

than the optimal runtime of the single-view version (63.8s). Therefore, Observation 2 is correct since the runtime of multi-view is shorter than the runtime of single-view.

Table V also shows that $\delta(Q_2)$ and the number of aborts in view 2 decreases as Q_1 increases. This is expected because as Q_1 increases, contention of view 1 increases, and therefore wasted time in aborted transactions accessing view 1 will also increase. As a result, the time spent outside view 2 will increase. Therefore the contention of view 2 will decrease.

In the multi-view version of Intruder, where both Q_1 and Q_2 are set to 16, The runtime is 17.4s. Both $\delta(Q_1)$ and $\delta(Q_2)$ are equal to 0.01, indicating little contention in both views, thus both $Q_1 = 16$ and $Q_2 = 16$ (the number of threads) are optimal. Since both views have low contention, the multi-view version has only slight performance advantage over single-view, as shown in Table VI.

Table VI shows how RAC can adapt to contention situation and adjust Q to the optimal value. For Eigenbench, RAC prevents livelock in both single-view and multi-view by restricting Q. In single-view, access to data in object 2 (with low contention) is unnecessarily hindered by RAC, as RAC can only consider overall contention as a whole, and set the overall Q to 2. However in multi-view, RAC is able to accurately set Q of view 1 (with high contention) to 1, without hindering access to view 2, hence gives a 200% performance improvement over single-view.

In Intruder, the very low values of $\delta(Q)$ (0.02 for single-view at $Q = 16$ and 0.01 for multi-view at $Q_1 = Q_2 = 16$) suggest that the contention is not high enough to justify reducing the value of Q in both single-view and multi-view. Anyway, Table VI shows the adaptive RAC has correctly set Q to the maximum in all cases. In addition, compared with multi-TM, multi-view shows little extra overhead from the RAC mechanism in VOTM.

2) VOTM-NOrec: In this part of the experiment, Q is fixed to 1, 2, 4, 8 and 16 respectively without dynamic adjustment during runtime.

Table VII
SINGLE-VIEW EIGENBENCH WITH VOTM-NOREC

Q	1	2	4	8	16
Runtime(s)	64.0	46.1	35.1	34.5	33.6
$\#abort$	0	648k	2.91m	8.25m	14.0m
$\#tx$	3.2m	3.2m	3.2m	3.2m	3.2m
$CPUcycles_{aborted\_tx}$	0	27.5G	141G	431G	718G
$CPUcycles_{successful\_tx}$	146G	188G	192G	196G	205G
$\delta(Q)$	N/A	0.15	0.25	0.31	0.23

Table VIII
SINGLE-VIEW INTRUDER WITH VOTM-NOREC

Q	1	2	4	8	16
Runtime(s)	113	86.7	55.1	52.7	49.3
$\#abort$	0	338k	1.01m	1.84m	5.21m
$\#tx$	23.4m	23.4m	23.4m	23.4m	23.4m
$CPUcycles_{aborted\_tx}$	0	10.8G	54.9G	208G	611G
$CPUcycles_{successful\_tx}$	123G	262G	346G	601G	1.39T
$\delta(Q)$	N/A	0.04	0.05	0.05	0.03

In the single-view version of both Eigenbench and Intruder, Table VII and VIII show that $\delta(Q)$ is much smaller than 1 in all cases. Therefore, according to Observation 1, Q should not be decreased. The results of runtime show that Observation 1 is correct for both applications, since the runtime of $Q = 16$ is the shortest for both cases. The runtime of $Q = 16$ for VOTM is similar to the runtime of the original TM system (see Table X), showing that the extra overhead of RAC is low.

Now we examine the the performance of RAC in multiple views. As explained before, Q_2 in Eigenbench is fixed to 16, as view 2 has low contention and does not need access restriction. However Q_1 is manually set to different values to test whether Observation 1 is correct for individual views.

Table IX
MULTI-VIEW EIGENBENCH WITH VOTM-NOREC

Q_1	1	2	4	8	16
Runtime(s)	24.1	32.7	32.3	31.7	30.2
$\#abort_1$	0	1.60m	4.60m	9.73m	14.6m
$\#tx_1$	1.6m	1.6m	1.6m	1.6m	1.6m
$CPUcycles_{aborted\_tx_1}$	0	79.5G	233G	487G	682G
$CPUcycles_{successful\_tx_1}$	52.5G	73.8G	73.8G	75.7G	78.4G
$\delta(Q_1)$	N/A	1.07	1.05	0.92	0.58
$\#abort_2$	7.46k	5.14k	5.25k	5.38k	5.69k
$\#tx_2$	1.6m	1.6m	1.6m	1.6m	1.6m
$CPUcycles_{aborted\_tx_2}$	335m	221m	226m	234m	251m
$CPUcycles_{successful\_tx_2}$	109G	108G	108G	108G	108G
$\delta(Q_2)$	N/A	0.002	0.0001	0.0003	0.0002

In Eigenbench, Table IX shows that at $Q_1 = 16$, $\delta(Q_1)$ is much smaller than 1, therefore it does not need to reduce Q_1, and the actual runtime of $Q_1 = 16$ is indeed slightly shorter than Q_1 of 2, 4 and 8. However, $\delta(Q_1)$ increases to around 1, for $Q_1 = 2$ and $Q_1 = 4$. In these cases, Observation 1 indicates an increase of contention when Q is decreased, and does not quite reflect the real situation. One reason is that, in NOrec, time wasted in ultimately-aborted transactions is minimal because conflicts will be detected at the next read operation, and thus the theoretical model of Observation 1 does not match well with the handling of aborted transactions in NOrec. Another reason is that, since NOrec takes cares of the contention control which has not been taken into account

by the approximation Equation 5, the CPU cycles by aborted transactions cannot truly reflect the time wasted by aborted transactions in NOrec. Further research is required to refine the theoretical model to accommodate this issue. This issue will be further discussed later in this section.

Table IX also shows the runtime of $Q_1 = 1$ (24.1s) is much shorter than the other Q_1 values set to 2-16. As mentioned before, this performance improvement at $Q_1 = 1$ can be attributed to switching to lock-based approach at $Q = 1$, which avoids the TM overhead. Therefore, as mentioned in Section II-C, the programmer can optimize performance by manually setting the Q of a view to 1 if access of the view is known to be highly contentious.

In the multi-view version of Intruder, at $Q_1 = Q_2 = 16$, the runtime is 30.7s, as shown in Table X. Since both $\delta(Q_1)$ and $\delta(Q_2)$ are equal to 0.0004, there is little contention in both views and $Q_1 = Q_2 = 16$ has the best performance.

Table X also show how adaptive RAC performs in VOTM-NOrec for both Eigenbench and Intruder.

In Eigenbench, the contention is not high enough to justify the decrease of Q by RAC in both single-view and multi-view versions, and RAC performs correctly by not restricting the admission quota Q in both cases. However there is 10% performance improvement in the multi-view version over single-view and NOrec, even when none of the views is restricted to admission. In addition, with RAC disabled, the multi-TM version also gives similar improvement over NOrec. This performance improvement is because of splitting data into two views which reduces contention on the global clock in NOrec. In VOTM such as multi-view and multi-TM, each view is essentially a separate TM system and has its own global clock.

For Intruder, Table X shows that in both single-view and multi-view, the values of Q of all views are settled at 16. RAC performs correctly here, as the very low $\delta(Q)$ values at $Q = 16$ mentioned before suggest the contention is not high enough in single-view and multi-view.

Although Q stays at 16 in all cases, multi-view has a runtime of 30.7s, which has a 60% improvement over single-view, and 55% improvement over NOrec. multi-view has similar performance as multi-TM, and both are faster than single-view and NOrec. The above results show that splitting shared data into multiple views can improve performance for even the most efficient TM system NOrec. The performance gain is from the reduction of contention on the global clock, as discussed above.

It is worth noting there is a slight runtime improvement in multi-view over multi-TM in both Eigenbench and Intruder, and the number of aborts in the multi-TM version in both cases are slightly higher. This could be attributed to the extra overhead of RAC causing a small delay in restarting an aborted transaction, which reduces the contention slightly on the global clock. However, we don't have experimental evidence to support this claim yet. We need further microscopic investigation on the results.

D. How RAC can improve performance

The microbenchmark illustrates that in highly-contentious situations, RAC can improve performance by preventing live-locks in encounter-time locking algorithms [11, 12] such as OrecEagerRedo. In encounter-time locking algorithms, a transaction locks a shared memory location at the first write operation. Then when it is accessed by other transactions before the owner transaction commits, a conflict is detected, and one or more transactions will abort and restart. However these restarting transaction may cause further conflicts with the originally-winning transaction before it commits, and may cause the originally-winning transaction to ultimately abort. This can become a vicious cycle and eventually no transactions can successfully commit. As a result, no progress is made, and this is known as livelock. However in this situation, $\delta(Q)$ will rise very quickly, and RAC will promptly drive Q down to a very low value, such as 1, thus ensuring progress. In the case of multiple views, if one of the views livelocks, RAC can restrict access to that view without affecting concurrency of the other views. In this way, using multiple views in VOTM can considerably improve performance over the single-view version, as demonstrated in the microbenchmark.

Commit-time locking TM (CTL) algorithms such as NOrec [8] locks an address at commit time. Therefore, in CTL algorithms, conflicts are generally detected at commit-time. When a transaction detects a conflict when it tries to commit, it may abort and restart. However, since those transactions that it has conflicts with can be committed successfully, the aborted transaction is impossible to have conflicts with those committed transactions again. In this way, CTL algorithms can avoid livelock.

If conflicts are detected at commit time, then the time between the occurrence of a conflict (i.e. a location in its readset is written to by another committing transaction while the current transaction is still running) and the commit of the current transaction could be wasted in a transaction that ultimately fails, although the time wasted will be limited to the aborted transaction itself. This is different from the case of livelocking in ETL algorithms where the wasted time can involve multiple other transactions.

However in some advanced CTL algorithms such as NOrec, the readset is validated at every read operation after the first write operation in a transaction. Therefore, conflict will be quickly detected at the next read operation after the occurrence of the conflict. As a result, the time wasted by transactions that ultimately abort will be very little. Since the aim of RAC is to restrict access to save this wasted time, the benefit of RAC in NOrec diminishes, as shown in the microbenchmark and Intruder.

However the cost of NOrec is that its frequent read-set validation causes considerable contention at the global clock in its metadata. Therefore, it performs worse than OrecEagerRedo in memory-intensive applications such as Intruder.

For NOrec, VOTM can help in two ways:

- Reduce contention on global clock by splitting shared objects into multiple views, where each view is essentially

Table X
PERFORMANCE OF ADAPTIVE RAC IN VOTM-NOREC

	single-view			multi-view				multi-TM		TM	
TM Algorithm	time(s)	Q	#abort	time(s)	Q1	Q2	#abort	time(s)	#abort	time(s)	#abort
Eigenbench	33.7	16	14.1m	30.2	16	16	14.1m	30.5	14.2m	33.7	14.1m
Intruder	52.6	16	5.2m	30.7	16	16	1.13m	30.9	1.20m	47.8	5.0m

a separate TM system with its own global clock.
- For a view that is accessed by memory-intensive but short transactions (i.e. little computation work), performance can be optimized by manually setting Q of that view to 1, which allows RAC to fall back to lock-based mode, thus removing the high TM overhead.

IV. RELATED WORK

Generally, approaches to concurrency control can be classified into two types: transactional scheduling and adaptive locks. We will also discuss work on adaptive transactional memory.

A. Transactional scheduling

Transactional scheduling can control the admission of transactions when contention is high. It can prevent conflicts before they occur, therefore reducing wasted work on aborted transactions. For example, transaction scheduling algorithms such as [13] 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. A similar approach is adopted in [14], 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 [15] also adopts a similar approach.

All transaction scheduling algorithms described above treat the entire TM with the same scheduling decision. Therefore, access to objects of low contention can be unreasonably restricted due to the high contention of other objects in TM. Also the statistics collected for the entire TM are not as accurate as those collected per view basis. Since RAC in VOTM treats each view individually, the estimation of $\delta(Q)$ is more accurate.

B. Adaptive locks

The speculative lock elision (SLE)-based model [16] 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 [17] 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. These models either allow all threads to hold the elidable lock in speculative mode, or only allow exclusive access to one thread during non-speculative (mutex) mode. However, as shown in [3], 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 the optimal concurrency rather than only choosing between the two extremes – exclusive access to one thread or admitting all threads.

C. Adaptive Transactional Memory

There are also TM systems, such as [18] by the RSTM group, that choose a TM algorithm at runtime according to the access pattern and contention situation of the transactional memory. These adaptive TM systems use machine learning methods such as decision trees and neural network to learn from a training set of microbenchmarks and TM algorithms, to create an executable adaptive policy. Then, when a real application is run, "profiles" are taken at some pre-defined events such as thread creation/destruction and consecutive aborts. These profiles are subsequently used to compare with the adaptive policies to select the best TM algorithm on the fly. For example, when the contention increases, the system can switch to a more pessimistic algorithm.

Adaptive TM is orthogonal to VOTM. It can be adopted by VOTM, where different views can have different access patterns, and therefore have different optimal TM algorithms. We will investigate this area in the near future.

V. CONCLUSIONS AND FUTURE WORK

As shown in the experimental results, VOTM can effectively prevent livelocks in encounter-time locking TM systems. Moreover, by partitioning shared data with different access patterns into different views, VOTM can further improve performance by allowing RAC to separately control access to each view according to its contention. Therefore, RAC can easily restrict access to a view with high contention without unnecessarily restricting access to other views with low contention. In this way, VOTM can optimize concurrency control and provide a better performance to single-view applications. It has also a better performance than the TM systems without RAC in situations of high contention.

The RAC mechanism alone cannot improve performance of NOrec dramatically, since NOrec is livelock-free and its algorithm limits the time wasted by ultimately-doomed transactions. However, NOrec can have high contention in its global clock for memory-intensive applications. The memory-intensive application Intruder shows that VOTM can improve NOrec performance by partitioning shared data into multiple views, since it can reduce contention on the global clock by using a separate TM system for each view.

In the near future, we will refine the RAC model to take better account for TM overhead in different TM systems. We will further investigate the potential benefits of applying adaptive TM algorithms in each view of VOTM. In addition, we will also compare VOTM to other transactional scheduling and adaptive lock systems to identify performance gains and overheads of different applications.

REFERENCES

[1] J. R. Larus and R. Rajwar, *Transactional Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan and Claypool, 2007.

[2] K. Leung and Z. Huang, "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.

[3] K.-C. Leung, Y. Chen, and Z. Huang, "Restricted Admission Control in View-Oriented Transactional Memory," *The Journal of Supercomputing*, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11227-011-0733-y

[4] Z. Huang, S. Cranefield, M. K. Purvis, and C. Sun, "View-Based Consistency and its implementation," in *First IEEE International Symposium on Cluster Computing and the Grid*, 2001, pp. 74–81.

[5] J. Zhang, Z. Huang, W. Chen, Q. Huang, and W. Zheng, "Maotai: View-Oriented Parallel Programming on CMT processors," in *Proceedings of the 37th International Conference on Parallel Processing*, 2008, pp. 636–643.

[6] Z. Huang, M. Purvis, and P. Werstein, "Performance evaluation of View-Oriented Parallel Programming," in *Proceedings of the 34th International Conference on Parallel Processing*. Oslo: IEEE Computer Society, June 2005, pp. 251–258.

[7] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," in *The First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2010, pp. 67–78.

[9] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "Eigenbench: A simple exploration tool for orthogonal TM characteristics," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5648812

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[11] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 237–246.

[12] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *20th International Symposium on Distributed Computing*, September 2006.

[13] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2008, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378564

[14] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2009, pp. 7–16.

[15] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, and I. Watson, "Adaptive concurrency control for transactional memory," University of Manchester, Tech. Rep., 2007.

[16] A. Roy, S. Hand, and T. Harris, "A runtime system for software lock elision," in *Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2009, pp. 261–274. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519094

[17] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency," in *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009.

[18] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear, "A transactional memory with automatic performance tuning," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 54:1–54:23, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086733